Concept Paper

Not peer-reviewed version

FEDSTR: Money-In AI-Out | A Decentralized Marketplace for Federated Learning and LLM Training on the NOSTR Protocol

Konstantinos Nikolakakis [*] , George Chantzialexiou , Dionysis Kalogerias

Posted Date: 24 April 2024

doi: 10.20944/preprints202404.1563.v1

Keywords: distributed; parallel; and cluster computing; social and information networks; federated learning; LLMs; communication protocols

Article

FEDSTR (fɛdstər): Money-In AI-Out A Decentralized Marketplace for Federated Learning and LLM Training on the NOSTR Protocol

Konstantinos E. Nikolakakis [1,*], George Chantzialexiou [2] and Dionysis Kalogerias [1]

[1] Yale University; dionysis.kalogerias@yale.edu
[2] Independent Researcher; george.chantzialexiou@gmail.com
[*] Correspondence: konstantinos.nikolakakis@yale.edu

Abstract: The NOSTR is a communication protocol for the social web, based on the w3c websockets standard. Although it is still in its infancy, it is well known as a social media protocol, thousands of trusted users and multiple user interfaces, offering a unique experience and enormous capabilities. To name a few, the NOSTR applications include but are not limited to direct messaging, file sharing, audio/video streaming, collaborative writing, blogging and data processing through distributed AI directories. In this work, we propose an approach that builds upon the existing protocol structure with end goal a decentralized marketplace for federated learning and LLM training. In this proposed design there are two parties: on one side there are customers who provide a dataset that they want to use for training an AI model. On the other side, there are service providers, who receive (parts of) the dataset, train the AI model, and for a payment as an exchange, they return the optimized AI model. The decentralized and censorship resistant features of the NOSTR enable the possibility of designing a fair and open marketplace for training AI models and LLMs.

Keywords: distributed; parallel; and cluster computing; social and information networks; federated learning; LLMs; communication protocols

1. Introduction

The NOSTR (nɔstər) - Notes and Other Stuff Transmitted by Relays [1] is an open protocol that enables censorship-resistant communication. In fact, it is resilient and does not rely on any trusted central servers. The communication over the NOSTR is tamperproof since it relies on cryptographic keys and signatures. The first successful application of the NOSTR protocol is the decentralized social media with at least 150000 known trusted users [2]. The NOSTR has several common structural properties with the competitive open protocol Bluesky [3,4]. Bluesky and the NOSTR both aim to solve the problem of decentralized social media, under partial consistency trade-offs [5,6, CAP Theorem]. Herein, we focus on the NOSTR protocol because it is the most flexible and safest of the protocols in terms of developer control and future development.

In parallel with the NOSTR protocol development, distributed AI training has attracted major attention the last few years, with the Language Model breakthroughs leading the progress. By combining recent advanced tools and breakthroughs from open communication protocols [1], bitcoin payments [7] and distributed optimization algorithms [8], we propose a system design that enables open marketplaces for training AI models. Specifically, the NOSTR has already built-in protocol flow for payments through the lightning network [7], as well as communication flows for a distributed data processing system, which makes it the state of the art protocol for designing a decentralized market for distributed computation and AI directories. Such a network system can mitigate privacy issues, security risks and potentially minimize operational costs of machine learning and language models training by leveraging the structure and capabilities of the NOSTR protocol. We begin by discussing some fundamental aspects of the NOSTR, and the core protocol application designed specifically for the social web.

A prominent challenge for social web applications is that of designing decentralized social networks through peer-to-peer architectures. Additionally, web 3.0 and blockchain implementations will either fail to scale or they will be highly centralized. Tokenized web 3.0 approaches introduce unnecessarily highly counter-party economic risk. Although decentralized payment systems require both blockchain and Proof-of-Work [9,10] (or smart contracts) for asynchronous distributed consensus under faulty processes, it becomes more and more clear that other applications including decentralized social media and marketplaces for computational resources work securely and efficiently without blockchain or global consensus. In contrast, with peer-to-peer networks, web 3.0 and blockchain applications, the NOSTR protocol works efficiently as a social media protocol, because it does not require a majority of nodes of the network to share a common database of the data history. As a consequence, it is also economically efficient since it does not require any form of tokens and it does not hide any counter party risk. In summary, the NOSTR offers a censorship-resistance, open, public network structure, as a well as a digital IDs (DIDs) solution through cryptography and a novel protocol implementation [11].

At the core of the NOSTR, there are two main structural components: clients and relays. For social media applications, users run clients to participate in the network and interact/communicate with other users, through the user interface of the client application. This includes, but it is not limited to retrieving posts from other users, posting notes, accessing profiles of other users, re-posts, likes or even sending payments through the lightning network [7]. To guarantee uniqueness of identities, every user is identified by a public key [12]. Ownership of each identity is guaranteed through a private key that users generate locally [13]. Every post (or action) is signed by using the private key and every client validates these signatures.

Users do not communicate directly with each other, but rather through relays and relays interact with users only. Relays listen only and propagate information. Although relays can not change the content (or action) of users, they can possibly act independently (with any other user or relay) and refuse to propagate pieces of information. The protocol is robust to such censorship attempts, since anyone including users can run one or multiple relays and users can choose an arbitrary set of relays to interact with. Also anyone can run their own relay. Clients retrieve and publish data from relays of their choice. As an example, assume that the User A "follows" a set S of Users, then the client just queries selected relays for posts from the public key of all users within the set S. That is, on startup a client queries data from all relays it knows for all users it follows (for example, all updates from the last day), then displays that data to the user chronologically. An event (user action, for instance a post or a like) can contain any kind of structured data, and clients choose how to display events (defined as any user generated action) of certain kind, while relays can handle them seamlessly. Examples of different kinds are simple note/post (kind #1), re-post (kind #6), reaction (kind #7) [1]. The NOSTR protocol offers flexible structure of messages through a customizable structure and different kind of events.

2. Related Work

The orchestration of middleware systems to distribute computational tasks over wide-ranging networks marks a cornerstone in the evolution of distributed computing. A pivotal advancement in this area has been the development of grid computing. Emerging as a potent global cyber-infrastructure, grid computing is engineered to bolster the forthcoming wave of e-Science applications. By amalgamating large-scale, diverse, and distributed resources, it fosters a cooperative model for the sharing of computational power and data storage on an unprecedented scale. This cooperative paradigm paves the way for unparalleled computational and data management capabilities, enabling the execution of sophisticated scientific, engineering, and academic endeavors previously beyond reach.

Within the diverse ecosystem of grid computing frameworks, centralized systems such as Pegasus [14] and Triana[15] have emerged as paragons. Pegasus, functioning as a workflow mapping engine, abstracts workflow execution from its underlying infrastructure, facilitating the seamless amalgamation of computational tasks across varied grid environments. This system is heralded for its ability to streamline and optimize complex workflows with remarkable efficiency. Conversely, Triana offers a dynamic platform for the composition of applications, harnessing distributed resources through meticulous integration and orchestration. These frameworks underscore the dynamic and adaptable nature of grid computing, illustrating its capacity to tackle complex challenges through the strategic leverage of distributed resources.

Concurrently, volunteer computing has established itself as an integral component within the broader spectrum of grid systems. The Berkeley Open Infrastructure for Network Computing (BOINC)[16] stands as a testament to this model, capturing the essence of volunteer computing by mobilizing the latent processing power of personal computers worldwide for scientific research necessitating extensive computational resources. This model champions the democratization of scientific inquiry, inviting public engagement while significantly enhancing computational prowess without the need for equivalent infrastructural investment.

The "BOINC: A Platform for Volunteer Computing" [17] discourse elucidates the enduring success of the BOINC project, spotlighting its sustained operation over the years as a beacon of collaborative scientific endeavor. This analysis brings to the forefront the prohibitive costs and reliability concerns associated with conventional cloud computing platforms, such as Amazon Web Services (AWS), which, despite their widespread adoption, pose financial and operational challenges for extensive, computation-intensive projects. Furthermore, the discourse addresses the intrinsic challenge of maintaining a robust volunteer base — a critical component for the sustenance of volunteer computing frameworks. The fluctuating levels of engagement and the logistical complexities of coordinating such a decentralized workforce underscore the necessity for innovative solutions to bolster volunteer retention and participation.

