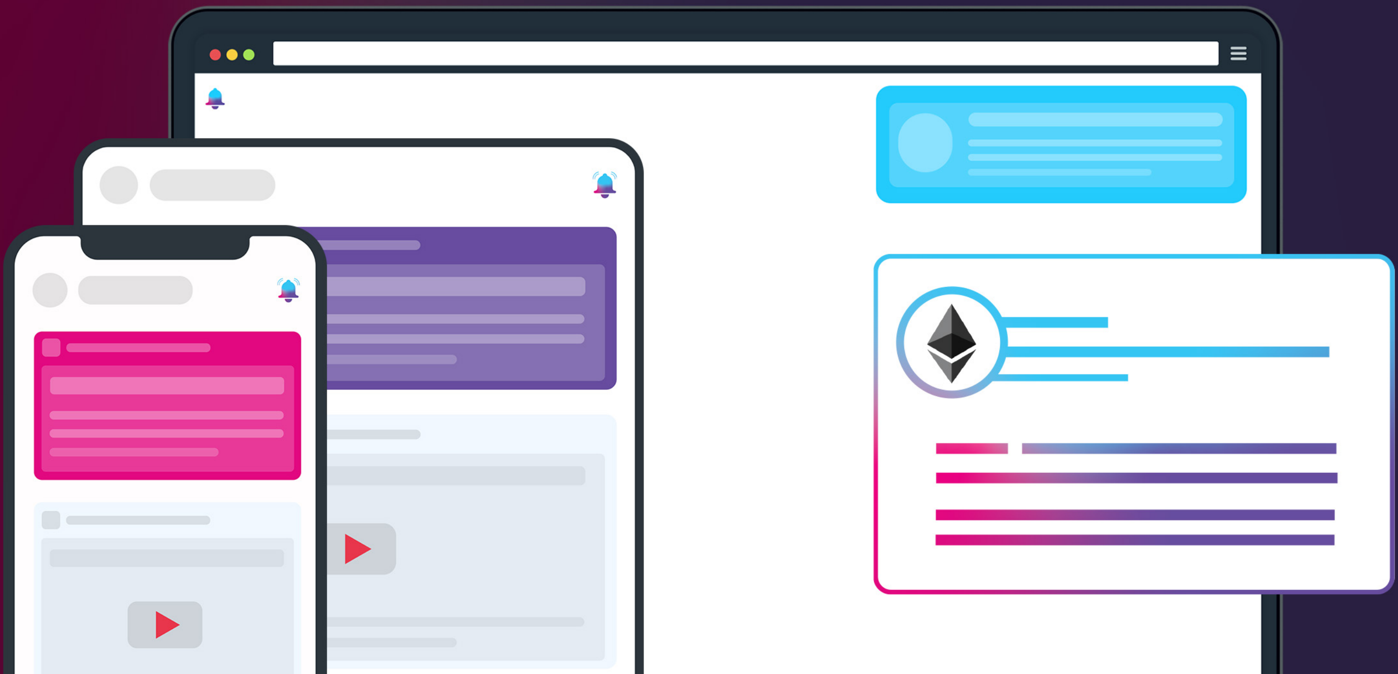
EPNS

Subscribe. Notify. Earn.

ETHEREUM PUSH NOTIFICATION SERVICE

WHITEPAPER V1.0 DRAFT



Ethereum Push Notification Service (EPNS)

Abstract

The document introduces a decentralized notifications protocol that enables wallet addresses to receive notifications in platform agnostic fashion from both decentralized and centralized carriers. It also explores and describes728478 the theory and technical aspects of the protocol / platform and the game theory that the protocol utilizes to ensure incentives for good actors in the ecosystem.

Authors: Harsh Rajat, Richa Joshi

founders@epns.io

Whitepaper Draft Version 1.0 | June 2020
Last Updated: 26 July, 2020

Index

Introduction

Introduction

The blockchain space is growing at an extremely rapid pace and the exponential growth is projected to continue rapidly in terms of users, services and revenue [1]. Despite this growth and expanding usage of blockchain tech, the services (dApps, services, smart contracts) still lack a genuine communication medium with their users which is sometimes filled by alternative communication mediums like twitter, telegram or email defeating the purpose of web 3.0.

More often though, dApps, smart contracts or services assume that users will come to them. This method is very similar to the early 2003 era of the internet where users were expected to perform an action, come back later and check the results of those actions (Gmail, Orkut, etc).

While this was okay with early internet days and web 2.0, it is not the case with traditional services now. In fact, no web 2.0 service really expects the user to come to them now, instead, they reach out to their users informing them about certain important events or any further actions required from users end. Modern push notification played a crucial role in this transition and has become a backbone for all web 2.0 services now [2].

However, for web 3.0, there still **doesn't exist a notification mechanism** that can notify users(wallet addresses) of important updates, events, actions, etc. This flawed mechanism has already led to pain points and side effects:

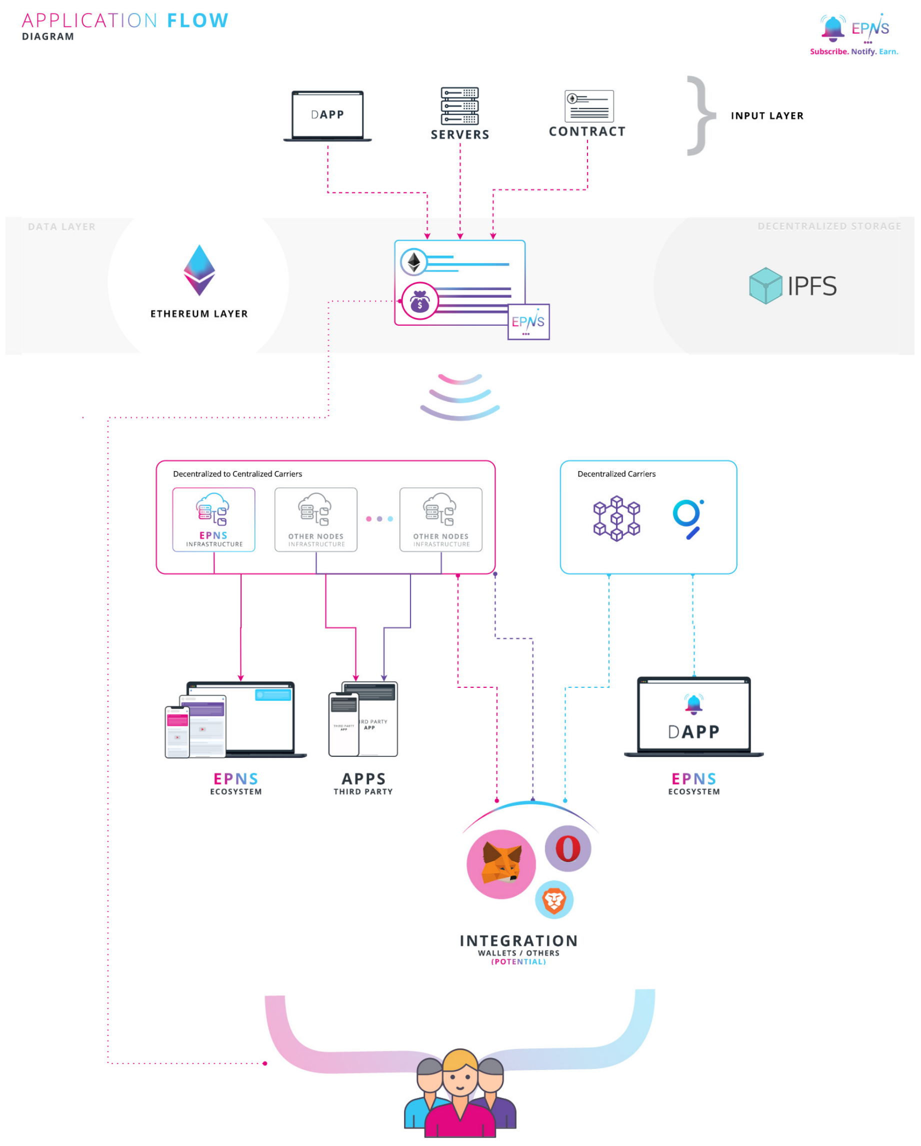
- Important events or user action requirements are missed completely (trading completed on dEx, liquidation alert on DeFi protocols, etc).
- Blockchain domains expiry have to be put on twitter in the hopes that the grace domain user might read it.
- A protocol getting compromised means sending information through Twitter and Telegram hoping the users of that protocol become aware of the vulnerabilities.

This is a major issue in adoption and the problem will worsen more as the services keep on growing on blockchain.

This paper describes the solution to these problems and aims to provide the missing piece of web 3.0 infrastructure with the introduction of the decentralized notification protocol. The protocol will enable users to receive notifications, be in complete control of the notifications they receive as well as enabling a passive means to earn from those notifications.

