Accepted to appear in IEEE Transactions on Mobile Computing, 2025

Domain-Specific Transport Protocols for In-Network Processing at the Edge: A Case Study of Accelerating Model Synchronization

Shouxi Luo, Peidong Zhang, Xin Song, Pingzhi Fan, Huanlai Xing, Long Luo, Hongfang Yu

***Abstract*—Nowadays, cross-device federated learning (FL) is the key to achieving personalization services for mobile users and has been widely employed by companies like Google, Microsoft, and Alibaba in production. With the explosive growth in the number of participants, the central FL server, which acts as the manager and aggregator of cross-device model training, would get overloaded, becoming the system bottlenecks. Inspired by the emerging wave of edge computing, an interesting question arises: *Could edge clouds help cross-device FL systems overcome the bottleneck?* This article provides a cautiously optimistic answer by proposing INP, a FL-specific In-Network Processing framework to achieve the goal. As in-network processing has broken the end-to-end principle of the involved communication and lacks the support of transport protocols, the key is to design domain-specific transport protocols for INP. To fill the gap, we propose the novel Model Download Protocol of MDP and Model Upload Protocol of MUP. With MDP and MUP, edge cloud nodes along the paths in INP can easily eliminate duplicated model downloads and pre-aggregate associated gradient uploads for the central FL server, thus alleviating its bottleneck effect, and further accelerating the entire training progress significantly.**

***Index Terms*—Federated learning, in-network processing, edge computing, transport protocol**

I. Introduction

By sharing the models rather than the raw privacy-sensitive data, cross-device Federated Learning (FL) has been widely deployed in production for various personalization services like *on-device item ranking*, *next-word prediction*, *content suggestions for on-device keyboards*, and *real-time e-commerce recommendations* [1], [2], [3], [4], [5]. As an emerging distributed machine learning (DML) approach, cross-device FL enables end devices like mobile phones to cooperatively train models in an iterative way [6], [7]: In each round of training, a set of dynamically selected *end devices* (EDs) first download the current model from the central FL server (FLS) to launch the on-device training; once completed, they then upload their local model updates (i.e., gradients) back to the FL server; finally, the FL server would aggregate received gradients to obtain a new model to iterate. Obviously, with an increase in the number of selected end devices, the central FL server would become the bottleneck of the entire FL system. Optimizing the performance of FL systems, especially removing the bottleneck effects of the FL server, becomes the key to supporting very large-scale federated learning tasks [3].

To enhance the performance of the FL server, in this article, we analyze the benefits of deploying in-network processing nodes at edge clouds to provide cache and aggregation services for large-scale cross-device federated learning, and propose the FL-specific **I**n-**N**etwork **P**rocessing framework of INP. We find that using edge nodes for in-network model caching and gradient aggregation brings a lot of benefits to FL systems. Nevertheless, a pair of new domain-specific transport protocols breaking the conventional wisdom of end-to-end transport semantics are required, as existing protocols like the raw TCP and UDP could not provide the needed features [8]. To fill the gap, along with INP, we further propose a suite made up of the Model Download Protocol (MDP), Model Upload Protocol (MUP), and the involved progress synchronization, congestion control, and cache management algorithms, to release the power of edge-aided in-network processing. Consider that multiple end devices residing in the same region are performing the same FL task. The key of INP is to let edge cloud nodes cache recently downloaded model chunks for possible subsequent duplicated requests from other end devices, and pre-aggregate the gradient uploads from different end devices in a short time interval. For a FL task involving m end devices, our proposed INP, together with MDP and MUP, is able to reduce both the traffic load of the FL server and the time needed by model downloads and gradient uploads, from the magnitudes of $O(m)$ to $O(1)$ at most.

Distinguished from the alternative idea of using edge servers as local parameter servers for cross-device FL [3], [9], both the cache of model parameters and the aggregation of gradients in INP can be implemented as specific types of in-network processing services [10]. They can be built upon existing Network Function Virtualization (NFV) platforms [11] to provide best-effort acceleration service for cross-device FL traffic at the edge. As Section II-C will show, such a design is easy-to-manage, generic, future-proof, and achieves fine-

An earlier version of this paper was presented in part at the ICC 2022 - IEEE International Conference on Communications [DOI: 10.1109/ICC45855.2022.9838381] *(Corresponding author: Shouxi Luo.)*

Shouxi Luo is with the School of Computing and Artificial Intelligence, Southwest Jiaotong University, Chengdu 611756, China, and also with the Manufacturing Industry Chains Collaboration and Information Support Technology Key Laboratory of Sichuan Province, Southwest Jiaotong University, Chengdu 611756, China.

Peidong Zhang and Xin Song are with the School of Computing and Artificial Intelligence, Southwest Jiaotong University, Chengdu 611756, China.

Pingzhi Fan is with the Key Laboratory of Information Coding and Transmission, CSNMT Int Coop. Res. Centre, Southwest Jiaotong University, Chengdu 611756, China.

Huanlai Xing is with the School of Computing and Artificial Intelligence, Southwest Jiaotong University, Chengdu 611756, China.

Long Luo and Hongfang Yu are with the School of Information and Communication Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China.

grained resource usage. A lot of recent papers have also put forward the vision of enhancing the performance of mobile and IoT applications by deploying NFV-enabled services in edge clouds [12], [13]. However, they mainly focus on the abstracted coarse-grained resource allocation problem, without considering the detail of how functions could be implemented and supported by the underlying network. Indeed, as this article will show, by using the domain-specific transport protocols of MDP and MUP, INP would achieve very flexible resource allocation at the granularity of per-packet. For in-network model cache, the well-known data-centric network architecture of Named Data Networking (NDN) meets the requirements naturally. However, NDN has not been supported by today's Internet yet because of its clean-slate, incompatible design [14]. Instead, INP only relies on widely deployed techniques and thus is readily deployable.

In summary, our contributions are five-fold:

- A thorough analysis that identifies the benefits and challenges of applying edge-cloud-based in-network processing for large-scale cross-device FL (Section II).
- INP, a FL-specified In-Network Processing framework enabling cross-device FL systems highly scalable and accelerating training processes significantly (Section III).
- MDP, a UDP-based domain-specific transport protocol along with a suite of progress synchronization, congestion control, and cache management schemes achieving efficient model downloads for FL jobs, by using the in-network cache services provided by edge boxes (Section IV).
- MUP, a UDP-based domain-specific transport protocol along with a suite of flow control, progress synchronization, congestion control, and cache management schemes achieving efficient gradient uploads and aggregations for FL jobs, by using the in-network aggregation services provided by edge boxes (Section V).
- Extensive fine-grain simulation studies verifying the efficiency of the designs of both MDP and MUP protocols and confirming the significant benefits of INP for the model synchronization of FL jobs (Section VII).

The rest of the paper is organized as follows. We first introduce and analyze the related background and motivation in Section II. Then, we overview the design of INP in Section III, and explain the details of MDP and MUP in Sections IV and V, respectively. After that, we further describe how MDP and MUP gracefully deal with hierarchical FL characteristics in Section VI, and assess their performances in Section VII. Then, Section VIII discusses related work; and finally, Section IX concludes the article.

II. Background and Motivation

FL applications today can be broadly categorized into two kinds, namely cross-device FL and cross-silo FL [5]. The classification is based on whether the participating clients are resource-limited devices, e.g., mobile phones, vehicles, or source-abundant clusters, e.g., private clusters owned by financial or medical organizations, respectively [5]. In this article, we focus on the optimization of cross-device FL systems. In these systems, a large number of end devices are dynamically selected to train a global model iteratively over wireless and WAN connections, with the assistance of a centralized FL server as Figure 1 shows.

In the following, we first overview the characteristics of cross-device FL (Section II-A) and summarize the benefits of communication optimization on it (Section II-B), then discuss why in-network processing at the edge is more promising than the alternative idea of edge-based hierarchical FL server (Section II-C); after that, we review why existing in-network aggregation (INA) solutions designed for intra-datacenter DML could not solve the problem we focus on (Section II-D), and finally summarize the design challenges of our solution (Section II-E).

A. Characteristics of Cross-Device FL

A cross-device FL system generally involves one logical central parameter server (i.e., FLS) and a lot of dynamically available clients (i.e., EDs); the iterative training is conducted in rounds and each round is made of three phases namely *selection*, *configuration*, and *reporting* as Figure 1 sketches.