To address the lack of incentives and participation in volunteer computing, reward based approaches have been introduced in prior works. For instance, Gridcoin[18] is a proof-of-stake cryptocurrency that rewards participants for contributing computational power to scientific research projects, notably through the Berkeley Open Infrastructure for Network Computing (BOINC) and Folding@Home. However, a proof-of-stake cryptocurrency may introduce several technical issues, since the payment network requires a global synchronized clock [10]. Such issues and significant outages become prominent in many non-proof-of-work blockchain systems (for instance [19]).

While blockchain based approaches aim to provide innovative solutions to incentivize contributions in distributed computing, such approaches differ significantly from the NOSTR protocol. The NOSTR offers a broader set of capabilities, extending beyond the computational resource sharing model to include secure communications and permissionsless interaction between customers and service providers. In contrast with volunteer computing and proof-of-stake cryptocurency models, the decentralized protocol NOSTR integrates bitcoin through the lightning network and a dedicated protocol flow for payments. The NOSTR solves the user-to-user micro-payment problem, incentivizes honest collaborative computation, enables the possibility of an open market through instant, permissionless payments and presents a viable solution to the challenge of volunteer maintenance. These together with the secure and robust digital IDs implementation, make the NOSTR the state of the art protocol, capable of supporting a wide range of decentralized applications and services, offering a pathway to enhance viability, sustainability, and effectiveness.

3. A Decentralized Marketplace for Distributed Computing

As the name of the protocol indicates (Notes and Other Stuff Transmitted by Relays), the type of communication messages and user-to-user interactions are not restricted to simple notes, and they support a variety of alternative applications including direct messaging, file sharing, audio/video streaming, collaborative writing, marketplaces for data processing and more [20]. Another example is that of marketplaces for data processing, where users interact with each other to execute AI algorithms. For the implementation of data processing on the NOSTR protocol, dedicated event kinds have been developed to facilitate distributed data processing (see [21,22]). These implementations have been considered for designing a decentralized AI-directories marketplace of the form "Money-In Data-Out". For instance, a customer can provide some input data, such as text or audio, and pay a service provider to receive a short summary of the input data. More specifically, the kinds in range #5000 − #5999 [21] are reserved for text manipulation jobs including text extraction, summarising text, text generation and translation. Image generation corresponds to kind #5100, and video/audio manipulation corresponds to kinds in range #5200 − #5299.

In this work, we propose an extension of the protocol. Our goal is to design a decentralized marketplace for federated learning and LLMs training. In a nutshell, a customer provides a data-set, model specification and a required payment to a set of service providers. Then the service providers execute the model training, and they return the parameters of the model in order to receive the payment for their work (Money-In AI-Out). In other words, we propose a protocol flow for developing AI-Model Vending Machines (AI-VM), where a customer provides a data-set, a set of specifications, and submits a payment to a set of service providers to acquire a model trained on the input data-set. There are several components of the protocol that allow us to build upon existing protocol rules and design a decentralized marketplace. These include Digital IDs, payments, fast communication though websockets, a flexible protocol structure, also suitable for distributing computing. We proceed by briefly explaining these features.

4. Signatures & Digital IDs

The NOSTR identity is a dual-key cryptographic system [13,23, Schnorr Signatures]. Each user profile is associated with a private (nsec) and a public key (npub). These keys are generated locally without the need of a trusted party. Cryptographic keys are essential to the protocol security, ensuring the protection of user identities and messages. The private key is used to sign messages and other actions by the user, verifying that they originate from the authentic identity owner. On the other hand, the public key is employed to authenticate these messages, confirming that they were signed using the corresponding private key. Identically, to the social media applications, the same DID, private and public keys can be used for alternative applications, for instance private messaging applications, or marketplaces. This allows for a consistent and secure way to authenticate users DIDs and across different applications over the NOSTR protocol.

Nostr Digital Identity And Reputation:

Owing a well known DID or building a solid reputation over the NOSTR offers the benefits of recognizability and trust from other users, similarly to all other online media platforms. Notably, this identity and its accompanying reputation can be seamlessly transferred between different applications. In the context of a marketplace, a good reputation can help a service provider maximize their profit. A high reputation indicates a consistent service, which in turn makes more customers trust the service provider, ensuring a steady stream of business. Alternatively, bad reputation may assist customers to avoid incompetent service providers. As a matter of fact, a reputation system for a decentralized and open marketplace for distributing computing will be valuable. However, this remains out of the scope of this work, but it is an interesting implementation possibility for future extensions of the protocol.

5. Protocol Design for Distributed Computing and AI Directories

The NOSTR protocol supports a primal version of AI directories for data processing in the form of an open and decentralized marketplace [21,22], known as Data Vending Machines. On one side, there are users (customers) who provide some data that should be processed (image, video or text) and they request jobs that can be done with some AI tools, for instance image generation from text, a transcript of a video, a summary of an article. On the other side, there are AI agents (service providers) that offer their services to users, and in exchange for a payment, they deliver the requested results in the form of processed data. This system works in the form of Money-In Data-out; a marketplace for data processing/manipulation by applying AI tools. This design sets the foundations of distributed data processing over the NOSTR protocol and offers a unique protocol structure and flow for additional development.

Our goal is to utilize the existing architecture and to go one step further by extending components of the protocol if necessary, in order to design a decentralized marketplace for Federated Learning (FL) and Large Language Models (LLM) training. In what we propose later (see Section 7), users (customers) provide an input data-set and certain specifications and request a sequence of jobs that aim to train a deep neural network or an LLM model. Then service providers receive the data, model specifications and the requested task (train through regular FL or LLM training (e.g., via the DiLoCo algorithm) [8]), they train the model through multiple rounds and jobs requests, and they return the model parameters. In exchange for their computational resources, the service providers receive recurrent payments from the customer upon delivering valid results. A main difference of our design in comparison with existing approaches is the concept of delivering a trained model instead of processed data; a set of service providers are required to optimize the same model in parallel and deliver valid results through multiple rounds of computation. For that purpose we extend the structure of protocol by introducing new event kinds, a new protocol flow and we deploy the classical FL approach, as well as the DiLoCo algorithm [8] for LLM training. Next, we delve into the existing protocol structure for distributing computing over the NOSTR network. We also emphasize the modifications we introduce to support the Money-IN AI-Out marketplace design.

5.1. *Distributed Computing on the NOSTR Protocol*

Herein, we discuss the existing dedicated protocol messages and protocol flow for distributed computing. We mainly focus on five components that we will utilize later with slight extensions or variations, these include job request events, job result events, job feedback events, events for service provider discoverability and job chaining rules. These type of events consist the core of the communication messages between customer and service providers. We refer the reader to the NOSTR Implementation Possibility - 90 (NIP-90 [24]) for further details.

Basic Protocol Flow for Data Vending Machines (DMVs) [24]

- Customer publishes a job request (e.g. kind:5000, speech-to-text).
- Service Providers may submit job-feedback events (kind:7000, e.g. payment-required, processing, error, etc.).
- Upon completion, the service provider publishes the result of the job with a job-result event (kind:6000).
- At any point, if there is an amount pending to be paid as instructed by the service provider, the user can pay the included bolt11 invoice [25] of the job result event that the service provider has sent to the user.

The above protocol is designed to be deliberately ambiguous. For instance a customer may choose a set of service providers and restrict or not restrict the job request to be completed by this specific set of service providers. Also, a service provider may not start a job until they receive a form of payment, or their response can depend on the other participants reputation based on their public key. The flexibility of the protocol flow allows us to extend parts of it, and implement additional communication steps for the design of on demand LLMs training over a decentralized marketplace. Next we proceed by presenting the existing events as appear in the protocol flow for DMVs.