Protocol / Product Flow

EPNS uses the following flow to ensure storage, broadcasting and sending notifications in a platform agnostic and decentralized way.



High Level Application Flow of EPNS protocol / product

Notification is stored and treated like JSON payload which is transformed as per the rules of the different carriers when the notification reaches them. JSON Payload can differ with payload types which ensures the flexibility of the content, data, storage interpretation and delivery. This helps in creating different rules and content interpretation of the notification (for example: carrying images, call to action, live videos, etc)

The protocol allows users be in direct control of what services they get notification from, it imposes rules on the services including spam protection for users, limiting their ability to add wallets as subscribers, etc.

The protocol incentivizes users who receive notifications.

This on-chain abstraction of data enables delivery to centralized as well decentralized carriers, notifications are treated more like a social feed (e.g. Twitter) than an ephemeral piece of information (though means to do so also exists).

It also enables rules, incentives, settings and configuration to be retrieved from a single source of truth and is not dependent on a single point of failure.

Storing the JSON payload on decentralized storage and just it's pointer / hash on on-chain logs enables cost optimization. Though the protocol also allows storing the entire payload on-chain for services that intend to do so.

This can be further optimized by moving parts of these mechanisms to the layer-2 (L2).

> Abstracting the data layer on chain (directly or indirectly) ensures notifications are platform agnostic and available on decentralized mediums as well (e.g. dApp, wallets that might not want to trust a central point of truth).

Basic Definitions

The following definitions are used in the rest of the whitepaper to refer to certain roles. Most of the terminologies when it comes to services, users and subscribers are very similar to YouTube.

EPNS Related

Role	Description
Service	Any dApp / smart contract / centralized service / etc who wishes to send notifications.
Channel	Any service that has activated themselves on the protocol and thus can send notifications to their subscribers.
Subscriber	The user who have subscribed to a particular channel.
Users	Any user who is present in protocol registry.
Stake Pool	The pool of staked fee charged when a **service** activates themselves as a **channel.**
Interest Pool	The interest generated by the stake pool meant for distribution among subscribers of a channel in a weighted ratio favoring early subscribers.
Fee Pool	The fee earned by the protocol during certain actions, i.e. micro fees when sending notification, part of the penalty, etc.
Earning Reserve	The user earnings that haven't been moved to the user wallet and still present and mapped to the user in the protocol.

Other Concepts

Role	Description
IPFS	The InterPlanetary File System (**IPFS**) is a protocol and peer-to-peer network for storing and sharing data in a distributed file system. [3]
JSON payload	JavaScript Object Notation is used across the ecosystem for storage of data meant for consumption at frontend.

Unless explicitly mentioned, within the context of this whitepaper, the terms **contract owners, services, channels** or **users** always means **wallet addresses** that are anonymous.

Specs and Architecture

Specifications

EPNS is a notification protocol at its heart. We see notifications as the means to communicate information which can be of different types, carry different utilities and perform different tasks as per their use cases. To enable this, we assign each notification payload a payload type that defines certain characteristics of both the data they carry and the medium of storage.

Besides the flexibility of notification payload, we also see the rules established between **channels** and their **subscribers** to also have an impact on their utility and use cases.

Keeping in mind with the above analogies, the following specifications are rolled out to ensure that the specifications and the payload determine the use case of the notification rather than treating them as plain standard ones.

Some of the use cases we see opening up by this approach are:

- To allow users to receive notifications with different content type (images, call to actions, videos, etc.) .
- To allow future content or payload to be different and be interpreted differently.
- Makes interpreting payload storage flexible as changing payload type can indicate the storage medium (on-chain, ipfs, other storage solutions).
- To allow service to create channel that any user can subscribe to.
- To allow channels to approve users as subscribers and create business cases around them.
- To allow channel and users to approve each other before information is shared between them.

Channel Payload Specs

Creating a channel requires sending a JSON payload containing information about the channel to a decentralized storage. This JSON payload is uploaded to a decentralized storage solution (IPFS at the time of writing) which is emitted on chain to ensure that channel meta data can be constructed from the said decentralized storage.

> Uploading the payload is easily taken care by our dApp or any frontend solutions. While provisions to do it on smart contract directly exists (with pre-filled IPFS hash), it's recommended to do it on frontend due to it being a one time process.

Channel JSON Payload

Parameter	Description
name	Your Channel name (Recommended Limit: 40 Chars)
info	Short Description of your channel (Recommended Limit: 240 Chars)
url	Your Channel's website (Recommended Limit: 160 Chars)
icon	Base64 encoded image (Recommended Limit 128x128)

Example

```
{
  "name": "ENS (Ethereum Name Service)",
  "info": "ENS offers a secure & decentralised way to address resources bo
  "url": "https://ens.domains/",
  "icon": "data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAIAAAACACAIAAABMX
}
```

The hash / pointer of the payload is recorded on-chain along with the payload type. This payload type defines how to interpret what storage solution is used and how to retrieve its content on frontend.

Notification Payload Specs

Each notification sent to the protocol is essentially a JSON payload, bytes data or hash / pointer of the JSON payload. The protocol distinguishes payloads content and the storage medium it is on by assigning payload types to each notification.

> The JSON Payload can differ with payload types ensuring flexibility of the content, data, storage interpretation and delivery.

Notification JSON Payload

Parameter	Description
notification	[Required] Represents the notification typically delivered on the home screen of the platform (mobile, tablet, web, etc.), the icon of the channel is automatically added to outline where the notification is coming from.
title	[Required] The title of the message displayed on the screen, this differs from the **data json** because the title while transforming the payload can be different than the title presented. For example, a secret notification title is always transformed to say **Channel has sent you a secret notification.**
body	[Required] The body of the message displayed on the screen, this differs from the **data json** because the title while transforming the payload can be different than the title presented. For example, a secret notification body is always transformed to say **Please open the dApp / app to view your notification.**
data	[Optional] The data field present here forms the visual **feedBox** for the user. It indicates the data field the payload will carry. This allows the notification to transform according to the payload type and the content defined on the platform frontend (i.e. app, dApp, wallet, etc).

type	[Required] Each payload has a type which tells how the data should be interpreted, this type is mirrored on the protocol function call as well.
secret	[Optional] is required for certain payload types to decrypt the data.
asub	[Optional] is the subject shown in the feed item.
amsg	[Optional] is the message shown in the feed item, has rich text formatting.
acta	[Optional] is the call to action of that feed item.
aimg	[Optional] is the image shown in the feed item, this field is also capable of carrying **other media** links.
atime	[Optional] time in epoch when the notification should be displayed, if present, the frontend should respect this field and delay the notification till the schedule is reached. If the time is before the current time, the notification is treated as to be dispatched and displayed immediately.
recipients	[Optional] When presented with appropriate payload type allows notification to be delivered to many subscribers (but not all subscriber) of that channel.

If no **data** is carried in the **payload** (or only **atime** is carried), it is assumed that the notification is not important and hence **persist (or appearance)** in the frontend **feedBox** of the user.

Payload Types

The following are few payload types that are already defined, though they can change as we move forward. Notification payload type for EPNS is infinitely extensible and opens a huge range of possibilities including **multi-factor authentication, payments, blacklisting address (Multi-sig contract as a channel with exchanges as their subscribers), etc.** The data defined in the JSON payload they carry is used to interpret and extend that functionality.

Direct Protocol Payload (Type 0)

Direct payloads are special payloads meant for sending directly to the protocol, the delimiter **+** divides the subject and message which are the only two fields it carries.

```
type+title+body
```

It's always recommended to use other payloads for dApp or server interaction. This payload should be used sparingly when it's absolutely necessary. The 'type' here is a special field which is different from the type in identity.

Always recommended to interface with **EPNS JS Library** for abstracting these details out.

Broadcast Payload (Type 1)

Broadcast notification goes to all subscriber of a channel, the notification payload in this case is not encrypted.

```
{
  "notification": {
    "title": "The title of your message displayed on screen (50 Chars)",
    "body": "The intended message displayed on screen (180 Chars)"
  },
  "data": {
    "type": "1",
    "secret": "",
    "asub": "[Optional] The subject of the message displayed inside app (8
    "amsg": "[Optional] The intended message displayed inside app (500 Cha
    "acta": "[Optional] The cta link parsed inside the app",
    "aimg": "[Optional] The image url or youtube url which is shown inside
    "atime": "[Optional] Epoch time for the notification to be shown or di
  }
}
```

Secret Payload (Type 2)

Secret notifications are intended to be delivered to one subscriber of the channel, these are encrypted using ECC(Elliptic Curve Cryptography) [4] and AES(Advanced Encryption Standard) [5] . The secret which is generated by the channel using whatever means they prefer should be kept to 15 characters or less, this secret (plain version) uses AES to encrypt the fields: **asub, amsg, acta, aimg**.