Following the generalized FL framework of FedOpt [7], to drive a round of training, the FL server first selects a set of end devices as the participant training clients from those checked in recently (i.e., selection); then, these selected end devices download the new model from the FL server to start the training (i.e., configuration); once completing their local training, end devices upload their local gradients to the FL server (i.e., reporting); based on the collected results, the FL server will obtain the updated global model via aggregations, then move to the next round of training. Due to the possible heterogeneity in the training and the mobility of users, some end devices might report their results later than others, or even worse, fail to do so [15]. Accordingly, the FL server is generally configured to wait a pre-defined period for the collected results and ignore those missed the deadline [15]. Depending on the settings [5], the FL server might move to the next round of training once it has received the results of exactly a pre-defined amount of end devices (e.g., k, referred to as top-k driven FL, hereafter), or a maximum pre-defined timeout has been reached and the collected results have already met the minimum amount requirement of end devices (referred to as deadline-driven FL, hereafter).

In practice, the convergence behaviors of DML jobs like FL are jointly controlled by various factors, e.g., the characteristics of the training data, the capacities of the hardware, and settings of hyper-parameters like the batch size, training algorithms, learning rate, model aggregation frequency, etc [16], [17], [18], [19]. As reported by related works, the FL server would select end devices for each round of training respecting various design aspects [18], [19], [20]; and under appropriate hyper-parameter settings, the increment in the number of contributor end devices per round could make the trained model converge to the targeted actuary with fewer rounds, i.e., yielding a faster convergence speed [18]. Thus, given a group of end devices selected by the FL server for a round of training, it is crucial to make them contribute to that round within the deadline as much as possible.

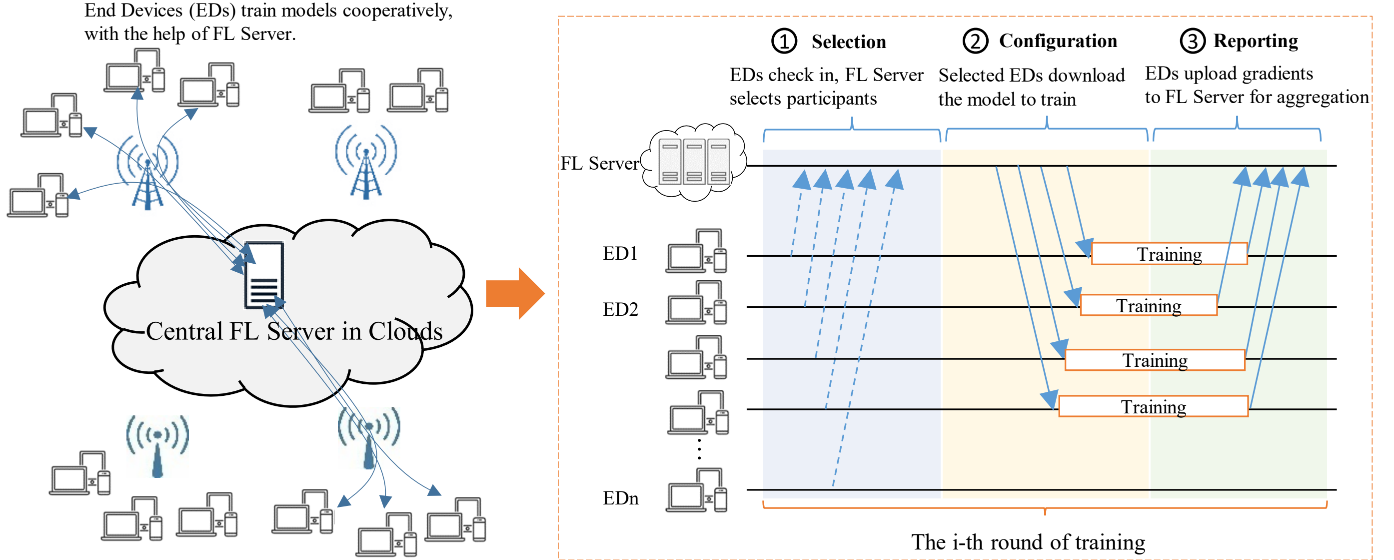


Fig. 1: The workflow and communication patterns of cross-device Federated Learning (FL) [6].

B. Importance of Communication Optimization

Obviously, in cross-device FL, with the number of participants scaling up, the central FL server would become the bottleneck of the entire training. With the emerging and widespread employment of edge clouds, one promising optimization for cross-device FL is to employ nodes at the edge to cache the downloaded model for the elimination of duplicated fetching, and pre-aggregate multiple correlative gradient upload requests, from a group of nearby end devices within a short interval. As a result, both the traffic and computation loads on the FL server can be greatly reduced.

Indeed, such an optimization design could be treated as a specific in-network processing service customized for cross-device FL [10], yielding two levels of advantages:

- **Making FL systems highly scalable**. With edge-based in-network processing, not only the traffic load on the FL server, for both the model downloads and gradient uploads would be greatly reduced, but also parts of the aggregation computation originally conducted by the FL server, are offloaded and distributed to edge nodes. This makes it easy for FL systems to adopt a large number of devices in each round of training.
- **Accelerating the training significantly**. With the cache and aggregation services provided by edge nodes, end devices would take less time to download the model and upload their local gradients, improving the training efficiency. For example, in top-k driven FL, the delivery of data generally takes a non-trivial proportion or even dominates the entire time of a round of training; thus, reducing the communication time would accelerate the training iteration, as the FL server can collect the desired number of results more quickly [5]. In deadline-driven FL, communication optimization also brings benefits. In practice, to avoid negative impacts on the user experience, an end device only trains models when idle and aborts the training once the condition is no longer met [6]. Thus, the availability of an end device is perishable and the time slot might be short. Failing to complete the local training and report the results back to the FL server within the deadline brings no benefits to the global model [15]. Accordingly, reducing the communication time would not only let more end devices intentionally selected by the FL server [20] contribute to that round of training within the given deadline, but also enable the usage of shorter deadlines, thus accelerating the entire training [5], [18].

C. Why In-Network Processing Instead of Hierarchical FLS

Compared with performing in-network processing at the edge, a very related alternative design is to deploy regional parameter servers at the edge, around the base station. These regional servers together with the central FL server aggregate devices' gradients hierarchically (i.e., Hierarchical FLS) [3], [9]. We argue that in-network processing is more attractive in three aspects as Table I summarizes.

More specifically, in terms of applicability, the scheme of hierarchical FL servers is a FLS-specific solution. Regarding implementation, FL applications owned by different companies and organizations are generally built upon their own specific versions of FL systems. To support them all, the design of hierarchical FL servers has to deploy FL servers at the edge for each FL system separately and explicitly. In practice, these edge FL servers generally run inside virtual machines (VMs) or Linux containers with pre-configured resources; accordingly, edge cloud resources are allocated very coarsely, at the granularity of VMs or containers. Moreover, using hierarchical FL servers, all the deployed edge FL servers and the central FL server form a distributed system, thus complicated gradient synchronization protocols are needed [3].

In contrast, as we will show through this article, in-network processing can be implemented as an optional and reusable edge service that provides best-effort in-network model caching and gradient aggregating for cross-device FL

TABLE I: In-Network Processing vs. Hierarchical FLS.

Solution	Applicability	Resource allocation	Manageability
Hierarchical FLS	FLS-specific, requiring case-by-case designs and deployments	Coarse-grained, per-VM or per-container	Complicated
In-Network Processing	FLS-agnostic, generic and future-proof	Fine-grained, per-packet	Easy

TABLE II: Limitations of related domain-specific transport protocols and schemes designed for in-network processing. Here, ✔: supported, 🔍: partially supported, ✖: not considered, **A-R**: asynchronous reduction, **A-B**: asynchronous broadcast, **P-S**: progress synchronization, **D-D**: deadline-driven INA, **JF-RA**: job-scale-aware fair resource allocation.