Job Request (DVMs)

A Job Request for Data Vending Mchines is an event, published by a customer through a set of relays. This event signals that a customer is interested in receiving the result of some kind of computation by an AI tool. The reserved range for job request events is #5000 – #5999. The field with label "content" is empty, while, all the required information should be provided in the field with label "tags". The field "tags" contains all the required information from an AI-agent in order to process the data.

```
{
  "kind": 5xxx // kind in range 5000 – 5999,
  "content": "",
  "tags": [
      ["i", " <data> ", "<input-type>, "<relay>", "<marker>"] ,
      ["output", "<mime-type>"] ,
      ["relays", "wss://..."] ,
      ["bid", "<msat-amount>"] ,
      ["t", "bitcoin"]
  ]
}
```

Event Type 1: Job Request Event with kind in range 5000 -5999

For completeness, we provide additional explanation regarding each tag of the events. We first present the existing events structure. Then we introduce any necessary changes or additions that we will consider in the protocol flow and in the algorithm. For the Job Request event (Event Type 1) there are the following tags: <i>: All the required input data for the job appear here. Specifically, the inputs of the AI algorithm. <data>: The argument for the input. <input-type>: The type of the argument input, it must be one of the following: url: A URL to be fetched of the data that should be processed. event: A Nostr event ID. job: The output of a previous job with the specified event ID. The determination of which output to build upon is up to the service provider to decide (e.g. waiting for a signaling from the customer, waiting for a payment, etc.) text: <data> is the value of the input, no resolution is needed <relay>: The relay where the event/job was published <marker>: An optional field indicating how this input should be used within the context of the job <output>: Expected output format. <param>: Optional parameters for the job. Different job request kind defines this more precisely. (e.g. ["task", "run option", "initial state"]). <bid>: Customer MAY specify a maximum amount (in millisats) they are willing to pay <relays>: List of relays where Service Providers SHOULD publish responses to. p: Service providers the customer is interested in. Other service providers might still choose to process the job.

Job Request (AI VMs)

The existing job request events include a large number of different jobs, dedicated to data processing applications. Such job request have a certain structure (see Event Type 1). For clarity and for efficacy of our approach, we propose to introduce the kinds 8000 - 8999 as Job Request events for federated learning, LLM training and distributed optimization. Although, each kind can correspond at a different algorithm or variation of an algorithm, specific details can be considered by introducing additional fields in the tags. As an example, notice that the predefined structure of Event Type 1, requires a single input data type as we discussed earlier, while additional information may be included in parameters field (<param>).

```
{
  "kind": 8xxx // kind in range $8000 – 8999$,
  "content": "",
  "tags": [
      ["i", "<data>", "<input-type>, "<relay>", "<marker>"] ,
      ["output", "<model-parameters-type>",] ,
      ["relays", "wss://..."] ,
      ["bid", "<msat-amount>"] ,
      ["t", "bitcoin"]
      ["p", "<service-provider(s)-public-key(s)>"]
      ["param", "task", "Inner-or-Outer"] ,
      ["param", "run option", "Fevavg-or-DiLoCo"]
```

```
        ["param", "data_set", "<URL>"] ,
        ["param", "initial/current-model-state", "<raw-data>"]
        ["param", "model", "LLaMA-2"] ,
        ["param", "source_code", "<URL>"] ,
        ["param", "expected_execution_time", "<time>"] ,
        ["param", "recommended_hardware_specification", "<text>"] ,
        ["param", "validation_rules_for_the_output", "<URL>"]
        ["param", "time-out-specification", "max-time>"]
    ]
}
```

Event Type 2: Job Request Event for training AI models, kind in range 8000 -8999

Similarly, for the purpose of training an LLM, the customer needs to provide a training data-set though a URL as input data, and additionally a set of parameters related to the training task, for instance initial model parameters, a URL to the source code of the training or optimization method, model specification (example "LLaMA-2"), rules for validation of the output. To maintain the existing protocol flow as much as possible, we consider the input data as a URL to the training dataset for the initial job request, or the output of a previous job with a specified event job id to declare a job chaining and multiple rounds of training. In both cases (initial or chain job request), we consider all the required information for executing algorithm in the parameters fields for consistency. Below some pieces of that information follow (Event Type 2):

- a source for the code: While the service provider may have its own code, there should be reference to a standard approach that guarantees consistency
- a source for the data: The customer splits and sends an encrypted URL to the service providers. The customer decides how to split the data. The way of splitting the data may be arbitrary.
- job details: expected execution type or hardware that is needed, rules related to time-out conditions.
- Declaration of validation process. This is how the customer will decide if the output is accurate or not.

Through the optimization process we introduce a new job request at each round. We also consider job chaining to speed up the process and most likely to execute the training by assigning the jobs to the same service providers, as long as they provide valid outputs. A running job will be assigned to new service provider, if a previous service provider goes offline, or fails to provide valid result. Finally, the customer has the option to encrypt all the parameters, and only the selected service provider(s) in the tag "p" will be able to decrypt them. We proceed by briefly discussing the Job Chaining rules, for a detailed description we refer the reader to [24, (NIP-90)].

Job Chaining

A Customer has the option to request multiple rounds of jobs to be processed as a chain, where the output of a job is the input of another job. In the context of data processing and DMVs this can be a sequence of processing as a podcast transcription by one service provider and then extracting a summary of the transcription by another. One way to implement this is by specifying as input an event id of a different job with the job type. In general, service providers may begin processing a subsequent job the moment they see the prior job result, but they will likely wait to receive a (partial) payment first.

In the case of federated learning and LLM training such options still exist, however we propose certain protocol rules to minimize the risk introduced by bad actors and malicious users. To do this, we consider necessary for the customer to validate the progress made by each service provider at every single round. After the validation is completed, each service provider who produced successful results receives a partial payment for the work done in the corresponding round. Then the customer updates the current state of model and submits a follow-up job request for the next round (and optionally a partial payment to the service providers upfront). Upon completion of the optimization task at each round, the service providers communicate with the customer through a Job Result event.

Job Result (kind: 6000-6999)

Service providers communicate with the customer through Job Result events, by providing the (optionally) encrypted output of the job. In the case of AI training, the output corresponds to an optimized version of the model parameters. The format of the Job Result event appears in the Event Type 3 above. The tag of the Job Result event include the original job request event id (stringified-JSON), the public key of the customer, amount that the Service Provider is requesting to be paid or an lightning invoice (bolt11 [25,26]), and the optimized model parameter as an encrypted output. Finally, a tag "i" may be considered for additional information related to the initial job request or for validation of the output by the customer.

```
{
  "pubkey": "<service-procider pubkey>",
  "content": "<enctypted payload>",
  "kind": 6xxx // kind in range 6000 - 6999,
  "tags": [
      ["request", " <job-request> "] ,
      ["e", "<job-request-id>", "<relay-hint>"] ,
      ["p", "<customers's-pubkey>"] ,
      ["amount", "requested-payment-amount", "<optional-bolt11>"] ,
      ["i", "additional-information-for-validation", "<information>"]
      ["output", "encrypted: model parameters (and loss for validation purposes)"]
  ],
  ...
}
```

Event Type 3: Job Result Event

Job Feedback (kind:7000)

During the model training process, service providers can communicate and provide feedback for an ongoing job through the Job Feedback event. The Job Feedback events may be considered in order to avoid time-outs and ensure that a service provider has achieved some partial progress. Information about partial progress can be optionally included by introducing additional fields in the tags of the event format in the Event Type 4.

```
{
  "kind": 7000,
  "content": "<empty-or-payload>",
  "tags": [
      ["status", " <status> ", "<extra-info>"] ,
      ["amount", "<requested-payment-amount>", "<bolt11>"] ,
      ["e", "job-request-id", "<relay-hint>"] ,
      ["p", "<customer's-pubkey>"] ,
  ],
  ...
}
```

Event Type 4: Job Feedback Event

The predefined format of the Job Feedback event includes the following tags [24]: <content>: It may be empty, a final job-result or a partial-result (for a job in progress). This field will have an encrypted payload with p tag as key. <amount>: Requested payment amount. <status>: The service provider publishes the current status of the requested job. For instance: payment-required: Service Provider requires payment before continuing. processing: Service Provider is processing the job. error: Service Provider was unable to process the job. success: Service Provider successfully processed the job. partial: Service Provider partially processed the job. The content filed might include a sample of the partial results.