The rationale behind using ECC with AES is to ensure that the payload is not over bloated.

```
{
  "notification": {
    "title": "The title of your message displayed on screen (50 Chars)",
    "body": "The intended message displayed on screen (180 Chars)"
  },
  "data": {
    "type": "2",
    "secret": "No more than 15 characters, encrypted using public key of t
    "asub": "encrypted by secret using AES | [Optional] The subject of the
    "amsg": "encrypted by secret using AES | [Optional] The intended messa
    "acta": "encrypted by secret using AES | [Optional] The cta link parse
    "aimg": "encrypted by secret using AES | [Optional] The image url whic
    "atime": "[Optional] Epoch time for the notification"
  }
}
```

Why not just use ECC? ECC increases the length of the cipher text and hence the payload which will be delivered. Using ECC with AES ensure the payload length is kept at a manageable level and allows channels to send more information in the notification while still keeping the best encryption practices.

The discussion on using a public key - private key encryption or a derivation of it is still under discussion. Even if we decide to go with the above encryption for secret messages, we can in the future create a payload type that can offer encryption / decryption in a different way.

Example Notification Payload (Plain)

```
{
  "notification": {
    "title": "The title of your message displayed on screen (50 Chars)",
    "body": "The intended message displayed on screen (180 Chars)"
  },
  "data": {
    "type": "2",
    "secret": "vBGK71PFl7mzWob",
    "asub": "The Great Renewal: Your ENS Domain has expired and someone is
    "amsg": "[d:ENS] domains from 2017 that have expired.\n\nGo check your
    "acta": "https://ens.domains/",
    "aimg": "https://i.ibb.co/WKNVN9y/enssamplemsgimg.jpg",
    "atime": "1595083821"
  }
}
```

Recommended to use the **EPNS JS Library** to handle the generation of encrypted payload easily, it can talk to our protocol to also fetch the public key required.

Example Notification Payload (Encrypted)

```
{
  "notification": {
    "title": "The title of your message displayed on screen (50 Chars)",
    "body": "The intended message displayed on screen (180 Chars)"
  },
  "data": {
    "type": "2",
    "secret": "e291826a995ab03b0dd69c360f3deb3803e130dc7d3bd94f1016494bc1f
    "asub": "U2FsdGVkX181J09umWprAgmLOaDyZXojQjLPlJ31G0LDgXHBgnNsFEOKgjqhK
    "amsg": "U2FsdGVkX18J7Myet9yljBLtNMpqz86qWgmjrK/9WyP+LD9OVerohkl5jc791
    "acta": "U2FsdGVkX188zXRR3URQR2xedjftDOHD5E3k+ggKe+8F6MxW86464rl6y1ZhX
    "aimg": "U2FsdGVkX188AaU187LFzqaibpfoOXb+XkCNbsLpV29CrQOVjC9BfWpxwGXE9
    "atime": "1595083821"
  }
}
```

Targeted Payload (Type 3)

Targeted notifications go to a single subscriber of a channel, the notification payload in this case is not encrypted.

```
{
  "notification": {
    "title": "The title of your message displayed on screen (50 Chars)",
    "body": "The intended message displayed on screen (180 Chars)"
  },
  "data": {
    "type": "3",
    "secret": "",
    "asub": "[Optional] The subject of the message displayed inside app (8
    "amsg": "[Optional] The intended message displayed inside app (500 Cha
    "acta": "[Optional] The cta link parsed inside the app",
    "aimg": "[Optional] The image url or youtube url which is shown inside
    "atime": "[Optional] Epoch time for the notification to be shown or di
  }
}
```

Extending for Future Payload

> The following payloads are in discussion and form an example of how the payload specs is ever expanding and can include more than just information as notification.

Multi-Targeted Payload (Type x)

Multi-Targeted notifications go to a more than one subscriber of a channel, the notification payload in this case is not encrypted. The total number of subscribers supported is TBA.

```
{
  "notification": {
    "title": "The title of your message displayed on screen (50 Chars)",
    "body": "The intended message displayed on screen (180 Chars)"
  },
  "data": {
    "type": "4",
    "secret": "",
    "asub": "[Optional] The subject of the message displayed inside app (8
    "amsg": "[Optional] The intended message displayed inside app (500 Cha
    "acta": "[Optional] The cta link parsed inside the app",
    "aimg": "[Optional] The image url or youtube url which is shown inside
    "atime": "[Optional] Epoch time for the notification to be shown or di
  },
  "recipients": {
    [0xAb...],
    ...
    [0xEb...]
  }
}
```

Blacklist Payload (Type x)

Blacklist payload are a future forward payload meant to only demonstrate how the payload data and type defines what information the payload will carry.

```
{
  "data": {
    "type": "5",
    "blacklist": {
      address1,
      address2,
      ....
      address100
    }
  }
}
```

The support for payload types is left to third party frontend / infrastructure services to implement. The accompanying EPNS products however will implement all payloads types which are accepted as official after community discussion.

EPNS Protocol

Introduction

Ethereum Push Notification Service protocol will be on the Ethereum blockchain that provides and standardizes the ways by which notification on blockchain can operate.

> In the future, the protocol can also support other blockchain by exploring bridging or migrating the contract and service to a particular blockchain.

Primary Use Cases & Features

- Users
  - Users registry
  - Public key registry
- Channels
  - Types of Channels
  - Channels registry
  - Special channels
  - Channel Activation & Deactivation
    - Game Theory and User Incentives
  - Deriving fair share of interest for a channel from stake pool
  - Updating Channel
  - Spam rating and throttling
- Subscribers
  - Subscribing to Channels
    - User direct action subscribe
      - Game Theory and User Incentives
    - Deriving weighted earnings of a subscriber of a channel
    - Indirect subscribe action (delegate subscription of user by channel)
      - Game Theory and User Incentive
  - Unsubscribing from Channel

- Sending Notifications
    - Protocol interfacing for Notifications
    - Delegation of notifications
- Claiming Earnings from Protocol

> The primary use cases might change or tweaked in the future as more protocol features are built.

Users

Any entity (wallets) involved in protocol (contract owner, channels or normal users) is referred to as **Users** of the protocol. User can become a channel by activating themselves as a channel, or can become subscriber to one or more channel and start earning interest.

Users Registry

Concept

The users registry is maintained in the protocol to carry out various tasks such as stopping channels from adding them again (after a user has unsubscribed), if their public key is in the registry, what channels they are subscribed to and a mapping to get to those channels, if they own a channel, etc.

Key data stored on chain

- If the user is already in the ecosystem
- If their public key is already in the registry
- Whether they own a channel
- The channels they have subscribed to
- The channels they have graylisted

Public Key Registry

Concept

Encrypted notification generally uses an algorithm based on public key or its derivative thereof. To facilitate this, the EPNS protocol maintains a public key registry of all the users who enter the ecosystem. This is done by having mirrored function with public key registry for **creating a channel** and **subscribe** which allows the protocol to emit public key of the wallet out (or skip if it already exists).

> Since public key for any wallet is derivable if they have done even a single transaction, we consider this information as safe non-invasive to privacy.

The public key registry for users is completely optional and the protocol has alternative methods that completely skip this process but carry the required functionality. The registry is helpful on the frontend side to quickly poll and find the key of the wallet and use it to encrypt the notifications wherever necessary.

Channels

Any user who activates themselves on the protocol to send notification is called a **Channel**. Channel stakes fees in a stake pool, the interest earned from which is distributed among their subscribers in a weighted proportion.

Types of Channels

The protocol allows a service to choose what type of channel they want to create. This enables several other business use cases than just delivering information via notification.

- Open Channel
- Closed Channel
- Mutual Channel

Open Channel

The default channel, this channel is created by the service and is intended to be open for any user to come and subscribe without any restrictions. Channel can also indirectly subscribe the user to it by paying the user a minor fee.

Closed Channel

A service can opt to create a closed channel, this channel cannot be directly subscribed by the user. Instead, the channel needs to add the user indirectly by paying a minor fee to the user.

Mutual Channel

A service can opt to create a mutual channel, which requires user direct action to subscribe to it, but the subscription is only confirmed once the channel approves it as well.

> This is not a definitive list, we might add more types of channels as per the need in the future.

Channels Registry

Concept

The channels registry is maintained in the protocol to carry out various tasks such as carrying weights and indexes for deriving the fair share of interest from the common stake pool, mapping the number of subscribers, adjusting the spam rating, etc.

Key data stored on chain

- Type of channel
- Channel status
- Stake pool contribution
- Time at which the channel was created (block number)
- Mapping of its subscribers
- Mapping of key data points of subscribers for calculating their fair share from the derived interest of the channel

Special Channels

There are two special channels in the protocol that are owned and controlled by the contract owner. These channels are special because:

- They don't give any interest to their subscribers.
- They are the only automated opt-in channels in the protocol that doesn't pay subscribers (yet add them without their direct consent).