Proposal	A-R	A-B	P-S	D-D	JF-RA
SwitchML [21]	✔	✖	✖	✖	✖
ATP [22]	✔	✖	✖	✖	✖
A2TP [23]	✔	✖	✖	✖	✖
PA-ATP [24]	✔	✖	🔍	✖	✖
Canary [25]	✔	✖	✖	✔	✖
ASK [26]	✔	✖	✖	✖	✖
NetReduce [27]	✔	✖	✖	✖	✖
MTP [8]	🔍	🔍	✖	✖	✖
NetRPC [28]	✔	🔍	✖	✔	✖
Desired proposal	✔	✔	✔	✔	✔

tasks. Indeed, in-network processing is a FLS-agnostic solution: in practice, the most common model aggregation strategy is to compute a weighted average of the gradients [3]; by implementing this function as a network service, in-network processing based solution is generic and able to support present and future FL algorithms. Also, there is no need to deploy and run separate software instances for different FL systems. For each FL training job, the in-network processing instance can determine the cache or aggregation operation of each packet solely, resulting in packet-level resource allocation. Moreover, as an optional network service, the management of in-network processing is simple and decoupled from those of the supported FL systems.

D. Desirable Properties and Drawbacks of Existing Solutions

This paper is not the first to accelerate distributed model training with in-network processing. Recently, researchers have applied a similar idea of INA for intra-datacenter DML (DC-DML), e.g., by configuring powerful Top-of-Rack (ToR) switches as aggregators [21], [22], [29], [30]. However, since the workflow, underlying network, and available intermediate processing nodes of cross-device FL are quite distinct from those of DC-DML [21], [22], [29], [31], [32], the solutions they prefer differ significantly in many design aspects. Particularly, as highlighted by the recent work of [8], it is vital to design new transport protocols to provide in-network processing services for distributed applications [8]. Currently, a series of domain-specific transport protocols and schemes have been proposed to support in-network processing (e.g., INA) for intra-datacenter distributed applications like DC-DML. However, as summarized in Table II, they fail to fully meet the requirements of cross-device FL.

More specifically, in the context of cross-device FL, due to various factors like the dynamic availability and heterogeneity of training capacities, selected end devices might not complete their local training at the same time; and similarly, they are likely to start the download of training tasks asynchronously. Thus, the desired proposal should support both asynchronous reduction (A-R) and asynchronous broadcast (A-B). Moreover, if end devices differ significantly in their progress of model download or gradient upload, progress synchronization (P-S) is vital to make efficient usage of both the limited cache memory of edge boxes and the bandwidth of its path from and to the FL server for in-network acceleration. As summarized in Table II, despite existing INA solutions like SwitchML [21], ATP [22], A2TP [23], PA-ATP [24], Canary [25], ASK [26], and NetReduce [27] all supporting A-R, they either do not consider the dissemination of model parameters or just employ using IP multicast for the delivery (i.e., A-B is not supported). Among them, only PA-ATP partially supports P-S; however, PA-ATP is far from optimal, since it conducts synchronization by indirectly controlling the allocation of bandwidth, rather than relocating the process—Such a design also does not guarantee work-conserving resource allocation [33], [34], i.e., there are still available link capacities that could be allocated to active transfers. Moreover, as explained in Section II-A, some end devices in FL might fail to report their results within the given deadline; thus, the proposed scheme should support deadline-driven in-network aggregation (D-D), accordingly; however, only Canary [25] supports this feature. Furthermore, as we will explain in Sections IV and V, when multiple cross-device FL jobs coexist in the system, the shared in-network processing resources like the cache memories and link capacities should be allocated fairly with the awareness of the scale of each job, e.g., in terms of the number of involved participating end devices, referred to as job-scale-aware fair resource allocation (JF-RA). Unfortunately, non-existing schemes have provided this.

Different from the above INA solutions, MTP [8] aims at providing a generic protocol to support various types of in-network processing; however, a lot of design details still need to be built; and according to its preliminary designs [8], features like P-S, D-D, JF-RA are not supported. NetRPC [28] tries to provide an RPC framework to simplify the use of in-network processing, but it still does not support P-S and JF-RA, and only partially supports A-B.

The idea of letting boxes inside the network cache the transmitted data is also employed by the new networking architecture of NDN [14]. However, INP is fundamentally different from NDN in at least two aspects. Firstly, INP is designed to work as an optional network service built upon

edge computing and UDP, thus readily deployable in today's Internet relying on IP. By contrast, NDN employs a clean-slate design incompatible with IP-based network architecture and is thus hard to deploy. Secondly, in INP, packets are routed based on their destination IP address since INP does not touch the routing intelligence. Such a design makes INP easy to deploy since only involved endpoints and edge boxes should be upgraded to support INP. Instead, in NDN, packets (messages) are routed based on an identification of the data, requiring an upgrade of the whole network. These differences make INP distinguished from NDN, being readily deployable in today's IP-based Internet.

E. Design Challenges

To realize the vision of implementing in-network cache and aggregation at the edge as a best-effort and optional communication acceleration service for cross-device FL, three primary design challenges must be addressed.

- First of all, in-network processing has broken the end-to-end and one-to-one communication principles; as a result, the supports of transport protocols are needed to release the benefits of in-network processing to applications, while providing features like A-R, A-B, and D-D; however, existing protocols fail short to do so [8], [35].
- Secondly, to accelerate the model downloads and gradient uploads for FL with limited in-network processing resources, the transmission progress of different devices should be similar—This is because only the related data chunks can share caches and/or be aggregated during the delivery. However, due to the system's heterogeneity, end devices generally have slightly skewed transmission progress; thus, novel schemes are needed to support P-S.
- Last but not least, in practice, there might be multiple FL jobs in the system [36], to make efficient and fair usage of the available in-network processing and network resources, novel resource allocation algorithms are needed to conduct fair and work-conserving congestion control and cache management, i.e., providing JF-RA.

In the following, we will first overview the design of INP in Section III, then describe the details of its involved data channel transport protocols MDP and MUP in Sections IV and V, respectively. Basically, the novel workflows of MDP and MUP enable INP to overcome the first primary challenge. With two schemes of progress synchronization and flow control (if involved), elements in INP can cooperate to increase the possibility of in-network cache and aggregation, thus addressing the second challenge. Finally, using a suite of congestion control and cache management algorithms, MDP and MUP are also able to make work-conserving (efficient) and fair usage of the available in-network processing and network resources, thus addressing the third challenge.

III. INP Framework

Distinguished from proposals that directly control the participating workers, their training workloads, or the hyper-parameter settings such as the learning rate of training devices [37], [38], INP targets accelerating the communication efficiency of model download and gradient upload in a best-effort manner, without touching other training settings. As analyzed in Section II, such a design is generic and powerful to improve the efficiency of both top-k driven FL and deadline-driven FL by making the training iterate faster. Now, we overview its framework and key designs.

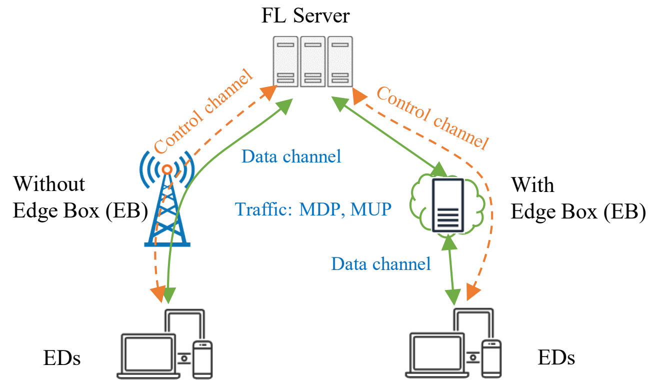


Fig. 2: The framework of INP.

A. Framework Overview

As Figure 2 shows, a typical INP-enhanced cross-device FL system involves three types of elements, namely *FL Server (FLS)*, *End Device (ED)*, and *Edge Box (EB)*, respectively, in which the EB acts as an optional cache and aggregation box residing between the other two. To start a round of training, the FL server first selects a group of end devices to pull the new model, using the cache service provided by edge boxes. When completing the training, end devices push their local gradients to the FL server, through the optional aggregation services provided by edge boxes. Once sufficient gradients are obtained, the FL server generates the final global aggregated model and moves to the next round of training.

Inside INP, participant elements set up two types of channels, i.e., *control channel* and *data channel*, for the involved data transmissions, respecting whether their transfers can be optimized/accelerated by the edge boxes of INP, or not. The *control channel* is employed for the exchange of signal messages like the *join*, *leave*, and *select* of end devices between the FL server and end devices; such a feature is already supported by modern commercial FL systems, and existing point-to-point transport protocols like TCP and QUIC can be used [2], [6]. While the *data channel* is used for the delivery of model parameters and gradients, with the assistance of edge boxes.