Discoverability

Service Providers have the option to announce availability or to advertise their support for specific job kinds by publishing discoverability events [27, (NIP-89)]. The format structure of the discoverability event appears in the Event Type 6. By including a field tag "i", service providers may provide additional information related to their available computational capabilities, hardware specification or execution time limits.

```
{
  "kind": 31990,
  "pubkey": "<pubkey>",
  "content": [
     \"name\": "Translating DMV" ,
     \"about\": "I'm a DMV speciliazed in translating text."
  ] ,
  "tags": [
       ["k", "5005"], // e.g. translation
       ["t", "bitcoin"]
  ],
  ...
}
```

Event Type 5: Discoverability Event

A possible extension follows:

```
{
  "kind": 31990,
  "pubkey": "<pubkey>",
  "content": [
     \"name\": "Federated Learning AI-VM" ,
     \"about\": "I'm a AI-VM for federated learning."
  ] ,
  "tags": [
       ["k", "8000"], // e.g. optimization methods, federated learning
       ["t", "bitcoin"]
       ["i", "specifications", "hardware", "maximum-execution-time", "model-dimensions-range"]
  ],
  ...
}
```

Event Type 6: Discoverability Event

6. Payments

The NOSTR protocol has an integrated protocol flow for payments; requests for payment, invoice notifications, receipts for payments, as well as payments to certain events or directly to a user or a group of users. The payments take place through the lightning network [7] . For any form of application including social media, Internet-of-Things applications [28] and marketplaces [29], the lightning network enables bitcoin payments with instant settlement and virtually zero transaction fee. These properties together with in-channel messages are ideal for reliable handshaking mechanisms, anti-spam and on the fly payments between multiple job rounds among devices.

To further explain how the payments blend in with the rest of the protocol we proceed with an example. In the case of federated learning the aggregator itself may act as a customer who pays service providers to optimize a model on different subsets of a training data-set through multiple job rounds. Initially, the customer (aggregator) identifies possible service providers through the discoverability event (kind 31990, event type 6), selects a number of them and submits a Job Request event (kind #8000, event type 2). The service providers respond with a Job Feedback event (kind #7000, event type 4) with status payment-required, to announce that they are available to complete the job. Then the customer proceeds by submitting a small amount (of the total) as a payment and the service provider starts the job. Upon delivery of the final or partial output result (Job Result 3 & Job Feedback 4), the customer proceeds with a partial or total payment. This process repeats for each optimization round and for all service providers (assuming that they provide valid results). The ability to break down the payments into partial amounts and pay upon delivery of a valid result significantly minimizes the chance of stealing funds without completing the requested task. This strategy together with a reputation system can make the marketplace robust against adversarial or spam attacks up to a large extend. However, an in depth implementation of such sophisticated strategies remains out of the scope of this work. We discuss strategies for validation of the delivered output in Section 7.1.

6.1. Payments Implementation

Herein, we discuss the payment system which is integrated into the NOSTR protocol. We indicatively explain three components of the payment system; Payment Request, Payment Receipts and Validation of Receipts. For detailed documentation of the payment protocol flow, we refer the reader to implementation 57 [30,31].

Payment Request

A payment (known as zap) request is an event of kind 9734 that is not published to relays, but is instead sent to a lnurl pay callback url [32] of the recipient. The field content may be an optional message to send along with the payment. The event must include the following tags:

- Relays is a list of relays for the wallet of the recipient to publish its payment (zap) receipt to.
- Amount is the amount in millisats the sender intends to pay, formatted as a string.
- Lnurl is the lnurl pay url of the recipient, encoded using bech32 [25] with the prefix lnurl.
- p is the hex-encoded pubkey of the recipient.
- e is an optional hex-encoded event id. Implementations must include this while paying an event rather than a public key.

```
{
  "kind": 9734,
  "content": "Zap!",
  "tags": [
    ["relays", "<wss://relay-domain>", "wss://anotherrelay.com"],
    ["amount", "<msats-amount>"],
    ["lnurl", "<a-static-lightning-invoice>"],
    ["p", "<hex-encoded-pubkey-of-the-recipient>"],
    ["e", "<optional-hex-encoded-event-ID>"]
  ],
  "pubkey": "<NPUB>",
  "created_at": <timestamp>,
  "id": "<event-ID>",
  "sig": "<signature>"
}
```

Event Type 7: Payment Request Event

Payment Receipt

A payment receipt is created by a lightning node when an invoice generated by a payment request is paid. Payment receipts are only created when the invoice description (committed to the description hash) contains a payment request note. When receiving a payment, a series of steps are executed and a NOSTR event of kind 9735 (as appears below; Event Type 8) should be published to the relays declared in the payment request. We refer the reader to the detailed documentation [30] for a complete description of the Payment Receipt event structure and the full protocol flow.

- Get the description for the invoice. This needs to be saved somewhere during the generation of the description hash invoice. It is saved automatically for you with CLN, which is the reference implementation used here.
- Parse the bolt11 description as a JSON nostr event. This SHOULD be validated based on the requirements in Appendix D, either when it is received, or before the invoice is paid.
- Create a nostr event of kind 9735 as described below, and publish it to the relays declared in the zap request.

The following should be true of the zap receipt event:

- The content SHOULD be empty.
- The created_at date SHOULD be set to the invoice paid_at date for idempotency.
- tags MUST include the p tag (zap recipient) AND optional e tag from the zap request AND optional a tag from the zap request AND optional P tag from the pubkey of the zap request (zap sender).
- The zap receipt MUST contain a description tag which is the JSON-encoded invoice description.
- The zap receipt MUST have a bolt11 tag containing the description hash bolt11 invoice.
- SHA256(description) MUST match the description hash in the bolt11 invoice.
- The zap receipt MAY contain a preimage tag to match against the payment hash of the bolt11 invoice. This isn't really a payment proof, there is no real way to prove that the invoice is real or has been paid. You are trusting the author of the zap receipt for the legitimacy of the payment.

```
{
    "id": "<event-ID>",
    "pubkey": "<NPUB>",
    "created_at": <paid-at-date>,
    "kind": 9735,
    "tags": [
      ["p", "<payment-recipient)>"],
      ["P", "<optional-P-tag-from-the-pubkey-of-the-payment-request>"],
      ["e", "<optional-tag-same-as-payment-request-e-tag>"],
      ["bolt11", "<lightning-invoice>"],
      ["description", "<SHA256(description)-from-invoice>"],
      ["preimage", "<payment-preimage>"]
    ],
```

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 April 2024 doi:10.20944/preprints202404.1563.v1

```
    "content": "",
}
```

Event Type 8: Payment Receipt Event

Receipt Validation

The service provider can retrieve payment receipts using a NIP-01 filter [33] and validate each receipt to verify that a customer fetched their invoice and claims payment submission for a certain job. Payments are validated by using the following steps:

- The public key in the payment receipt event must be the same as the public key of the service provider.
- The invoice Amount contained in the bolt11 tag of the payment receipt must be equal to the amount tag of the payment request.
- The lnurl tag of the payment request should be equal the lnurl of the recipient.

We highlight the fact that a receipt is not a proof of payment, and it only proves that some NOSTR user fetched an invoice. As a final step of payment validation, the recipient must verify the payment on their lightning node or payment processor.

7. Money-In AI-Out: Marketplace for Federated Learning and LLMs

Herein, we introduce the protocol flow and algorithmic design for federated learning and LLMs training over the NOSTR protocol. By leveraging the existing structure of the protocol for data processing, and by introducing additional components when it is necessary, we construct an open and decentralized marketplace for training AI models. In this market, a user (customer) provides a dataset, model specifications, the service providers (AI Vending Machines) receive a payment, and they return an AI model trained on this dataset. Customers and service providers communicate through relays, which can be public or private (and self hosted). Also, the communication can be implemented with encrypted information to guarantee privacy for the dataset or for the output of the computation. We proceed by explaining further the protocol flow and then we present a detailed algorithmic implementation for customers and service providers.

We propose a protocol flow that involves multiple job rounds between the customer and service providers. Through multiple job rounds we ensure that there is a valid progress in the optimization process by each of the service providers individually. This allows to consider on the fly payments after validation of the output at each round. If the service providers give an invalid output or if they delay the job beyond a certain amount of time, then the customer proceeds to reassign the job to another service provider without paying for the invalid (or delayed) job output.