EPNS Channel

This channel is an auto subscribe channel for all the users of the protocol. The purpose of the channel is to send out extremely important events or alerts to the subscribers. While an auto subscribe, the channel can be unsubscribed by the user and like all the other channels will lose permission to add that user again indirectly.

EPNS Alerter Channel

The channel is an auto subscribe channel for all the other channels of the protocol. The purpose of this channel is to send out updates, alerts to other channels of the protocol. While an auto subscribe, the channel can be unsubscribed by the user and like all the other channels will lose permission to add that user again indirectly.

Channel Activation & Deactivation

Concept

A **service** needs to activate itself on the protocol as a one time step before they can send notifications to their subscribers. When a service is activated on the protocol, they are referred to as a **Channel**.

The channel is also required to stake fees in DAI which is **50 DAI or higher**, this is used to create a staking pool which in turn interacts with other DeFi protocols to earn interest. This interest is then distributed to all the **subscribers** of that service in a weighted manner preferring the earliest subscriber more than the later ones.

> There is always 1:1 mapping between a service (wallet) and the channel it creates.

Game Theory and User Incentives

In order to ensure the proper participation of all players, following game theory is applied to keep each player in the system motivated to be a good actor in the ecosystem.

- Contract owner cannot send notification on behalf of channel unless the control is delegated to it by the channel owner as an on-chain event.
- The **service** needs to deposit **50 DAI or higher** as staking fee to activate themselves, this ensures that serious players participate in the game.
- The **channel stake** goes to a combined stake pool of channels and starts earning interest using **AAVE Protocol** (at the time of writing, this can be replaced with a similar DeFi protocol in the future).
- The **channel owner** decides the fees, since this interest is distributed among their users weighted towards the earliest subscriber, and is refundable, having **higher stake incentivizes more users** to subscribe to the channel.

- This **channel** can be deactivated by the **channel owner** with a penalty of **20 DAI**, this ensures that bad actors are unable to abuse the system as a monetary loss will occur with each recurrence, small enough for serious players to not worry about it but act as a further deterrent for bad players.
- Upon deactivation, the channel is unable to send notification and its ratio in the **stake pool** is reduced to **10 DAI**, as a result, the proportionate ratio from interest pool for the channel is also reduced accordingly. The channel continues giving its fair share of interest to all existing subscribers of the channel, though the channel loses the ability to add more users.
- The rest of the penalty (**10 DAI**) goes into the revenue pool.

> There is an upper ceiling limit applied for staking fee to ensure that a service is not approving an unlimited amount of fees to transfer (250k DAI at the time of writing). This is mostly to avoid the mentioned problem and should not really be effecting any features or UX of the protocol.

> The penalty fee, stake fee can be adjusted in the future before going to the mainnet. After mainnet, it can only be adjusted by voting mechanism of the protocol.

Deriving fair share of interest for a channel from stake pool

Concept

The stake pool is a pool that contains all the staking fees from all the channels. There are two important concepts to keep in mind to derive the fair share of interest for an individual channel.

- The staking fee starts earning interest as soon as a channel is activated, this means that the fair share of a channel would need to account for the time since the channel is activated to ensure fair play. This is achieved by recording the **block number** at which a channel is activated.
- The staking fee is dynamic and can differ from one channel to another, i.e. channel decides the amount based on their game theory and incentive plan they put in place. To account for this, the **weight** of the channel and **aggregated weight of all channels** are recorded at the time a new channel creation.

Problem Statement

We need to derive the formula which gives us the ratio of the interest the channel has generated so that it can be distributed among the subscribers.

While this can be solved by recursion of all channels, getting their share in the pool, deriving their weight and the start block number. This will be inefficient as more channels continue to get activated on the protocol.

Solution

Instead of recursion, we are going to approach the formula in a linear way. We know that each time a channel is added, the previous number of channels, their block number, etc can be treated like a constant as their values will not change moving forward.

For example, if **channel x** got added at **block y**, then the channels count before **block y** would never change, nor will the differential which we have before **channel x** was added. We utilize this to form our own ratio calculation that is not based on recursion but instead is based on the aggregation that happens whenever a channel is added or removed. The different weights of the channel are also mapped to an aggregated weight variable which helps us in determining the total sum needed to derive the fair share of interest for the channel.

Putting it together, the formula by which we can derive a channels' fair share of interest from the stake pool for current block can be written as:

Formula to calculate fair share interest ratio

$$ratio = (d * a)/(z + n * x * w)$$

Formula to calculate historical constant (z)

Occurs whenever a channel is added, deactivated or the stake fees changes

$$z = z + (n * x * w)$$

Parameter	Explanation
d	Differential of current block number and the block number on which the channel was last modified in relation to stake fee.
a	The actual weight of the channel, measured by dividing the stake fee of the channel with the minimum stake fee.
z	The historical constant of the **channels**, calculated whenever a modification in number of channels occurs or whenever the stakes fee of the channel is changed.

n	The previous count of the number of channels present in the protocol.
x	Differential between the latest block number and the last modified block number. Last modified block number is whenever a channel was added, deactivated or fees of any channel was changed.
w	The averaged out weight of all the channels, the average is normalized / averaged out whenever a new channel is added, deactivated or fess of any channel is changed.

Updating Channel

A channel can be updated as long as the number of subscribers to that channel is equal to one, which means that the channel owner is the only subscriber of the channel.

The channel becomes immutable as soon as a new subscriber joins the channel, to avoid, constant changes on the channel (even with a single subscriber). A fee of **10 DAI** is charged that goes to the **revenue pool**.

> The penalty fee can be adjusted in the future before going to the mainnet. After mainnet, it can only be adjusted by the voting mechanism of the protocol.

Spam score and throttling

The protocol uses game theory and incentives to ensure that the channel stay useful to the users. However, there might come a point where a channel might become a bad actor despite the incentives and penalty imposed. This can happen in various scenarios, a couple of them can be:

- Popular channel is bought by someone else with the purpose to promote advertisements.
- Channel private key is compromised.

These edge cases are handled by the protocol by introducing the concept of spam rating and throttling of sending notification, this is done in the following way:

- The channel always starts with a spam score of neutral position (0.5), this rating can go towards 0 (positive position, good actor) or 1 (negative position, bad actor).
- The score is adjusted to move towards positive position (0) whenever a new user is added, this is weighted by the total number of users in the ecosystem as well.
- The score is adjusted to move towards negative position (1) whenever a user unsubscribes and indicates spam, this rating is weighted against the total number of subscribers of the channel.
- The score slowly inches towards positive position (0) as block number increases (i.e. time increases) but takes the spam score into account in relation to how quick the spam score will decrease, i.e. the higher the spam rating, the more minute adjustment is done towards the positive position.
- As the spam score start reaches or exceeds **0.8**, the number of notifications starts getting limited for a channel as a quadratic function.

The feature is expected to change heavily as it's still in development.

The spam score throttle can be adjusted in the future before going to the mainnet. After mainnet, it can only be adjusted by voting mechanism of the protocol.

Subscribers

Subscriber is any user that subscribes to one or more channel. A channel can also be a subscriber of other channels. Furthermore, a channel cannot send a notification to a user who is not a subscriber of their channel to ensure user is in complete control of what channels can or cannot send them notification.

Though means to delegate subscribe by the channel is given with greater incentiv es for the user (subscriber) to ensure win-win scenario [6] and to cover the need of the channel as well.

> The channel owner will always be the first subscriber of the channel as soon as the channel is activated, it's not possible to remove the channel owner (subscriber) from their own channel.

Subscribing to Channel

Concept

The service benefits from establishing a genuine communication channel with the user and the user benefits from receiving the notifications useful to them which is a great incentive in itself.

However, the decentralized web has also introduced some limitations of its own which are needed to be counteracted with incentives of decentralization to ensure seamless adaption and more benefits than traditional services. The common inconvenience with protocols in general and with EPNS in particular when a user needs to do a subscription action can be summarized as follows:

- The users need to transact on blockchain to specifically subscribe or unsubscribe to a channel or multiple channels, this leads to an incentive issue, i.e. **why would a user spend gas in most cases?**
- The users can be added by the channel in delegated fashion (though only once), this feature while required at times to ensure backward compatibility and handling of different use cases also leads to the issue of adding users without their consent. **What can we do to ease the pressure points?**

To address these issues, we built means for the user to earn along with receiving notifications. The earning is designed to be relativity more rewarding for indirect action done on behalf of the user (channel subscribing them). The subscription section focuses a lot on the win-win aspect of Game Theory [6] and how the protocol ties to ease the pain points of the user by balancing them with incentives.

User direct action subscribe

Concept

The user can opt to directly subscribe to a channel via an on-chain event, this enables the service to send notifications to the users, channel benefits from engagement with their subscriber while the subscriber benefits from receiving useful information through the notification and earn from them as well.