B. Domain-Specific Transport Protocols

As pointed out by several concurrent recent works [8], [35], new transport protocols are pivotal to take advantage of in-network communication acceleration for distributed applications like cross-device FL; because existing protocols TCP, QUIC, and UDP fail to do so. Specifically, the introduced edge boxes residing in the path, break the end-to-end communication principles between endpoints; and multiple end devices talk to the same FL server with the assistance of the edge box asynchronously in practice, yielding the paradigm of

asynchronous many-to-one and one-to-many communication. Unfortunately, neither the TCP/QUIC nor the raw UDP is designed to deal with such scenarios, especially to efficiently use the edge box's cache and computation capacities. Notably, despite that UDP has been used to encapsulate data for IP-based multicasting, the multicast operation is carried out at the network layer by the involved routers and switches synchronously, which is different from that of INP.

To get the best advantage from edge boxes for *data channels*, INP employs two novel transport protocols, namely MDP (Model Download Protocol) and MUP (Model Upload Protocol), to achieve efficient and reliable model downloads and gradient uploads, respectively. By making efficient use of the edge box to cache the model parameters delivered by the FL server, and to pre-aggregate gradient chunks sent by end devices, respectively, MDP and MUP could reduce both the traffic volume inside the network and the workload of the FL server. As a result, the communication is accelerated. The model parameters and gradient values in INP are split into chunks, each of which, along with the MDP or MUP header, is encapsulated in a single UDP packet. This design has two types of advantages. On the one hand, it enables INP traffic safe to go through today's wide-area networks made up of a lot of middleboxes; and on the other hand, it makes the proposed MDP and MUP easy to deploy at end devices like smartphones, since they can be implemented at the application layer without modifying the network stack. Moreover, as Sections IV and V will show, both MDP and MUP do not involve time-consuming complex computations; thus, just like existing transport protocols, software implantation can make them work efficiently in practice.

C. Edge Box Designs

The edge box in INP is designed to act as a transparent proxy between end devices and the central FL server. For FL systems involving a huge amount of end devices across multiple regions, a group of edge boxes could be deployed and configured to work hierarchically, such that the load of the central FL server could be further reduced. Note that, in-network processing can be implemented as a network service provided by the Internet service or cloud providers using the pay-as-you-go price model like cloud computing and NFV. Accordingly, the number of edge boxes in the network employed by cross-device FL jobs is jointly determined by two factors: *i*) whether there are in-network processing services already deployed at the edge, and *ii*) whether the job is allowed (or willing) to employ these services.

As Sections IV and V will explain in detail, the functions required by edge boxes to support both MDP and MUP are not complex and thus easy to implement as software. Accordingly, edge boxes in production can be built upon modern NFV systems, thus naturally supporting scale-out and scale-in, respecting the workloads [10], [11]. Fundamentally, similar to solutions based on hierarchical FLS, the performance gains obtained by in-network processing stem from the design of offloading parts of the FL server's job to the edge box. This reduces traffic volumes inside the network and relieves the

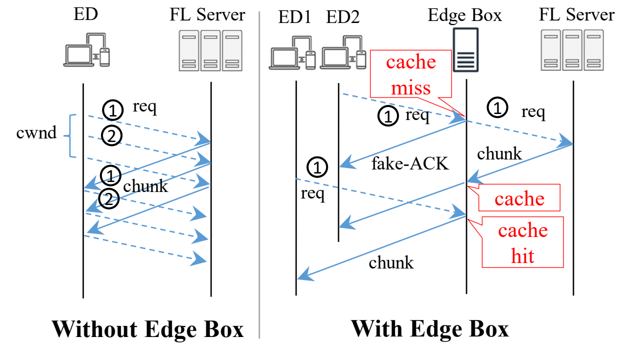


Fig. 3: The workflow of MDP with and without EB.

loads of the FL server. Thus, there are trade-offs between the edge box's loads and that of the FL server. If the edge box has limited resources, the operations can selectively enable in-network processing only for a subset of jobs.

IV. Model Download Protocol

In this section, we first overview the design of MDP (Section IV-A), then describe how the involved end devices synchronize their download progress to maximize the benefits of in-network cache (Section IV-B), which is the key for performance optimization when the edge box only has a limited size of cache and end devices start their downloads asynchronously. Finally, we present the novel congestion control designs of MDP enabling end devices to fairly use bottleneck bandwidth (Section IV-C) and the edge box's caches (Section IV-D).

A. MDP *Overview*

As Figure 3 shows, MDP is a "request-reply" based protocol: to download the model, the end device first sends a request (or *req* for short) to the FL server; on getting a *req*, the FL server replies with the desired chunk(s) made of model parameter values. Like the reliable design of TCP, in case a sent request is considered lost, the end device would resend. Compared with TCP, besides relying on UDP rather than raw IP, MDP mainly has four differences, making it outstanding for downloading chunked model parameter values.

- Firstly, as signal and management messages in INP are delivered via the separated *control channels*, MDP does not require handshakes to establish or close *data channel* connections for chunks. Accordingly, for the download task to a given end device, MDP only needs a one-way, rather than bidirectional, connection.
- Secondly, an end device completes its model download only when all the involved values are fetched, thus MDP does not require severe in-order delivery; as Section IV-B will show, such a characteristic enables end devices that are downloading the same model parameters to synchronize their progress to improve the rate of cache hits, even if they do not start their downloads simultaneously.
- Thirdly, MDP is designed to leverage edge boxes on the way as cache nodes, so that multiple download requests of the same chunk from nearby end devices can be accelerated by their shared edge box(es). However, the hit-and-miss of the cache makes the allocation of bandwidth on

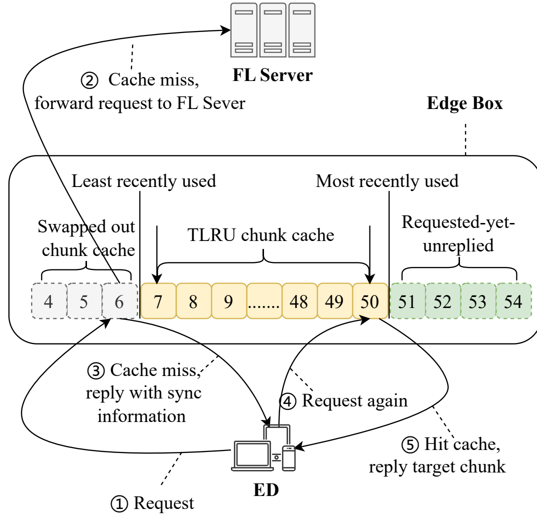


Fig. 4: An example showcases the workflow of how the progress synchronization of MDP is triggered by a cache miss.

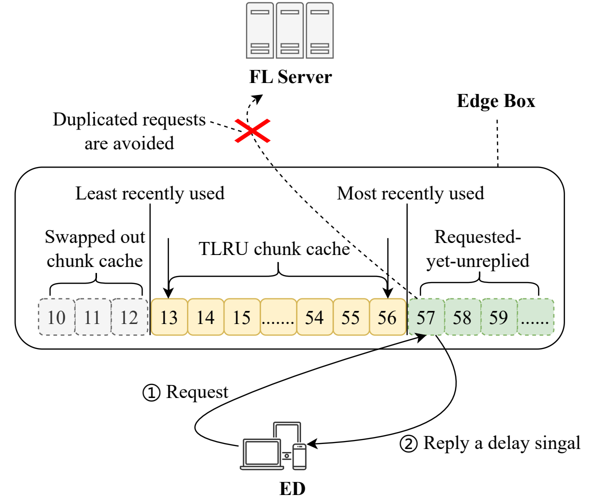


Fig. 5: An example showcases the workflow of how the edge box eliminates duplicated requests for MDP.

bottleneck links sophisticated. To deal with these issues, as Section IV-C will show, MDP leverages a suite of novel congestion control designs.