In this work we consider two training methods: the classical federated learning [34, FEDAVG] and DiLoCo algorithm [8] for training LLMs. Both of these have an inner and outer optimization part. This allows us to present our approach in parallel. Specifically, we define the job round as the sequence of the following successful publication of events and actions; for the first part (inner optimization): job request, inner optimization computation executed by the service providers, retrieval of the outputs from inner optimization routines by the customer through job result events, validation of the output result. For the second part (Outer optimization): use the latest inner optimization outputs as inputs and execute the outer optimization (through a job request, if needed or locally directly executed by the customer), complete any pending payments and update the model parameters for the next round (if any). Upon validation of the outputs a (partial) payment is submitted by the customer to each of the service providers. To explain this further we provide the sequence of the steps below.

Protocol Flow.

At every job round: Customer (Algorithm 1) publishes a job request (e.g. kind:8000, Federated Learning, inner optimization). The job request may include a list of chosen service providers, and it must include all the required information: the current state of the model (for instance for the first round initial value of model parameters), a link (source) to the data, list of relays where the service providers should publish their job results job feedback at, as well as all the required parameters (see Event Type 2). Service Providers (Algorithm 4) must submit job-feedback events (kind:7000, e.g. partial payment-required) before they start the job with STATUS = payment_required, if they wish a (partial) payment before starting computation (Event Type 4). This requests a partial payment and signals that a service provider is available to start upon receiving the payment. Customer submits a initial payment to service providers (responsible for inner optimization). Upon receiving the partial payment the service providers start executing the inner optimization job (Algorithm 5). Within a fixed time period service providers publish Job Feedback events (Event Type 4) with partial/sample result. If results delivery exceeds a predefined time limit, then customer stops the procedure and seeks for other service providers (responsible for inner optimization). Upon completion, each service provider publishes the output of the job through a Job Result event (kind:6000) (Event Type 3). Customer validates the results/ouputs (Algorithm 3) If the result/ouput is valid then the customer completes the payments to all service providers (responsible for inner optimization) with valid output. If the result is not valid then reassigns the job to a different service provider (Algorithm 2). With successful outputs for all inner optimization jobs: The customer proceeds to the outer optimization (Algorithm 6). If the customer selects to execute the outer optimization locally, then the customer executes the outer optimization (Algorithm 6) with input the output of inner optimization procedures. Else if the customer selects to assign the task to a service provider for the outer optimization, then the customer follows the steps of reassigning the job from SELF to a service provider (Algorithm 2), retrieving and validating the output, and completing any payments identically to the inner optimization job handling as was described above.

At the end of each round the customer updates the model parameters to use them as input for the next round. If the current round is the terminal round then the customer completes any pending payments and ends the procedure with output the final result. At any point, if there is an amount pending to be paid as instructed by the service provider, the customer can pay the lightning-invoice [25,26, bolt11] (see lnurl Event Type 7). We present an implementation of the protocol flow in Section 7.1 and Section 7.2.

7.1. Customer

A complete implementation of the customer routine appears in Algorithm 1. This includes the steps that the customer follows and the events that the customer publishes. First, the customer discovers and requests a set of service providers to run inner optimization part of the job to as we explained in the protocol flow above. Upon successful delivery of outputs, the customer proceeds to the outer optimization. The outer optimization for both FedAvg and DiLoCo is a single or double update of local variables respectively. However we consider the option for the customer to also assign this computation step to a service provider. This can be useful in practice if someone prefers to implement an outer optimizer with higher computational complexity (for instance multiple iterations) and the customer is not able to perform such computations locally. We proceed by explaining further three parts of the main customer routine: how the customer handles disconnections and failures of a service provider to respond (Algorithm 1), how the customer validates the output results given by a service provider at job completion (Algorithm 3), and how the customer reassigns a job to a different service provider if needed (Algorithm 2).

Handling Disconnections and Failures to Respond.

The customer may consider a time-out period. If a service provider did not provide the output within a certain time window (from the time point of receiving the initialization payment), then the customer proceeds to reassign the job to a different service provider (Algorithm 2). As an example of the time-out implementation see Algorithm 1 (lines 13-14). All the information regarding the time-out rules should be included in the Job Request event (Event Type 2).

The definition of failure to respond may vary from one implementation to another. As an example, the customer may choose to receive a valid output within a certain time period. Alternatively the time-out decision may depend on the periodic responses by the service provider through Job Feedback events (Event Type 4), that notify the customer about the status of the job and provide partial results of the computation.

Validation of the Output (Algorithm 3).

Upon receiving a Job Result event (Event Type 3) the customer proceeds to verify the accuracy of the output. In the context of training AI models with FEDAVG and DiLoCl, we consider two conditions that test the decay of the loss on a validation dataset. Specifically, Algorithm 3 requires as input the validation dataset, the current state of the model (model parameters), and a threshold for the two following policies. As a first option, we compare the loss decay progress relative to other service providers (see accuracy against service providers, Algorithm 3). This validation test has been introduced in [35, Accuracy Checking]. As a second option, we consider a test that checks if a moving average of loss decays with a certain rate (Algorithm 3). Alternative accuracy tests and approaches to identify malicious service providers appear in prior works [35–38]. Such implementations may be considered in parallel with our design but remain out of the scope of this work.

Algorithm 1 Customer

```
Require: C_NPUB: Customer's NOSTR public key, DATA_SOURCE: link to the data, NUM_PR: # of
    providers, NUM_JOBS : Total # of jobs for each provider, RELAYS: a set of relays
Ensure: Connection through websockets to all relays in RELAYS    ▷ Where events are published at
Search Routine: Search for NUM_PR AVAILABLE service providers.    ▷ Event Type 6
Search Routine returns: PROVIDERS with NUM_PR NPUBs entries
Split the data to NUM_PR parts and generate the list DATA_SOURCES with NUM_PR sources to
each data segment    ▷ DATA_SOURCES($i$) is the data source of the $i_{\text{th}}$ provider
Initialize the optimization parameters and model: model_parameters, model, ...
Publish Job Requests KIND: 8000 with all the required information    ▷ Event Type 2
Wait for Job Feedback KIND: 7000 and payment-request from service providers    ▷ Event Type 4
Submit an initialization payment to PROVIDERS
for job in (1,NUM_JOBS) do
    Initialize done ← FALSE
    while done $\neq$ TRUE do
        Wait some time
        Fetch Job Feedback events (KIND:7000) from each provider in PROVIDERS
        if Time-out occurs then
            Call Reassign (Algorithm 2) to replace all the delayed service providers, update
            PROVIDERS
        if JOB_STATUS = success for all providers in PROVIDERS then done = TRUE
    Fetch Job Results (KIND: 6000) and "outputs" from all providers in PROVIDERS ▷ Event Type 3
    for service_provider in PROVIDERS call the Validation routine (Algorithm 3) and do
        while Validation_Test = FAIL for the service_provider do
            Reassign the job of service_provider; call Reassign (Algorithm 2), update PROVIDERS
        Submit (partial) payments to service_provider and publish Payment Receipt (KIND: 9735)
    if OUTER_OPTIMIZER = SELF then
        Run the Outer_Optimization routine (Algorithm 6) with input the current updates "outputs"
    else
        Reassign the Outer Optimization from SELF to a service provider (Algorithm 2)
        Submit (partial) payment to OUTER_OPTIMIZER and publish Payment Receipt (KIND:
        9735)
    Update model_parameters using the "output" of the OUTER_OPTIMIZER
    if stopping condition is satisfied then
        break
    else if job $\leq$ NUM_JOBS then
        Publish Job Requests KIND: 8000 with all the required information    ▷ Event Type 2
Finalize payments to every service_provider in PROVIDERS and publish receipts    ▷ Event Type 8
return model_parameters
```

Reassigning the Job to a Different Service Provider (Algorithm 2).