Game Theory and User Incentives

In order to ensure that the user is incentivized to do an on-chain transaction (however negligible), we have applied the following game theory:

- The interest earned from the interest pool, which gets generated by the stake pool of the channels is distributed to the subscribers of the channel in a weighted manner.
- This means that the earliest subscriber receives a higher share of the interest earned by the channel than the subscriber coming later than them.
- The user continues to earn interest from the channels they are subscribed to until they unsubscribe, in which case, the part of the interest that they have earned from that channel is automatically sent to their earning reserve on the protocol.
- The user can also withdraw all earnings whenever they want without unsubscribing from any channel.

Deriving weighted earnings of a subscriber of a channel

Concept

The interest pool is a pool that contains all the interest generated by the stake pool (of channels). There are two important things to keep in mind to derive the weighted earning of a subscriber of a channel:

- The interest pool contains entire interest generated by all the channels which means we need to first find the ratio of the interest earning belonging to a specific channel. We have covered how to do this in deriving fair share of interest for a channel from stake pool.
- Next, we need to calculate the weighted earning of a particular subscriber of the channel, this is necessary since subscribers who joined earlier would be receiving more notifications and thus should be entitled to higher share in the interest earning of the channel than the subscribers who joined later.

Problem Statement

We need to derive the formula which can give us the exact earning of the subscriber of a specific channel. The subscriber share of interest ratio needs to be weighted favoring earlier subscription.

While this can be solved by recursion of all subscribers, getting their join time (block number) and applying a simple ratio formula. This will be inefficient as more subscribers will lead to higher inefficiencies.

Solution

Instead of recursion, we are going to approach the formula in a linear way. We know that each time a subscriber is added, the previous number of subscribers, their block number, etc can be treated like a constant as their values will not change moving forward.

For example, if **subscriber k** is added to **block b**, then the subscriber count before **block b** would never change, nor will the differential which we have before **subscriber k** was added. We utilize this to form our own ratio calculation that is not based on recursion but instead is based on the aggregation that happens whenever a subscriber is added or removed.

Putting it together, the formula by which we can derive a channels' fair share of interest from the stake pool for the current block can be written as:

Formula to calculate weighted subscriber ratio

$$ratio = d/(z + n * x)$$

Formula to calculate historical constant (z)

Occurs whenever a subscriber is added, deactivated or the stake fees changes

$$z = z + (n * x)$$

Parameter	Explanation
d	Differential of current block number and the block number on which the subscriber was last removed or added.
z	The historical constant of the **channels**, calculated whenever a modification in number of subscribers occurs.
n	The previous count of the number of subscriber present in the channel.
x	Differential between the current block number and the last modified block number, last modified block number is whenever a subscriber was added or removed.

Indirect subscribe action (delegate subscription of user by channel)

Concept

The indirect subscribe action provides channels a way to subscribe users and thus send notification to them without the explicit permission of the user. While this action is obtrusive in nature and opens up the protocol for misuse, we do however also see it's use cases:

- When channel doesn't have any means to reach the user
- When user is not added in the protocol or the channel but notification is important (example: ENS domain expiry or security compromise)
- For closed channels and their business use case (For example, notification that do multi-factor authentication should not be open for any action of the user)

Seeing this as a necessity, we do have provisions in the protocol to initiate subscribe from a channel to a user. Since, this action has larger consequences on the user if misused, we have set the following rules and higher incentives for indirect subscribe action.

Game Theory and User Incentives

The indirect subscribe action imposes the following rules and conditions on the channel:

- The indirect subscribe can only occur once, if the user unsubscribes, there is no way to add them back.
- The user can set and change at will their fee that a channel should to pay to them for indirect subscribe.
- The channel needs to pay the user the above specified amount and only then can the indirect subscribe occur, this directly goes to the users earning reserve.
- If the user hasn't set an indirect fee amount, then a default amount **0.1 DAI** still needs to be paid before an indirect subscribe can occur.
- A minor fees (0.2% at the time of writing) is also charged which is sent to the Fee Pool.

The indirect fees can be adjusted in the future before going to the mainnet. After mainnet, it can only be adjusted by voting mechanism of the protocol.

Unsubscribing from Channel

Concept

The channel has no control on the unsubscribing action of its' subscriber. Though the subscriber can indicate whether they are unsubscribing because:

- They don't want to receive notification from the channel anymore, in which case, the channel spam score is unaffected.
- However, if the subscriber indicates that they unsubscribed due to spam then the spam score of the channel is affected as outlined in channel's spam score and throttling.

Sending Notifications

Concept

EPNS is a decentralized notification protocol that abstracts the data layer to enable sending notifications to both decentralized and centralized carriers.

- Sending notifications on-chain is free and no fee is charged for delivering notifications to decentralized carriers.
- A minor fee in **$ETH** is charged when the notification needs to be carried from decentralized to centralized carriers and finally to a centralized platform which requires infrastructure and a business model to maintain it.

The protocol charges micro fees on notifications that are sent out. The fees from this goes to the fee pool of the protocol.

> The fee pool is divided into 20:80 ratio. 80% of the fee goes to the protocol token holders while 20% is reserved for future upkeep of the protocol / product.

Protocol Interfacing for Notifications

Concept

All of the payloads are uploaded as JSON format in decentralized storage solutions (**or in some special future cases, even centralized ones**). The EPNS JS Library interfaces with Ethereum Push Notification Service protocol and calls:

```
sendNotification(address _recipient, bytes _identity)
```

Parameter	Sub Field	Description
_recipient		Differs with the payload type, broadcast and special multi payload notifications have the channel address as the recipient address.
_identity		The identity field consists of the following parameters joined together with a delimiter.
	pushtype	Indicates service wants to push the notifications out and can in the future be segmented to different platforms push (i.e. : 1 for every platform, 2 for mobile, 3 for web browsers, etc).
	payloadtype	Payload type not only indicates the content of notification but also the storage implementation stored.
	payloadhash	Indicates the hash of the payload through which payload data can be obtained.

The delimiter **+** is used for joining the fields together, this is done to optimize the payload written on chain.

Example _identity:
1+2+QmcdzjicUnxv8ASKKSgEEYjhK7symwxqDG4BeCS82rdNBk

Always recommended to interface with **EPNS JS Library** for abstracting these details out.

This feature of protocol will keep evolving for further optimizations in the future.

Delegation of Notifications

Concept

The channel can also opt to delegate sending notification from another wallet address in the protocol. If the delegation is present, that wallet can send notification on the protocol on behalf of that channel. The delegation can be revoked at any point of time by the channel owner.

This is useful in providing value added services to the channels and having mechanisms that can be used by EPNS infrastructure or other third party infrastructure to send notifications on-chain on a channels' behalf.

Claiming Earnings from Protocol

Claiming Earnings

The protocol provides ways to earn for users based on some of their direct or indirect actions. These include earnings resulting from:

- Subscribing to channels by their direct action (earn interest).
- Indirect subscribe by a channel (channel pays user either a default minor reward or the expectation set by the user).

These earnings are mapped to the user in the protocol and the user is free to claim these anytime they want to. The earning are generated and kept as aDai (AAVE Interest bearing DAI) for users to keep generating interest automatically for them. They are converted to protocol tokens when the user withdraws them.