- Last but not least, as the workflow of EB-assisted MDP at the right of Figure 3 shows, on receiving a request, the edge box checks whether it holds the desired chunk; if so (i.e., a cache hit), it replies directly, otherwise (i.e., a cache miss), forwards the request to the FL server, immediately replies a *fake-ACK* for congestion control, and caches the corresponding reply when it goes by. Essentially, the edge box works as a transparent cache proxy for MDP; thus, it is the duty of MDP to deal with the loss of packets. In practice, to achieve high performance, the cache needed by MDP is generally implemented in memories with limited volume sizes. For the management of the cache, the well-known algorithm of *Time-aware Least Recently Used (TLRU)* is a solution but job-aware inter-job cache management is still needed as Section IV-D will show.

B. Progress Synchronization

1) Relocation Designs: In case edge boxes have limited sizes of cache, only the recently active chunks would be cached. To improve the rate of cache hits, MDP would synchronize the download progress of related end devices to maximize the benefits of in-network caching. Obviously, if there is only one end device fetching the model, progress synchronization is not needed. Instead, when multiple end devices are fetching the same model, the end device with the largest fetching throughput would continue to create cache chunks at the edge box when its reply message passed by; then other end devices would take advantage of these caches. Hereafter, we refer to the end device triggering the cache of model chunks on the edge box as the *cache-creator*, and other end devices that hit these caches for fetch acceleration as the *cache-users*. Accordingly, the synchronization of model download progress is necessary, only when there is already an active *cache-creator* but a cache miss is still encountered at the edge box for a *cache-user*. Given that MDP works in a "request-reply" manner, the synchronization of progress can be easy to implement using the following principles:

- Each end device computes and piggybacks its recent throughput on requests; based on the messages, the edge box selects and records the one with the highest throughput to act as *cache-creator* for following fetch requests.
- For each cache miss event, the edge box would forward the request to the FL server by default (e.g., the case shown in the right-hand of Figure 3); in case the source sender is not the current active *cache-creator*, a short reply specifying the suggested sequence number (saying $\hat{k}$) for the next chunk request would be sent back to the corresponding end device, to trigger progress synchronization (see the example shown in Figure 4).
- On getting the relocation reply of $\hat{k}$, the end device adjusts its next request sequence to $\min\{k : k \in U^{\text{MDP}} \wedge k \geq \hat{k}\}$ for synchronization, where U^{MDP} denotes the set of sequence numbers of chunks that this end device has not held yet.

Here, $\hat{k}$ is set to the sequence number of the newest chunk that has just been added to the TLRU cache of the edge box. Using the above designs, if there is only one active end device, it would always be recognized as the *cache-creator*; otherwise, a newly active (except the first) end device would be recognized as a *cache-user* by default. Then, during the transmission, if the download process of a *cache-user* has overtaken that of the current *cache-creator*, it would become the new *cache-creator*.

2) Request Deduplication: In case the *cache-user*s catch up with the *cache-creator*, or all end devices have similar download throughput leading to similar progress, *cache-user*s are likely to fetch a chunk that is still in flight and has not been cached at the edge box yet. As a result, their cache misses would lead to multiple replicated requests and replies between the FL server and edge box, lowering the power of in-network

caching and resulting in a waste of resources. To address this issue, MDP employs the following designs.

- Once a fetch request sent by the *cache-creator* has been forwarded to the FL server, the edge box records the involved chunk sequence number and maintains an expiration timer, with the initial value of τ_e.
- If the fetch request of the same chunk has triggered a cache miss at the edge box, instead of forwarding the request to the FL server and replying with a relocation message to the end device, the edge box would only reply with a "delay" signal, without forwarding (see the example shown in Figure 5).
- On getting a "delay" signal, the end device would pause the request of chunks for τ_d seconds; during this interval, any other "delay" signals would be ignored.

By default, τ_e is set to 2× of the measured RTT from the edge box to the FL server, and τ_d is set to the measured RTT from the end device to the edge box. With such a "delay" design, the amount of duplicated requests from the edge box to the FL server can be greatly reduced.

C. Congestion Control

To make efficient use of the available bandwidth, each end device maintains an *Additive Increase Multiplicative Decrease (AIMD)* congestion window (i.e., *cwnd*) like that of TCP [39] to control the number of in-flight requests. Regarding packet loss, like the design of the PPush protocol [33], besides timeout events, end devices would treat an MDP request or reply as lost, on receiving the replies of AHEADREPLYNUM (e.g., 3) requests sent after it. Note that, due the the assistance of edge box, if a request happens to miss the cache but its follow-up three requests hit the cache; then, in the view of the end device, the responses generated by the cache hit would reach the sender faster than the response of the first request missing the cache, leading to *false-loss detection*. We argue that the root cause of this issue is as follows: when an end device receives inconsecutive responses, it has no idea of whether this is caused by the miss of cache, or the loss of packets. To fix this issue, as Figure 3 shows, for each cache miss event, the edge box would immediately reply a *fake-ACK* to the MDP sender, with which, end devices can avoid the above mentioned *false-loss detection*.

In the following, we describe the details of how MDP conducts job-aware fair bandwidth allocation.

1) Problem Descriptions: Consider that there are n concurrent FL training jobs whose traffic goes through the same edge box and the same FL server, and this shared path happens to be the bottleneck. We assume that there are m_i end devices for the i-th job, leading to the total amount of $\sum_{i=1}^{n} m_i$ MDP flows in the systems. If there is no in-network cache, the j-th job would obtain about $\frac{m_j}{\sum_{i=1}^{n} m_i}$ of the bottleneck bandwidth under AIMD-based congestion control. However, due to the assistance of in-network caching, for each training job, generally only the *cache-creator*'s MDP requests and replies would go through the links between the edge server and the FL server. As a result, each job would obtain about $\frac{1}{n}$ of the bottleneck bandwidth, yielding unfair bandwidth allocations at the level of FL jobs.

2) Solutions: Assume that the bandwidth of the path from the FL server to the edge box is B^{MDP}. As a remedy, we can limit the total bandwidth occupied by end devices belonging to the j-th FL job not exceeding B_j^{MDP} calculated via E.q. (1). To accelerate the job that has only one uncompleted end device, we can treat it as having $\max_{i=1}^{n} m_i$ end devices.

$$B_j^{\text{MDP}} = \frac{m_j}{\sum_{i=1}^{n} m_i} B^{\text{MDP}} \tag{1}$$

Regarding implementation, the edge box can compute the suggested B_j^{MDP} for each FL job j and piggyback this information on the response message to the current *cache-creator* end device. Upon receiving this information, the end device would calculate the upper bound for the allowed congestion window size, via E.q. (2). Here, s_j denotes the chunk size used by this FL job, and RTT_j^{MDP} denotes the current averaged round-trip time measured by the current *cache-creator* end device.

$$\bar{W}_j^{\text{MDP}} = \left\lceil \frac{B_j^{\text{MDP}} RTT_j^{\text{MDP}}}{s_j^{\text{MDP}}} \right\rceil \tag{2}$$

However, a naive implementation of the above design might not guarantee work-conserving bandwidth allocations. In practice, there are multiple bottleneck links in the systems. Due to the limited link capacities between the edge box and the end devices, a FL job (saying j for instance) might not be able to use up $\frac{m_j}{\sum_{i=1}^{n} m_i}$ of the total capacity from the FL server to the edge box. However, other FL jobs are unable to use the remaining link capacities either, leading to a waste of bandwidth. To address this issue, motivated by recent studies [32], we employ an overbooking design for the estimation of B_j, and dynamically adjust the overbooking rate respecting the actual observed link utilization of the path from the FL server to the edge box. Assume that the actual capacities and current throughput of the path are B_*^{MDP} and R^{MDP}, respectively. Then, we use the E.q. (3) to compute a B^{MDP} for the calculation of B_j^{MDP}s used in E.q. (1).

$$B^{\text{MDP}} = \alpha^{\text{MDP}} B_*^{\text{MDP}} \tag{3}$$

Here, α^{MDP} is the rate of overbooking, which is dynamically adapted using E.q. (4), in turn. Note that, for MDP, as the reply message is generally much larger than the request, thus both B_*^{MDP} and R^{MDP} are dominated by the available bandwidth of the directed path from the FL server to the edge box.

$$\alpha^{\text{MDP}} = \begin{cases} 1, & \text{if } \frac{R^{\text{MDP}}}{B_*^{\text{MDP}}} \geq \lambda_1 \\ \max(1, \alpha^{\text{MDP}} - \delta), & \text{if } \lambda_1 > \frac{R^{\text{MDP}}}{B_*^{\text{MDP}}} \geq \lambda_2 \\ \alpha^{\text{MDP}} + \delta, & \text{otherwise} \end{cases} \tag{4}$$

Initially, $\alpha^{\text{MDP}} = 1$. During the delivery, the edge box would update both R^{MDP} and B^{MDP} at the interval of its RTT to the FL server, to update the value of the overbooking rate α. If the observed average bandwidth utilization (i.e., $\frac{R^{\text{MDP}}}{B_*^{\text{MDP}}}$) does not reach λ_2, α^{MDP} would be increased by about δ every RTT. If the utilization is larger than λ_2 but less than λ_1, it would decrease δ each time. And if it is larger than λ_1, overbooking would be temporarily disabled. In Section VII, we use $\lambda_1 = 0.95$, $\lambda_2 = 0.9$, and $\delta = 0.05$ as the default settings.