Under a time-out or a failed validation instance, the customer reassigns the job to a different service provider. We provide an example of such implementation in Algorithm 2. Further, under perpetual time-outs and invalid output detection, the customer continues to reassign the job, until a service provider delivers a valid output. Then the routine returns the public key of new service provider and the customer updates the list of the service providers under consideration.

Algorithm 2 Reassign the job of a Service Provider to a different Service Provider

```
Require: SP_NPUB: public key of the service provider that will be be replaced, <computation information & time>, RUN_OPTION: <FEDAVG or DiLoCo>, DATA: DATA_SOURCES , PROVIDERS, TASK: <INNER or OUTER>
Ensure: Connection through websockets to all relays in RELAYS    ▷ Where events are published at
Search Routine: Search for an AVAILABLE service provider for KIND: 8000.    ▷ Event Type 6
Search Routine returns: A single public key: SP_NPUB_NEW
Publish a Job Request KIND: 8000: Identical to the last job request for the service provider SP_NPUB, but with updated field "p" to SP_NPUB_NEW    ▷ Event Type 2
Wait for Job Feedback KIND: 7000 and payment-request from service provider SP_NPUB_NEW
Submit an initialization payment to SP_NPUB_NEW
while JOB_STATUS $\neq$ success for SP_NPUB_NEW do
    Wait some time
    Fetch Job Feedback events (KIND:7000) from service provider SP_NPUB_NEW    ▷ Event Type 4
    if time-out occurs then break and Reassign again to replace SP_NPUB_NEW
    if JOB_STATUS = success for SP_NPUB_NEW then
        Fetch Job Result (KIND: 6000) and "output" from SP_NPUB_NEW    ▷ Event Type 3
        Call the Validation Routine
        if Validation_Test = PASS then
            return SP_NPUB_NEW, "output"
        else if Validation_Test = FAIL then break and Reassign again to replace SP_NPUB_NEW
```

Algorithm 3 Validation of the Service Provider Output

```
Require: SP_NPUB: service provider public key, RUN_OPTION: <FEDAVG or DiLoCo>, validation-dataset: $\mathcal{D}_{\text{test}}$, model_parameters: $\theta_{\text{global}}$, the last "outputs" of the Inner Optimization jobs: $\theta_{\text{Inner}}$ and the increments $\Delta\theta_{\text{Inner}}$, policy thresholds: $\gamma_t, \beta_t$, TEST_TYPE
Ensure: A history of model parameters is available. (TEST_TYPE = "B" requires the previous $\tau_c$ model parameters for test condition)
if TEST_TYPE = "A" then    ▷ Check accuracy against other service providers [35]
    $\tilde{\theta} \leftarrow \theta_{\text{global}} + \Delta\theta_{\text{Inner}}^{\text{SP\_NPUB}}$
    $\tilde{\theta}_{G\backslash\text{SP\_NPUB}} \leftarrow \theta_{\text{global}} + \sum_{\text{npub}\in\text{PROVIDERS}} \Delta\theta_{\text{Inner}}^{\text{npub}}$
    if $\sum_{z\in\mathcal{D}_{\text{test}}} \ell(\tilde{\theta}, z) - \ell(\tilde{\theta}_{G\backslash\text{SP\_NPUB}}, z) > \gamma_t$ then
        SP_NPUB is malicious
        return FAIL
else if TEST_TYPE = "B" then    ▷ Check progress over time (with time lag)
    $\theta^{\tau} \leftarrow \theta^{\text{SP\_NPUB}}_{\text{Inner at time }\tau}$, for all $\tau \in [t-\tau_c, t]$    ▷ For Outer: replace $\theta^{\text{SP\_NPUB}}_{\text{Inner at time }\tau}$ with $\theta^{\text{SP\_NPUB}}_{\text{global at time }\tau}$
    if $\frac{1}{\tau_c+1}\sum_{\tau=t-\tau_c}^{t}\sum_{z\in\mathcal{D}_{\text{test}}} \ell(\theta^{\tau}, z) > \beta_t$ then    ▷ Check if a moving average of the loss decays
        return FAIL
return PASS
```

7.2. Service provider

Algorithm 4 is an implementation of the routine for the service providers. For announcement and advertisement of their service, the providers publish a discoverability event of kind 31990 (Event Type 6). Initially, the service providers wait until they receive a job request. Then they signal initiation through a job feedback event. Upon receiving any required payments they proceed with the optimization routine, as the customer has requested. The optimization procedure may be one of the run options (FEDAVG or DiLoCo) and the task for either run option may be Inner or Outer optimization. Depending on the run option the service providers calls the corresponding algorithmic routine (Algorithm 5 or Algorithm 6). Periodically the service provider announces the status of the job through a Job Feedback event and provides a progress of the computation, for instance see Algorithm 5.

After the completion of the optimization routine the service providers respond with a job result event, and they expect any pending payment to be settled. If the customer does not complete payment(s) after a reasonable time period despite the delivery of accurate outputs, then the service providers may stop receiving job request from the specific customer immediately.

Algorithm 4 Service provider

```
Require: SP_NPUB: service provider public key, <computation information & time>, <lightning address>
Ensure: Connection through websockets to all relays in RELAYS          ▷ Where events are published at
Publish an event (NIP-89) of kind 31990 for discoverability           ▷ Event Type 6
while TRUE do
    Routine: Search for JOB REQUEST for provider SP_NPUB
    if JOB REQUEST with appropriate KIND was found then               ▷ Event Type 2
        Publish Job Feedback (KIND: 7000) with STATUS = payment_request ▷ Event Type 4
        Wait until the initial payment is submitted, fetch and validate Payment Receipt (KIND: 9735)
        if payment = successful then
            Publish Job Feedback (KIND: 7000) with STATUS = processing  ▷ Event Type 4
        else
            Publish Job Feedback (KIND: 7000) with STATUS = error
        Fetch model_parameters, DATA_SOURCES(SP_NPUB),  RUN_OPTION, TASK from JOB REQUEST
        if RUN_OPTION = FEDAVG and TASK = INNER then
            Call Inner_Optimization( model_parameters, DATA_SOURCES , FEDAVG )
        else if RUN_OPTION = FEDAVG and TASK = OUTER then
            Call Outer_Optimization( model_parameters, DATA_SOURCES , FEDAVG )
        else if RUN_OPTION = DiLoCo and TASK = INNER then
            Call Inner_Optimization( model_parameters, DATA_SOURCES , DiLoCo )
        else if RUN_OPTION = DiLoCo and TASK = INNER then
            Call Outer_Optimization( model_parameters, DATA_SOURCES , DiLoCo )
        Publish Job Feedback (KIND: 7000) with STATUS = success         ▷ Event Type 4
        Publish a JOB RESULT event with "output" the result of the optimization ▷ Event Type 3
```

Federated Averaging vs Distributed Low-Communication Training of LLMs (DiLoCo)

The federated averaging algorithm [34] (FEDAVG) is a distributed and communication efficient learning method of AI models from decentralized data. This decentralized computation is known as federated learning; a set of different devices learn a shared model by aggregating locally-computed updates. In our design we consider as one optimization approach the FEDAVG. The customer provides a common model specification and the data to the servicce providers. The service providers have the role of different devices, they obtain a subset of the data-set and they train the shared model. Upon completion of the computation they return the optimized model parameters, and they receive a payment.

Although the FEDAVG is a successful approach for training neural netowrks and AI in a decentralized fashion, we also consider a variant of FEDAVG that has numerous advantages over the classical approach. Specifically, the DiLoCo algorithm [8] enables training of LLMs under constrained communication. In fact, the inner part of optimization (jobs by Service providers) requires the greatest amount of computation, while the request for a new job is being decreased significantly. This provides additional flexibility to our design since it minimizes the number of job requests by the customer. Nevertheless, the service providers have the option to frequently provide information (e.g. job status, partial output) to the customer through the Job Feedback events. We proceed by presenting the implementation of the Inner Optimization and Outer Optimization routines for both FEDAVG and DiLoCo algorithms.

Inner Optimization (Algorithm 5)

The Inner Optimization routine provides the implementation of the inner part of the computation. A service provider runs the inner part of FEDAVG (example SGD) or DiLoCo (AdamW [39, Algorithm 2]). The run option is provided by the customer through the job request event. The algorithm returns the updated model parameter.