EPNS Products

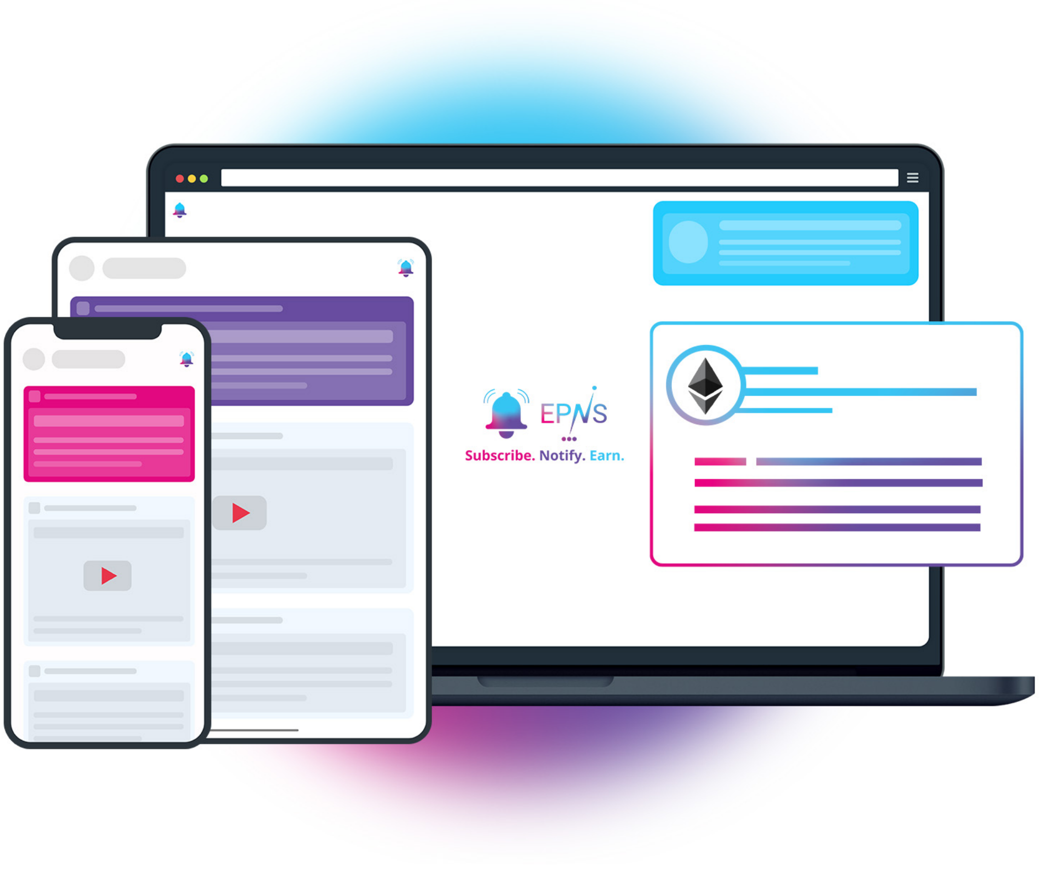
Problem Statement

EPNS notification protocol implements game theory, DeFi incentives and enables sending decentralized push notification in platform agnostic fashion. It's designed to be integrated with third-party wallets so that notifications could finally arrive and achieve the network effect on the blockchain.

Despite this, it does suffer from the classic **chicken and egg problem**! Which is, for dApps or services to implement the protocol, they would want the notifications to be delivered to their users and see the value before adopting it to **send notifications**, and unless they adopt it, it's going to be tough for user wallets to spend time in integrating a protocol and frontend for **receiving notification**.

Solution

In order to facilitate the adoption of the protocol and to provide value to services, we will also be building the product suite of EPNS.



Ethereum Push Notification Service Protocol / Product

Mobile App

Serves the purpose of delivering notifications from decentralized protocol to centralized EPNS Infra to centralized platforms (iOS and Android).

dApp

Enables receiving notifications from web browsers and also enables delivery of notifications from protocol to decentralized carriers.

EPNS Infra (Push Service)

Enables carrying notifications from decentralized protocol to centralized solutions (iOS, Android, Web, etc). Also enables third-party dApps, services and protocols to start experiencing the notification impact as notifications are delivered following the entire protocol / product lifecycle.

Showrunners

These are channels created and run by us for the benefit of the community and for users to come and see why push notifications transformed the traditional world. Few examples of showrunners which we will be running are:

- **Wallet crypto movement tracker**
- **ENS domain expiry**
- **Compound Liquidation alert**
- **EthGas abnormal price alerter**
- **Crypto price tracker**

JS Library

Considerably reduces the integration time required for third party dApps, servers.

We see these products to enable instant value add to the protocol and help in increasing awareness and eventually drive the adoption of the protocol.

Integration Flow for dApp / Server / Smart Contract

Some of the ways by which the protocol can be Integration into the dApp, Server or Smart Contract flow is designed to keep developer hassle to a minimum and as such, the recommended way to connect to protocol through various architecture are explained.

- Creating Channel
- Sending Notification from dApp / Serverless
- Sending Notification from Server
- Sending Notification from Smart Contract

Creating Channel on dApp / Server / Smart Contract

The creation of channel is a **one time process** and as such is recommended to do it either from EPNS dApp, EPNS JS library or a custom JS library of your choice. Information about the channel payload specs is described here.

Interfacing directly via smart contract to protocol to create a channel can also be done by calling it

- with **public key registry** function

```
createChannelWithFeesAndPublicKey(bytes calldata _identity, bytes calldata _
```

- without **public key registry** function

```
createChannelWithFees(bytes calldata _identity)
```

Parameter	Sub Field	Description
_identity		The identity field consists of the following parameters joined together with a delimiter.
	channeltype	The type of channel to create.
	payloadtype	Payload type not only indicates the content of notification but also the storage implementation stored.
	payloadhash	Indicates the hash of the payload through which payload data can be obtained.
_publickey		Pass the publickey of the wallet in bytes

channeltype	Description
1	Open Channel
2	Closed Channel
3	Mutual Channel

payloadtype	Description
1	Represents storage on IPFS with letters Qm happen to correspond with the algorithm (SHA-256) and length (32 bytes) used by IPFS.

The delimiter **+** is used for joining the fields together, this is done to optimize the payload written on chain.

Example **_identity**:
1+1+QmXSuc8iVsNFtsqrFvgHWpa6tJXFLoq2QEWYu2aS6KF8ux

Example **_publickey**:
0x187c0568118be8032ece2499135d16a69b1da955125185c195a900d45eed0a325f2a7bf3ce6eb01375011db55d3a311fd84c2a17d0476edf8c6290f36ed2280b

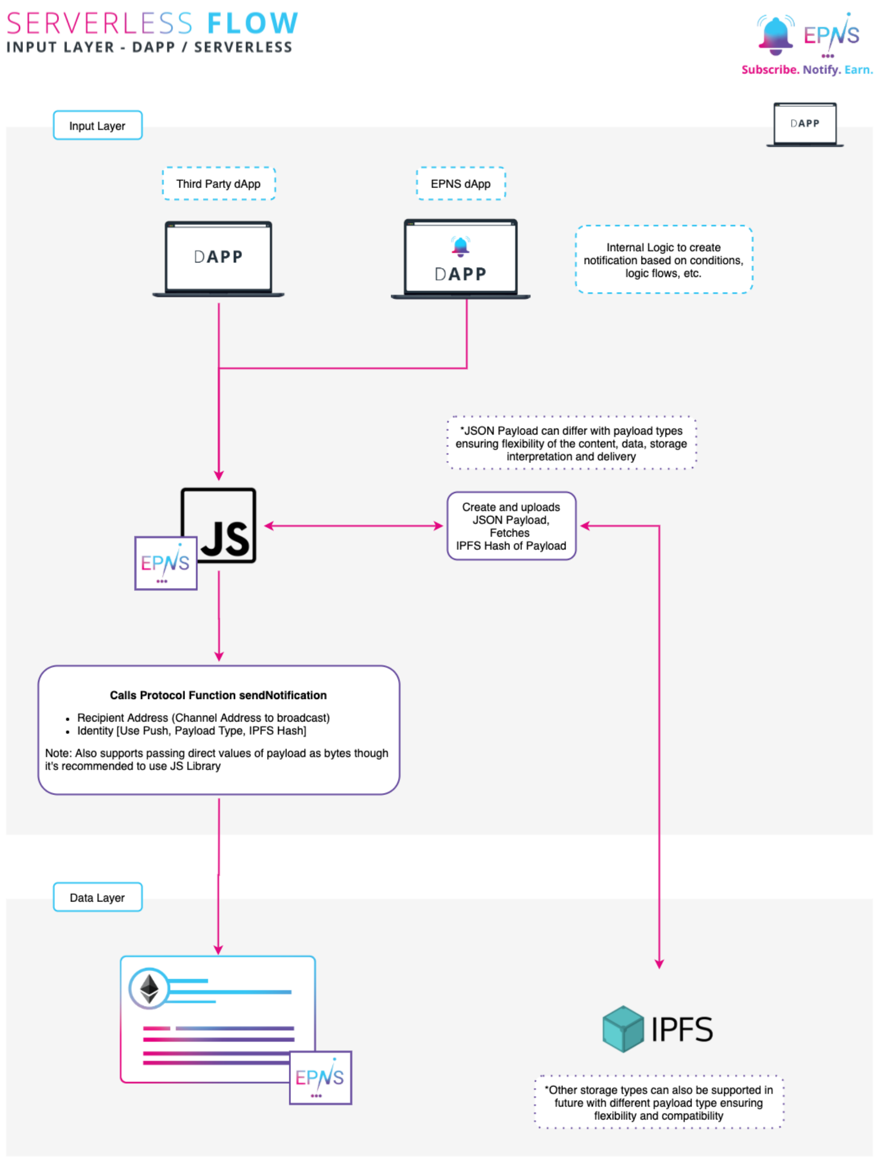
Always recommended to interface with **EPNS JS Library** for abstracting these details out.

This feature of protocol might change for further optimizations in the future.

Sending Notification from dApp / Serverless

dApp / Serverless integration workflow

EPNS allows various ways to integrate the protocol into your service. The following flow shows how a dApp can send notification to the protocol.