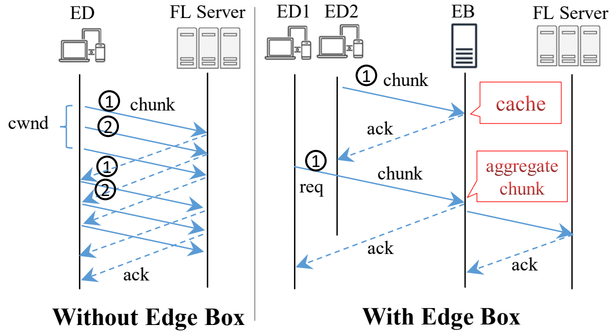


Fig. 6: The workflow of MUP with and without edge boxes.

In a nutshell, by limiting the cwnds of end devices using the above weighted and overbooked designs, MDP achieves job-scale-aware fair and work-conserving allocation of the bandwidth of the path from the edge box to the FL server.

D. Cache Management

Similar to bandwidth allocation, when the cache size of edge box for MDP is limited (saying V^{MDP} for instance) and there are multiple jobs, the FL job j should obtain v_j^{MDP} for per-job fair-sharing as E.q. (5) denotes. Likewise, strictly limiting the cache occupancy of FL job j not exceeding v_j^{MDP} would waste the cache if any other jobs have not used up their quotas.

$$v_j^{\text{MDP}} = \frac{m_j}{\sum_{i=1}^{n} m_i} V^{\text{MDP}} \tag{5}$$

To deal with the issue and to make efficient usage of all available caches, the edge box could count the actual cache size it already allocates to each job. If there are remaining caches available, it will try to use them for another FL job, even if the actual cache size this job occupies already reaches the quota. To implement the above design insights, for each job j, we define p_j^{MDP} via E.q. (6), to quantify the level at which it has occupied the remaining cache resources. Here, $\bar{v}_j^{\text{MDP}}$ denotes the cache size already occupied by job j. Then, in case the cache quota of job j runs out, instead of directly popping out the least recently used cached chunk belonging to job j, the edge box would select the job with the largest p_j^{MDP} value for TLRU cache replacement.

$$p_j^{\text{MDP}} = \frac{\bar{v}_j^{\text{MDP}}}{v_j^{\text{MDP}}} \tag{6}$$

Using the above quota-based lazy cache reallocation designs, the edge box allocates its MDP memories to concurrent jobs in a job-scale-aware fair and work-conserving manner.

V. Model Upload Protocol

To explain the design of MUP clearly, we first overview its designs in Section V-A, then describe how progress synchronization and flow control designs are employed to make efficient usage of the edge box's cache without overflowing it in Section V-B. After that, the involved congestion control schemes are presented in Section V-C, and finally, the management of cache follows in Section V-D.

A. MUP Overview

Similar to the design of MDP, MUP is a "request-reply" based protocol designed for the upload of gradients as Figure 6 shows. When local gradients are ready, end devices split gradients into chunks; then, each chunk, together with its index (for serialization) and weight (for weighted aggregation, with the initial value of 1), will be packed as the payload of a UDP datagram then sent to the FL server. On getting a gradient chunk, the FL server sends an acknowledgment packet (or *ack* for short) immediately. Also, by measuring the arrival of *ack*s, like the design of MDP, the end device adjusts its *cwnd*, learns the possible loss of data packets and resends.

As Figure 6 shows, when there exists an edge box in the path, it could work as an aggregator for MUP traffic. More specifically, on receiving a gradient chunk, besides generating the acknowledgment, this edge box would cache it, or update the cached weighted average value of the gradients. Here, the value of the carried weight represents the total number of end devices from which this gradient chunk is computed. Once the edge box has received most or all the possible gradients that would go through this edge box, or the time starting from the caching/creation of this gradient chunk exceeds a pre-defined timeout threshold, the edge box would act as a specified end device: it would send the aggregated gradient values, along with the updated weight value, to the final FL sever. To be flexible, MUP allows end devices to specify their expected timeout thresholds along with the reported results, which are configured by the FL server in turn. It should be noted that, thanks to the novel proposed Upper-stage Upload Filling (U2F) scheme (see Section V-D1 for details), as confirmed by the simulation in Section VII-C5, such a timeout design generally would not enlarge the completion of MUP flows. This is because if remaining bandwidth is available but all the cached data is not fully aggregated, the edge box would deliver these partially aggregated data to the FL server, to make efficient usage of all available bandwidth. By default, the weight value represents the number of end devices from which the aggregated gradient chunk is computed. In case an aggregated chunk does not get acknowledged, the edge box would conduct retransmissions just like an end device.

In general, the transmission of MUP traffic from end devices to the FL server is decoupled into two stages by the edge box, referred to as the lower stage and the upper stage, respectively. Unlike the process of MDP traffic, to achieve reliable transmission for MUP chunks, the edge box would maintain all these MUP chunks it has received from the end devices in the memory until these chunks or their aggregated results have been sent to and acknowledged by the FL server. As a result, MUP needs a flow control mechanism to avoid overflowing the edge box's memory. Besides, new progress synchronization and congestion control designs are also needed to make efficient usage of the available link capacities and the ability of in-network aggregation.

B. Progress Synchronization and Flow Control

The edge box maintains all received-yet-unsent and sent-yet-unacked chunks for reliable delivery. When remaining

memories are available, for a received chunk that could not be aggregated with any existing one, the edge box would store it in memory and reply to the sending end device with acknowledgment along with a desired chunk sequence number for progress synchronization. By default, the sequence number of the oldest chunk that this edge box has received yet unsent to the FL server would be used. On getting such a sequence number, saying $\bar{k}$ for instance, the end device would relocate its upload progress to $\min\{k : k \in U^{\text{MUP}} \wedge k \geq \bar{k}\}$, where U^{MUP} denotes set of chunks having not been successfully uploaded.

In case the edge box's memory for MUP runs out, for a newly received chunk that can not be aggregated into an existing one already residing in the memory, it would directly discard this chunk and reply to the source end device with a specific NACK message along with the desired relocation sequence number for progress synchronization. On getting such a reply, the end device would immediately relocate the upload progress accordingly if the chunk at the target location has not been sent. If the chunk at the target location has already been successfully uploaded to the edge box, or it is a sent-yet-unacked chunk, the end device would not relocate its upload progress; instead, it would stop sending and resume the uploading after a short interval, with the default value of the measured RTT from itself to the edge box.

C. Congestion Control

Given that the transmission of MUP traffic has been divided into two stages, they trigger two congestion control instances, accordingly. Similar to the case of MDP, in each stage, the sender maintains a congestion window (cwnd) with AIMD policy to limit the amount of sent-yet-unacked chunks, but with several slight differences.