Algorithm 5 Inner_Optimization

```
Require: RUN_OPTION: <FEDAVG or DiLoCo>, DATA: DATA_SOURCES, model_parameters: $\theta_{\text{global}}$
Ensure: Connection through websockets to all relays in RELAYS    ▷ Where events are published at
$\theta \leftarrow \theta_{\text{global}}$
if RUN_OPTION = FEDAVG then    ▷ Run the inner part of FEDAVG
    data ← DATA_SOURCES(SP_NPUB)
    $\mathcal{B}$ ← Split data into batches
    for epoch $e \in (1, E)$ do
        Publish Job Feedback (KIND: 7000) with STATUS = processing    ▷ Event Type 4
        for batch $\boldsymbol{b} \in \mathcal{B}$ do
            $\theta \leftarrow \theta - \eta \nabla \ell(\theta; \boldsymbol{b})$
else if RUN_OPTION = DiLoCo then
    for epoch $e \in (1, E)$ do
        Sample a batch $\boldsymbol{b}$ of data from DATA_SOURCES(SP_NPUB)
        Run AdamW($\theta, \nabla \ell(\theta, \boldsymbol{b})$)    ▷ AdamW [39, Algorithm 2]
        Periodically publish Job Feedback (KIND: 7000) with STATUS = processing ▷ Event Type 4
        $\theta \leftarrow$ AdamW($\theta, \nabla \ell(\theta, \boldsymbol{b})$)
return $\theta$    ▷ Returns $\theta_{\text{Inner}}^{(\text{SP\_NPUB})}$
```

Outer Optimization (Algorithm 6)

For the Outer Optimization routine, the customer has the option to run the outer optimization, or assign it to a service provider. Then the customer (or a service provider) runs the outer part of FEDAVG (example aggregation step) or DiLoCo [8,40, Nesterov Momentum]. The run option is determined by the customer. The algorithm returns the updated model parameter.

Algorithm 6 Outer_Optimization

```
Require: RUN_OPTION: <FEDAVG or DiLoCo>, RUN_OPTION: <FEDAVG or DiLoCo>, DATA:
    DATA_SOURCES, model_parameters: $\theta_{\text{global}}$, the last "outputs" of the Inner Optimization jobs: $\theta_{\text{Inner}}$
Ensure: Connection through websockets to all relays in RELAYS    ▷ Where events are published at
if RUN_OPTION = FEDAVG then    ▷ Run the aggregation (outer) part of FEDAVG
    $\theta_{\text{global}} = \frac{1}{\text{NUM\_PR}} \sum_{k=1}^{\text{NUM\_PR}} \eta_k \times \theta_{\text{Inner}}^{(k)}$    ▷ For some weights $\eta_k$
else if RUN_OPTION = DiLoCo then    ▷ Run the Outer part of DiLoCo [8]
    $\Delta\theta_{\text{Outer}} = \frac{1}{\text{NUM\_PR}} \sum_{k=1}^{\text{NUM\_PR}} \eta_k \times (\theta_{\text{global}} - \theta_{\text{Inner}}^{(k)})$
    Run Nesterov Momentum:    ▷ See FedMom [40, Algorithm 3]
    (Periodically) publish Job Feedback (KIND: 7000) with STATUS = processing    ▷ Event Type 4
    $\theta_{\text{global}}$ ← Nesterov_Momentum ($\theta_{\text{global}}$ , $\Delta\theta_{\text{Outer}}$)
return $\theta_{\text{global}}$
```

8. Conclusion

The NOSTR protocol is a relatively new paradigm shift for reliable and decentralized social web applications. In this work we introduced a design and protocol flow for LLM and AI training on decentralized marketplace system, that builds upon the NOSTR. The fast communication though websockets and the flexibility of development of the protocol enables the potential for a competitive, open and censorship resistant computation network, at which users (customers) provide data and a payment, and they receive a trained AI model. The integrated payment protocol flow rules and the lightning network offer fast, reliable and with zero transaction fee payments without a trusted party. As a result, service providers can maximize their profit and compete within an open market. Further, this payment system supports microtranscactions with instant settlement, which is essentially useful as anti-spam mechanism, reliable form of handshaking between customers and service providers, on the fly payments upon receiving partial results of the computation, or upon validation of the final output.

We considered a simple but effective mechanism for validation of the output, by periodically checking the progress of the trained model through a validation dataset. However, we believe that other alternative solutions for verifying accuracy of the computation may be considered as well. Such robust approaches can be designed by considering smart contracts and the BitVM mechanism [41], which enables the possibility of any computable function to be verified. Inclusion of such mechanisms for verification, together with a reputation system, or application specific protocol flow variations for alternative AI algorithms such as DeepSpeed [42] & Zero Redundancy Optimizer [43] or distributed LLM deployment and inference [44–51] could be potentially considered tasks for future work.

9. Funding

- https://tmf.cio.gov/ai/#our-expedited-ai-pilot-process
- https://new.nsf.gov/focus-areas/artificial-intelligence
- https://www.usgrants.org/business/artificial-intelligence-researchers
- https://www.nersc.gov/research-and-development/data-analytics/generative-ai-for-science-call-for-proposals-ay2024/
- https://new.nsf.gov/funding/opportunities/national-artificial-intelligence-research
- https://www.scaleai.ca/projects/
- https://www.cooperativeai.com/grants/cooperative-ai
- https://hessian.ai/entrepreneurship/laisf/

10. Notes

To do

- I made some changes in the text see blue
- Validation to validation an trexei ston customer isws na einai ligo bottleneck. 8a kanoume k peiramata? pistevw a3izei na doume posi wra pernei afto, giati an gia ka8e result pou trexeis kaneis k validation locally isws na einai full douleia apo mono tou. i na exeis allon service provider gia validation :P Ideally the customer wants to know if the result is valid before completing the payment, maybe the validation has to be simple and smart, other service providers may be responsible for validation, we may include this as a discussion instead of into the code
- parakatw polu swsta to paper anaferetai se service providers failures k pws na ta diaxeirizetai to sistima. pistevw prepei na pros8esoume mia extra paragrafo gia to pws diaxeirizetai to sistima ta failures to customer. afti ti stigmi an pesei o customer ti simvainei sto sistima? The service provider may consider a time-out period as well if the custoemr stops responding and submitting payments after some time the service provider aborts and looks for other job requests from different customers to feedback pou sou girnaei o service provider pou apo8ikevetai? It should be the same as how a client like damus stores the feed, this has by default a customizable rate limit (see also workshop build a client from scratch by react), also there are customizable filters

to keep only the relevant eventstopika se mia ram? 8a pesei isws polu douleia se afto ton komvo. 8a 8elei ligo rate limiting gia na min girisoun polla request mazi b) checkpointing gia na min pesei o komvos stin mesi tou training?

- Consider any changes related to the above into the code as well
- Discuss implementation and development for simulations

Simulations

- Startup time
- scalability
- comparison with existing alternative options: quantify trade-offs

CLI clients and tools:

- https://github.com/mattn/algia
- https://github.com/fiatjaf/noscl?tab=readme-ov-file
- https://github.com/fiatjaf/nak
- development kit (ndkit): https://ndkit.com/
- nostr tools https://github.com/nbd-wtf/nostr-tools

Optimization algorithms for (service providers):

- FEDAVG https://github.com/alexbie98/fedavg?tab=readme-ov-file
- DiLoCo https://github.com/xrsrke/pipegoose