SERVERLESS FLOW
INPUT LAYER - DAPP / SERVERLESS
EPNS
Subscribe. Notify. Earn.
Input Layer
DAPP
Third Party dApp
EPNS dApp
DAPP
DAPP
Internal Logic to create notification based on conditions, logic flows, etc.
*JSON Payload can differ with payload types ensuring flexibility of the content, data, storage interpretation and delivery
JS
EPNS
Create and uploads JSON Payload, Fetches IPFS Hash of Payload
Calls Protocol Function sendNotification
Recipient Address (Channel Address to broadcast)
Identity [Use Push, Payload Type, IPFS Hash]
Note: Also supports passing direct values of payload as bytes though it's recommended to use JS Library
Data Layer
EPNS
IPFS
*Other storage types can also be supported in future with different payload type ensuring flexibility and compatibility

---

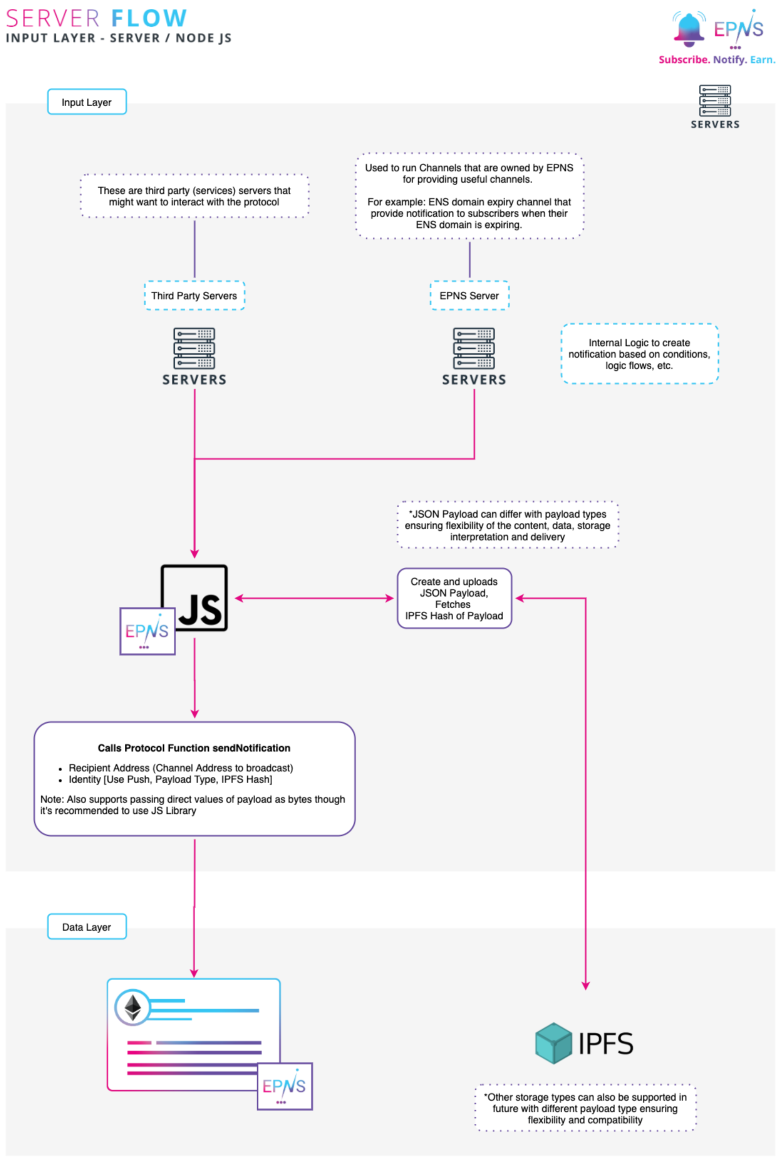
Sending notification via the dApp

1. Use your internal logic to figure out what notification you want to send (i.e. alerting users on some smart contract event, user actions, movement in their wallets, a podcast or post from your end, etc).
2. Form the JSON payload using our JS Library or your which you want to send as notification. Please check supported payload types and their requirements.
3. Interact with protocol using EPNS JS Library.

Sending Notification from Server

Server integration workflow

EPNS allows various ways to integrate the protocol into your service. The following flow shows how your server can integrate and send notification to the protocol.


SERVER FLOW
INPUT LAYER - SERVER / NODE JS
EPNS
Subscribe. Notify. Earn.
SERVERS
Input Layer
These are third party (services) servers that might want to interact with the protocol
Used to run Channels that are owned by EPNS for providing useful channels.
For example: ENS domain expiry channel that provide notification to subscribers when their ENS domain is expiring.
Third Party Servers
EPNS Server
SERVERS
SERVERS
Internal Logic to create notification based on conditions, logic flows, etc.
*JSON Payload can differ with payload types ensuring flexibility of the content, data, storage interpretation and delivery
JS
EPNS
Create and uploads JSON Payload, Fetches IPFS Hash of Payload
Calls Protocol Function sendNotification
Recipient Address (Channel Address to broadcast)
Identity [Use Push, Payload Type, IPFS Hash]
Note: Also supports passing direct values of payload as bytes though it's recommended to use JS Library
Data Layer
EPNS
IPFS
*Other storage types can also be supported in future with different payload type ensuring flexibility and compatibility

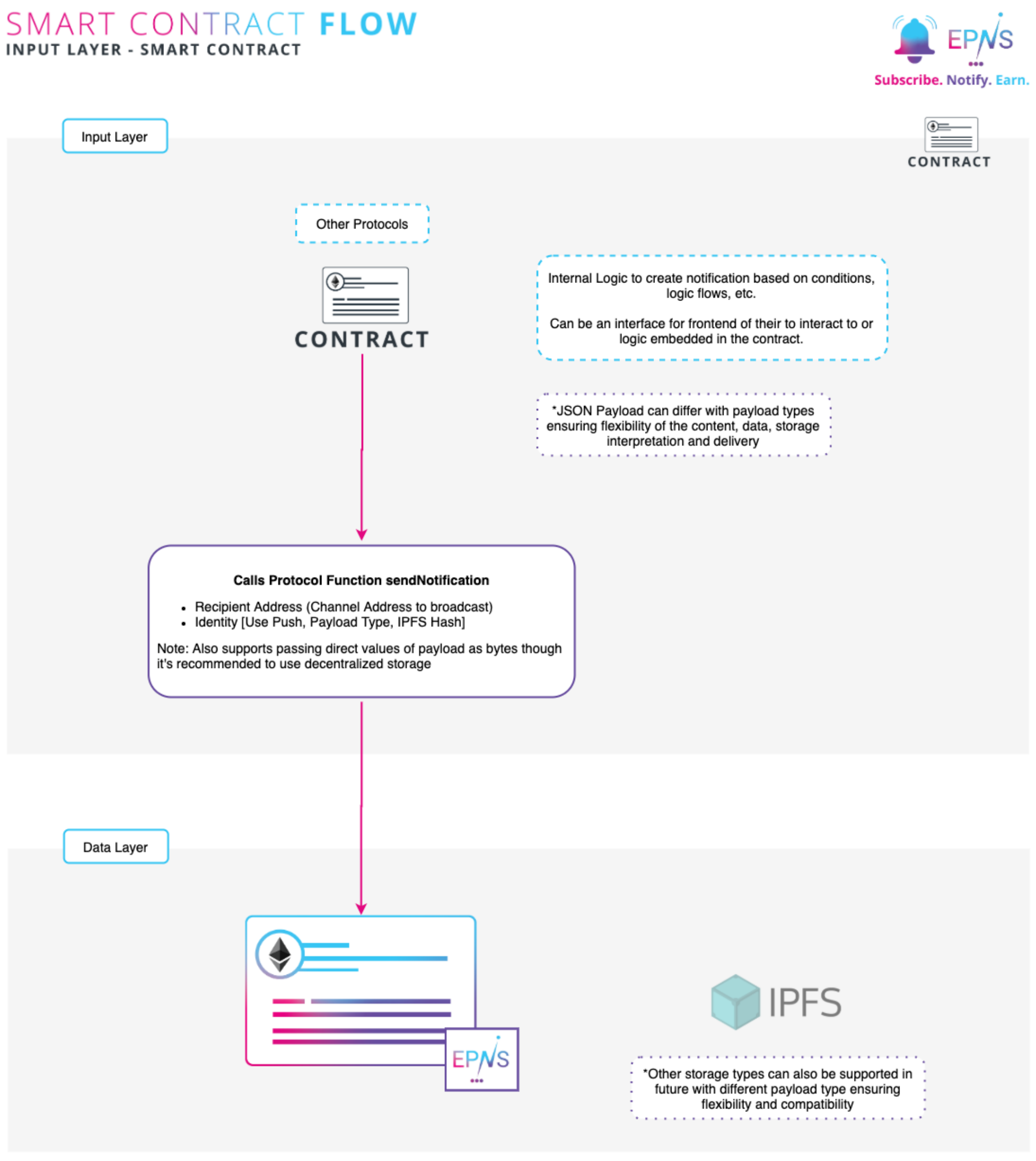
Sending notification via the server

1. Use your internal logic to figure out what notification you want to send (i.e. alerting users on some smart contract event, user actions, movement in their wallets, a podcast or post from your end, etc).
2. Form the JSON payload using our JS Library or your which you want to send as notification. Please check supported payload types and their requirements.
3. Interact with protocol using EPNS JS Library.