1) Upper Stage: To achieve job-aware fair sharing of the upper path, the edge box would limit the sending window of the upper stage of job j so that it does not exceed $\bar{W}_j^{\text{MUP}}$, as E.q. (7). Here, B_*^{MUP} and RTT_j^{MUP} denote the maximum bandwidth and current RTT of the upper path measured by the edge box, respectively. s_j^{MUP} is the size of each chunk. As the chunk message sent in MUP is generally much larger than the received reply, B_*^{MUP} is dominated by the available bandwidth of the directed path from the edge box to the FL server. Similar to the case of MDP, α^{MUP} is an adaptive parameter maintained by the edge box for dynamic overbooking controls. For job j, m_j generally denotes the number of active end devices it involves. Due to the acceleration of the edge box, some jobs might have no active end devices, once their end devices have uploaded their data to the edge box. For these jobs, once completed, they could release the occupied cache and these cache can be used by other jobs. Thus, accelerating their completion is beneficial. To do so, if a job does not have active end devices any more, we treat it as having a virtual group of $\max_{i=1}^{n} m_i$ end devices. Following the above design, MUP achieves job-scale-aware fair and work-conserving allocation of the upper-stage path's bandwidth.

$$\bar{W}_j^{\text{MUP}} = \left\lceil \frac{\alpha^{\text{MUP}} B_*^{\text{MUP}} RTT_j^{\text{MUP}}}{s_j^{\text{MUP}}} \frac{1 + m_j}{n + \sum_{i=1}^{n} m_i} \right\rceil \tag{7}$$

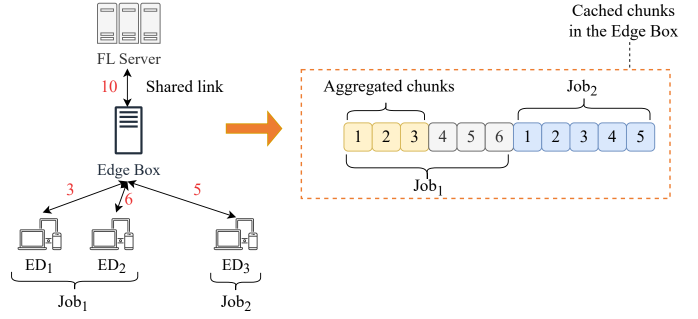


Fig. 7: To make efficient usage of all available bandwidth, for Job$_1$, after transmitting the aggregated chunks {1, 2, 3}, the edge box would continue sending the un-aggregated chunks {4,5,6}, when possible.

2) Lower Stage: Notably, for the congestion control of the lower stage, when a NACK message is received, since the loss of packet is not caused by congestion, the end device would reduce its congestion window size.

D. Cache Management

1) Intra-Job Cache Management: As the edge box has divided the transmission of MUP traffic into two stages, to make work-conserving usage of the upper path, cache-aware sending controls are needed. By default, the edge box would send a chunk to the final FL server if the time starting from the caching of this chunk exceeds a pre-defined timeout threshold, or if this chunk is exactly the aggregated result of the chunks sent by all end devices. As the example in Figure 7 shows, such a design might result in a waste of the bandwidth of the upper path. Consider that there are two upload jobs namely Job_1 and Job_2, which involve two end devices (ED_1 and ED_2) and one end device (ED_3), respectively. The paths from these end devices to the edge box have the available bandwidth of 3, 6, and 5, respectively, while the path from the edge box to the FL server has the bandwidth of 10. With the design of α-based overlooking, end devices belonging to these two jobs would obtain the actual throughput of 3 and 5, respectively. Obviously, such a result does not achieve work conservation, as there is remaining available for the end device ED_2 to upload its chunks to the FL server. The root cause relies on the edge box not sending these unaggregated chunks to the FL server, even though the remaining bandwidth is available.

To address this issue, when there are neither expired nor fully aggregated chunks to send, we enable the edge box to continue uploading these cached chunks (which might be partially aggregated), in the oldest first manner, to the FL server as well. Following this, the remaining bandwidth can be used properly. For these chunks that have been sent to the FL server without being fully aggregated, the corresponding "leftover" chunks sent by other end devices could never be fully aggregated at the edge box. To identify such "leftover" chunks and upload them to the FL server as soon as possible, for each chunk, the edge box also records the number of end devices whose reported data have been delivered. We refer to the above design as **Upper-stage Upload Filling (U2F)**.

2) Inter-Job Cache Allocations: Similar to congestion control, for job j, m_j generally denotes the number of active end devices it involves. If a job does not have active end devices any more, we treat it as having a virtual group of end devices, with a size equal to the maximum of the actual end device numbers involved by each job. Then, the j-th aggregation job would obtain the quota of v_j^{MUP} memories for aggregation.

$$v_j^{\text{MUP}} = \frac{1 + m_j}{n + \sum_{i=1}^{n} m_i} V^{\text{MUP}} \tag{8}$$

Given that the edge box could remove a received chunk or its aggregated value if and only if this chunk has been successfully delivered to the FL server. Thus, once the memory quota for a MUP job runs out and a newly received chunk uploaded by an end device can not be aggregated into these in the cache, the edge box would discard this chunk and respond to a specific NACK message as explained in Section V-B. Following this, the edge box could allocate its MUP cache to concurrent jobs for job-scale-aware fairness.

VI. Discussions

Now, we describe how INP could gracefully deal with hierarchical FL characteristics (Sections VI-A and VI-B), and discuss its newly introduced security issues (Section VI-C).

A. Data Heterogeneity

As described in Section III, the edge box in INP conducts coordinate-wise aggregation and cache operations on the uploaded and delivered data, respectively. Such a design makes INP generic and model-agnostic. Thus, the well-known problem of data heterogeneity faced by federated learning [40] would not impact the correctness and effectiveness of INP.

B. Training Straggler, Device Mobility and Failure

Due to the heterogeneity in cross-device FL, some end devices might report their trained results to the FL server later than others; even worse, they might fail to do so sometimes due to network errors and user interruptions [15]. To deal with these issues, the FL server generally supports two typical tolerance designs, which we refer to as *top-k driven FL* and *deadline-driven FL*, respectively, as summarized in Section II-A. Consider that the FL server has selected m active end devices to conduct a round of training. Then, in top-k driven FL, the FL server would generate the new aggregated model to launch the next round of training once it has received k end devices' results ($k \leq m$); while in deadline-driven FL, the FL server would wait for a (loosely) pre-defined reporting deadline to collect the results; then it conducts partial model aggregation to iterate to the next round of training if the collected number of end devices has met the minimum required amount [5], [15]. As described in Section V, INP edge boxes already support the above scenarios with a timeout-based design like Canary [25]—They conduct in-network aggregations in a best-effort manner and would send the partially aggregated results to the FL server under the driven of timeouts. Regarding MDP, as presented in Section IV, the dynamic change of the number of participating devices is trivial; no specific design is needed.

C. Security Issues

As shown in Figure 2, compared to the legacy cross-device FL system, INP mainly introduces the edge box, along with a suite of new domain-specific transport protocols built on top of UDP. This would introduce new threats beyond those studied in [41]. In fact, for threats also faced by legacy FL systems (e.g., poisoning attacks, gradient leakages [41]), similar designs can be applied in the context of INP. To keep this paper more focused, we briefly discuss these threat issues faced only by INP here and leave a full-fledged study of threat analysis and the design of solutions as future directions.

Basically, our design of INP is based on the assumption that edge boxes could provide trusted pre-aggregation and cache services for end devices and the FL server. If such a requirement is not satisfied, the acceleration service of INP should not be enabled. Then, the remaining threats are mainly from the network; thus, existing network security proposals can be used [42]. For example, by using existing message authentication schemes like HMAC [43], unauthorized packets could be filtered out and dropped by INP entities; by embodying messages with timestamps or serial numbers, replay attacks can be prevented; by letting INP entities encrypt the data before sending them out, the confidentiality of INP can be further enhanced; Indeed, for the acceleration of model delivery, the edge box can directly cache the ciphertext without decrypting, by not encrypting the chunk index. For the pre-aggregation of gradients, the edge boxes can decrypt first and then aggregate their original values. Alternatively, the well-known *Homomorphic Encryption (HE)* provides a promising solution since it supports the aggregation computation of *ciphertexts* without decrypting them in advance [44]. However, HE also greatly increases the amount of data that should be transmitted (two orders of magnitude for instance [44]), and advanced compress techniques are essential.

VII. Performance Evaluation

To verify the design of INP, we design an event-driven fine-grained simulator following the simulators used by PAM [32] and PPUSH [33], and conduct extensive tests to observe the detailed behaviors. Results confirm that the novel designs make INP able to accelerate both the dissemination of model parameters and the aggregation of gradient values, by making very efficient usage of the cache memories and computing ability of edge boxes. Indeed, for a FL job involving m end devices, INP equipped with MDP and MUP is able to reduce both the traffic load of FLS and the time needed for model downloads and gradient uploads up to m times.