References

1. domain., P. NOSTR - Notes and Other Stuff Transmitted by Relays. https://github.com/nostr-protocol/nostr, 2023.
2. Briugeman, A. Nostr Real Time Statistics. https://stats.nostr.band, 2023.
3. domain., P. Bluesky Architecture. https://docs.bsky.app/docs/advanced-guides/federation-architecture, 2023.
4. Kleppmann, M.; Frazee, P.; Gold, J.; Graber, J.; Holmgren, D.; Ivy, D.; Johnson, J.; Newbold, B.; Volpert, J. Bluesky and the AT Protocol: Usable Decentralized Social Media. *arXiv preprint arXiv:2402.03239* **2024**.
5. Gilbert, S.; Lynch, N. Perspectives on the CAP Theorem. *Computer* **2012**, *45*, 30–36.
6. Wei, Y.; Tyson, G. Exploring the Nostr Ecosystem: A Study of Decentralization and Resilience. *arXiv preprint arXiv:2402.05709* **2024**.
7. Poon, J.; Dryja, T. The bitcoin lightning network: Scalable off-chain instant payments. *https://lightning.network/* **2016**.
8. Douillard, A.; Feng, Q.; Rusu, A.A.; Chhaparia, R.; Donchev, Y.; Kuncoro, A.; Ranzato, M.; Szlam, A.; Shen, J. DiLoCo: Distributed Low-Communication Training of Language Models. *arXiv preprint arXiv:2311.08105* **2023**.
9. Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review* **2008**.
10. Dembo, A.; Kannan, S.; Tas, E.N.; Tse, D.; Viswanath, P.; Wang, X.; Zeitouni, O. Everything is a race and Nakamoto always wins. Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 859–878.
11. Liu, Y.; Kishore, K.N. Sovereign Individual System (SIS): An Autonomous Digital Platform for Sovereign Individuals. Proceedings of the Future Technologies Conference. Springer, 2023, pp. 142–153.
12. domain., P. NIP-19: NOSTR - Notes and Other Stuff Transmitted by Relays. https://github.com/nostr-protocol/nips/blob/master/19.md, 2023.
13. Pieter Wuille, Jonas Nick, T.R. Schnorr Signatures for secp256k1. https://bips.xyz/340, 2019.
14. Deelman, E. Mapping Abstract Complex Workflows onto Grid Environments. *Journal of Grid Computing* **2003**, *1*, 25–39. doi:10.1023/a:1024000426962.
15. Taylor, I.; Shields, M.; Wang, D. Resource Management of Triana P2P Services. *Grid Resource Management* **2003**.
16. Anderson, D.; Korpela, E.; Walton, R. High-performance task distribution for volunteer computing. First International Conference on e-Science and Grid Computing (e-Science'05), 2005, pp. 8 pp.–203. doi:10.1109/E-SCIENCE.2005.51.
17. Anderson, D.P. BOINC: A Platform for Volunteer Computing. *Journal of Grid Computing* **2019**, *18*, 99–122. doi:10.1007/s10723-019-09497-9.
18. The Gridcoin Network and Protocol Overview **2018**.
19. Yakovenko, A. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper* **2018**.
20. Apps, N. Nostr Apps. https://www.nostrapps.com/, 2023.
21. domain., P. Data Vending Machines. https://github.com/nostr-protocol/data-vending-machines, 2023.

22. domain., P. VenData. https://vendata.io, 2023.
23. Maxwell, G.; Poelstra, A.; Seurin, Y.; Wuille, P. Simple schnorr multi-signatures with applications to bitcoin. *Designs, Codes and Cryptography* **2019**, *87*, 2139–2164.
24. domain., P. NIP-90: NOSTR - Notes and Other Stuff Transmitted by Relays. https://github.com/nostr-protocol/nips/blob/master/90.md, 2023.
25. Antonopoulos, A.M.; Osuntokun, O.; Pickhardt, R. *Mastering the lightning network*; " O'Reilly Media, Inc.", 2021.
26. Andreas M. Antonopoulos, Olaoluwa Osuntokun, R.P. Lightning Payment Requests. https://github.com/lnbook/lnbook/blob/develop/15_payment_requests.asciidoc, 2023.
27. domain., P. NOSTR Implementation Possibillities, 89. https://github.com/nostr-protocol/nips/blob/master/89.md, 2023.
28. O'Hare, J.J.; Fairchild, A.; Ali, U. Money & Trust in Digital Society: Bitcoin, Nostr, Stablecoins, Digital Objects and Generative AI in B2B Spatial Mixed Reality. *arXiv preprint arXiv:2207.09460* **2022**.
29. Robert, J.; Kubler, S.; Ghatpande, S. Enhanced Lightning Network (off-chain)-based micropayment in IoT ecosystems. *Future Generation Computer Systems* **2020**, *112*, 283–296.
30. domain., P. NOSTR Implementation Possibillities, 57. https://github.com/nostr-protocol/nips/blob/master/57.md, 2023.
31. Zap, N. JeffG. https://nostr.how/en/zaps, 2023.
32. domain., P. LNURL. https://github.com/GaloyMoney/lnurl-pay, 2023.
33. domain., P. NIP-01: NOSTR - Notes and Other Stuff Transmitted by Relays. https://github.com/nostr-protocol/nips/blob/master/01.md, 2023.
34. McMahan, B.; Moore, E.; Ramage, D.; Hampson, S.; y Arcas, B.A. Communication-efficient learning of deep networks from decentralized data. Artificial intelligence and statistics. PMLR, 2017, pp. 1273–1282.
35. Bhagoji, A.N.; Chakraborty, S.; Mittal, P.; Calo, S. Analyzing Federated Learning through an Adversarial Lens. Proceedings of the 36th International Conference on Machine Learning; Chaudhuri, K.; Salakhutdinov, R., Eds. PMLR, 2019, Vol. 97, *Proceedings of Machine Learning Research*, pp. 634–643.
36. Demartis, M. Adversarial Attacks in Federated Learning, 2022.
37. Rathee, M.; Shen, C.; Wagh, S.; Popa, R.A. Elsa: Secure aggregation for federated learning with malicious actors. 2023 IEEE Symposium on Security and Privacy (SP). IEEE, 2023, pp. 1961–1979.
38. Awan, S.; Luo, B.; Li, F. Contra: Defending against poisoning attacks in federated learning. Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26. Springer, 2021, pp. 455–475.
39. Loshchilov, I.; Hutter, F. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* **2017**.
40. Huo, Z.; Yang, Q.; Gu, B.; Huang, L.C.; others. Faster on-device training using new federated momentum algorithm. *arXiv preprint arXiv:2002.02090* **2020**.
41. Linus, R. BitVM: Compute Anything on Bitcoin. *bitvm.org* **2023**.
42. Rasley, J.; Rajbhandari, S.; Ruwase, O.; He, Y. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2020, pp. 3505–3506.
43. Rajbhandari, S.; Rasley, J.; Ruwase, O.; He, Y. Zero: Memory optimizations toward training trillion parameter models. SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2020, pp. 1–16.
44. Fan, T.; Kang, Y.; Ma, G.; Chen, W.; Wei, W.; Fan, L.; Yang, Q. Fate-LLM: A industrial grade federated learning framework for large language models. *arXiv preprint arXiv:2310.10049* **2023**.
45. Lu, L.; Dai, C.; Tao, W.; Yuan, B.; Sun, Y.; Zhou, P. Exploring the Robustness of Decentralized Training for Large Language Models. *arXiv preprint arXiv:2312.00843* **2023**.
46. Tang, Z.; Wang, Y.; He, X.; Zhang, L.; Pan, X.; Wang, Q.; Zeng, R.; Zhao, K.; Shi, S.; He, B.; others. FusionAI: Decentralized Training and Deploying LLMs with Massive Consumer-Level GPUs. *arXiv preprint arXiv:2309.01172* **2023**.
47. Agrawal, A.; Panwar, A.; Mohan, J.; Kwatra, N.; Gulavani, B.S.; Ramjee, R. Sarathi: Efficient LLM inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369* **2023**.
48. Shen, H.; Chang, H.; Dong, B.; Luo, Y.; Meng, H. Efficient LLM inference on cpus. *arXiv preprint arXiv:2311.00502* **2023**.
49. Spector, B.; Re, C. Accelerating LLM inference with staged speculative decoding. *arXiv preprint arXiv:2308.04623* **2023**.
50. Cai, T.; Li, Y.; Geng, Z.; Peng, H.; Lee, J.D.; Chen, D.; Dao, T. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774* **2024**.
51. Del Corro, L.; Del Giorno, A.; Agarwal, S.; Yu, B.; Awadallah, A.; Mukherjee, S. Skipdecode: Autoregressive skip decoding with batching and caching for efficient LLM inference. *arXiv preprint arXiv:2307.02628* **2023**.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.