Sending Notification from Smart Contract

Smart Contract workflow

EPNS allows various ways to integrate the protocol into your service. The following flow shows how your server can integrate and send notification to the protocol.


SMART CONTRACT FLOW
INPUT LAYER - SMART CONTRACT
EPNS
Subscribe. Notify. Earn.
Input Layer
CONTRACT
Other Protocols
CONTRACT
Internal Logic to create notification based on conditions, logic flows, etc.
Can be an interface for frontend of their to interact to or logic embedded in the contract.
*JSON Payload can differ with payload types ensuring flexibility of the content, data, storage interpretation and delivery
Calls Protocol Function sendNotification
Recipient Address (Channel Address to broadcast)
Identity [Use Push, Payload Type, IPFS Hash]
Note: Also supports passing direct values of payload as bytes though it's recommended to use decentralized storage
Data Layer
EPNS
IPFS
*Other storage types can also be supported in future with different payload type ensuring flexibility and compatibility

Sending notification via the smart contract

1. Use your internal logic to figure out what notification you want to send (i.e. alerting users on some smart contract event, user actions, etc).
2. This can be done by either having internal logic cooked in your protocol or better yet having a function which you can call from outside which can interact with our protocol.
3. Please check for supported payload types and their requirements.
4. Either pass the hash of the content or the data in bytes to EPNS protocol.

Future Features & Research

The following future features are getting researched on

- Exploring IPNS (DNS) [7] for IPFS as a means to form a decentralized communication system between users.
- IPNS is a static file on IPFS that points to your website that's hosted on IPFS.
- We are researching ways in which we can use this to potentially form several communication point that are enabled between user to user.
- This can lead to exciting possibilities, for example, **having a chat thread** on IPFS that carries with itself the previous hash (cid) and IPNS keeps on updating the latest hash as a pointer towards that chat.
- This can also be explored to give way to possible **decentralized video messaging** and other exciting breakthroughs.

> Since this is at research phase, we will update the paper once we have carried out a PoC in this regard.

Governance

Governance

The protocol token has the following major functions that it performs:

- Getting share from fee pool (80% of the fee, split in equal parts for token holders).
- Setting up the price for micro-payment fees.
- Setting up the fee for indirect subscribe action.
- Penalty fee for Channel updating.
- Removing a channel permanently.
- Adjusting spam throttle.

Summary

Summary

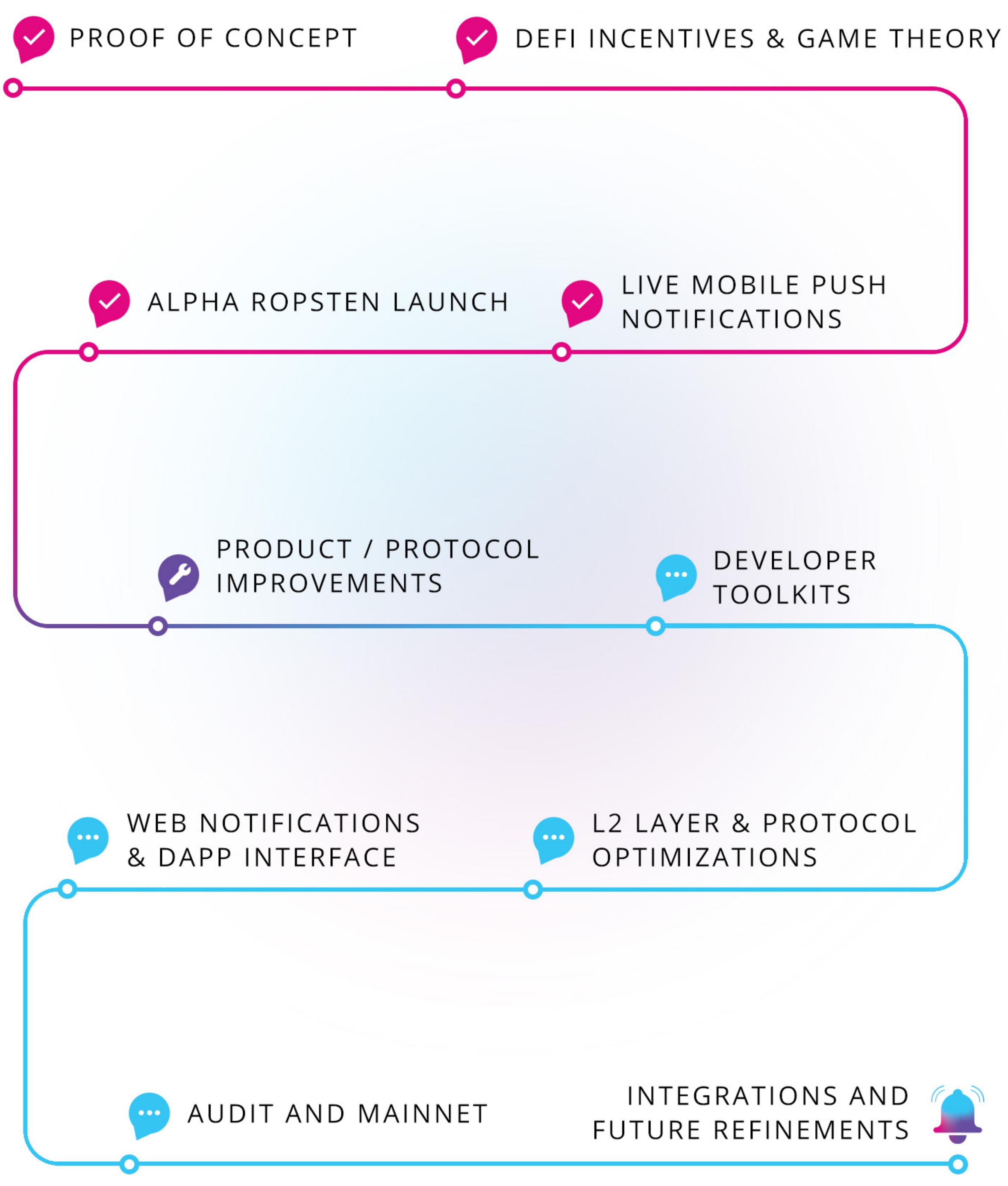
- Ethereum Push Notification Service Protocol introduces variety of ways to form information in notifications that are extremely powerful and extensible on content.
- The rules of the protocol ensures fair participation and encourages all parties through incentives (monetary or otherwise) to achieve win-win situation.
- The product shows how notifications are getting delivered on both centralized and decentralized carriers.
- Most of the features of protocol and product including the DeFi aspect were created before writing this whitepaper to ensure that the information is as accurate as possible.

Milestones

ETHEREUM **PUSH NOTIFICATION** SERVICE

MILESTONES

Subscribe. Notify. Earn.


PROOF OF CONCEPT
DEFI INCENTIVES & GAME THEORY
ALPHA ROPSTEN LAUNCH
LIVE MOBILE PUSH NOTIFICATIONS
PRODUCT / PROTOCOL IMPROVEMENTS
DEVELOPER TOOLKITS
WEB NOTIFICATIONS & DAPP INTERFACE
L2 LAYER & PROTOCOL OPTIMIZATIONS
AUDIT AND MAINNET
INTEGRATIONS AND FUTURE REFINEMENTS

Team & Acheivements

Founders

HARSH RAJAT

11 years of entrepreneurial experience in various spectrum of tech; including architecting, development and design in different tech fields (Mobile, Web Services, SaaS, Blockchain).

RICHA JOSHI

12 years of techno-functional experience in project management and marketing. Have worked with Blockchain Fintech startup, Deloitte, Wipro contributing and leading teams across multiple facets of product.

MENTOR
AND
GUIDANCE

ethereum
foundation

ETHGlobal
HACK
Money

PVD
IDEO

KERNEL
fellowship

GITCOIN

ACCELATOR
AND PRODUCT

ETH HUB

DEFI PULSE

DEFI DAD

FEATURED BY

EDCON

References

References

[1] Blockchain Growth Forecast: https://www.marketsandmarkets.com/Market-Reports/blockchain-technology-market-90100890.html

[2] Impact of Push Notifications: https://blog.e-goi.com/infographic-push-notification/

[3] IPFS Wiki: https://en.wikipedia.org/wiki/InterPlanetary_File_System

[4] Elliptic Curve Cryptography: https://en.wikipedia.org/wiki/Elliptic-curve_cryptography

[5] Advance Encryption Standard: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[6] Game Theory and Trust: https://ncase.me/trust/

[7] IPNS: https://docs.ipfs.io/concepts/ipns/