A. Methodology

1) Workloads: As Figure 8 shows, we mainly consider the case in which several groups of end devices are jointly training models with the assistance of a shared edge box and FL server. For ease of description, the end device labeled by ED[i,j] is referred to as the j-th training worker of the i-th training job. By default, we assume that both the model parameters and gradients are split into chunks with the size of 1KB such that each of them can be encapsulated in a UDP; and if

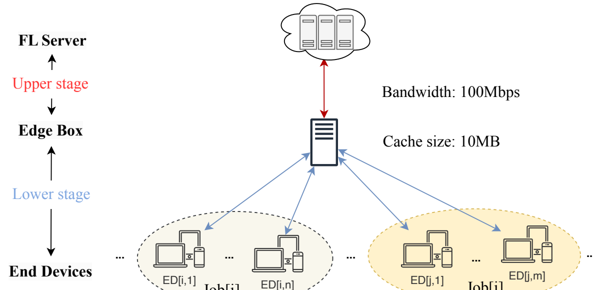
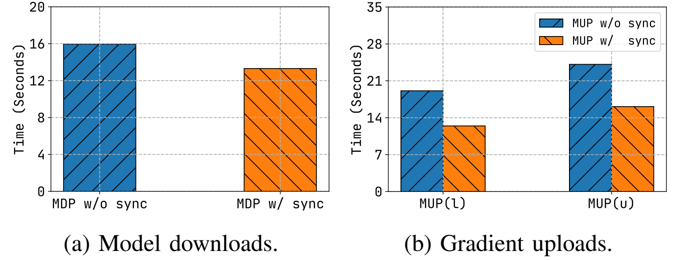


Fig. 8: The network environment used in tests.



(a) Model downloads. (b) Gradient uploads.

Fig. 9: With progress synchronization (i.e., w/ sync), the average completion times of model downloads and gradient uploads are significantly reduced, about 17% (MDP), 34% (MUP(u)) and 35% (MUP(l)), respectively, compared to the cases of no progress synchronization (i.e., w/o sync).

enabled, the edge box at the edge can perform in-network cache and aggregation for the passing model download and gradient upload requests, simultaneously. Respecting the test scenarios, there might be one or multiple FL jobs and the end devices might start their model download or gradient upload request synchronously or asynchronously.

To study and highlight the effects of INP on accelerating model downloads and gradient uploads, by default, we consider the case where 4 training jobs share the same edge box and FL server for either model download or gradient upload; each job involves 10 end devices; all links have a capacity of 100Mbps and latency of 10ms; the size of both the model parameters and gradients is 40MB; and the available cache that the edge box can use for MDP or MUP is 20MB. The start time of model downloads or gradient uploads for each end device follows a truncated exponential distribution: i.e., $t = \beta \min(X, 5)$, where X follows the exponential distribution of $Exp(1)$, and $\beta = 1$. In tests, we also change these settings to study their impacts.

2) Baselines, Metrics, and Tools: To the best of our knowledge, this paper is the first work that implements in-network processing as a best-effect acceleration service for cross-device FL and provides the support of transport protocols. Due to the significant differences with the existing schemes in the technological design, it is not comparable with other schemes like heterogeneous FLS. Thus, we mainly use the cases where in-network processing is disabled as the baselines. Indeed, this baseline can be treated as the cases of traditional TCP-based model downloads and gradient uploads.

As mentioned in Section II, the convergence behaviors of DML tasks like cross-device FL are jointly controlled by various factors, e.g., the characteristics of the training data, the capacities of the hardware, and settings of hyper-parameters like the batch size, training algorithms, learning rate, etc. [16], [17]; communication optimization is a generic and powerful design to improve the efficiency of both top-k driven FL and deadline-driven FL by making the training iterate faster. Note that, for deadline-driven FL, communication might also increase the number of end devices that contribute to each round of training. As recent studies have shown [18], [19], a larger number of participants for each round of FL generally benefits the convergence; however, this depends on various factors and thus the advantage is hard to analyze quantitatively. As a proposal aiming to provide generic yet best-effort communication acceleration services for cross-device FL, we argue that network-related metrics like the *achieved throughput* and *(average) communication completion time*, are more qualified than others to assess the performance of INP. Besides, we also investigate the detailed behaviors of the proposed protocols.

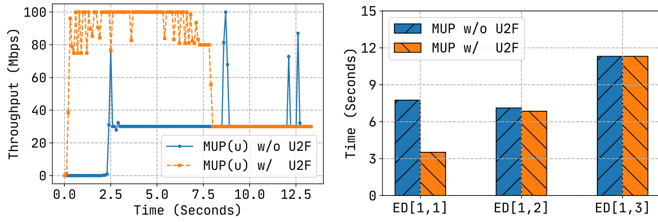
For each FL job, if multiple transfers are involved, its communication completion time is defined as the average completion time of all involved transfers. Note that, for the upload of gradients, MUP has decoupled its workflow into two stages (i.e., upper and lower), between which the edge box acts as endpoints. Accordingly, there might be gaps between the completion time of gradient uploads in the views of the end device(s) and the FL server. We label these two types of MUP traffic (flows) as "MUP(u)" and "MUP(l)", respectively.

To conduct the performance study, we implement a discrete event-driven simulator with Python 3 based on that of [32] and [33], which can precisely simulate the behavior of INP with and without the assistance of edge box, respectively. All tests are conducted on a 64-bit Ubuntu 22.04.3 server equipped with one Intel(R) Core(TM) i9-13900K CPU and four 32GB DDR5 memory cards. For each parameter setting, we perform 10 trials to compute and report their mean values.

B. Detailed Behaviors of MDP and MUP

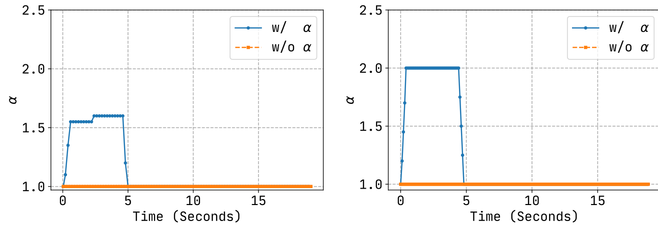
1) Progress Synchronization: Figure 9 shows the impacts of progress synchronization on the performance of MDP and MUP. Results confirm the remarkable benefits of progress synchronization for the cases where end devices do not start their download or upload requests at the same time. For example, with the help of progress synchronization, the average completion times of the model downloads and the gradient uploads could be reduced to about 83% (MDP), 66% (MUP(u)) and 65% (MUP(l)) of the cases of no progress synchronization.

2) U2F: To verify the benefits of U2F (see Section V-D1), we consider that there is only one FL job involving three end devices, namely, ED[1,1], ED[1,2], and ED[1,3]; they start to upload their gradient data to the edge box via links with the capacity of 100Mbps, 50Mbps, and 30Mbps, at the time of 0s, 1s, and 2s, respectively. As shown in Figures 10a and 10b, with U2F, the bandwidth of the shared upper-stage connection can be utilized more efficiently, leading to accelerated completion of gradient uploads for both ED[1,1] and ED[1,2].

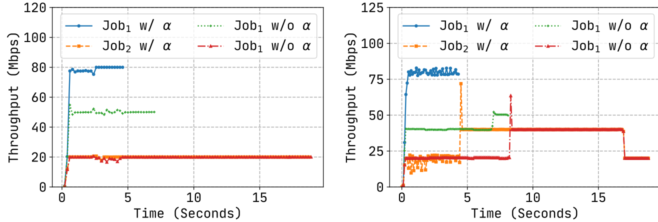


(a) The total aggregated throughput at the upper stage.
(b) The completion time of each end device's gradient upload task.

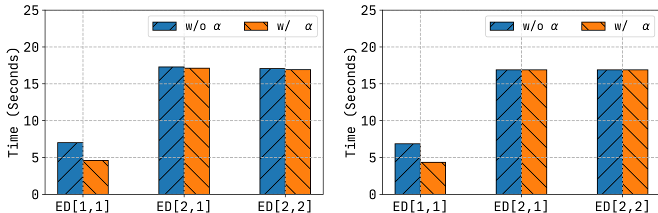
Fig. 10: With U2F, the upper-stage path has sufficient bandwidth, the overall data transmission progress of MUP will not be blocked by the end device with the lowest bandwidth.



(a) α of MDP in model downloads. (b) α of MUP in gradient uploads.



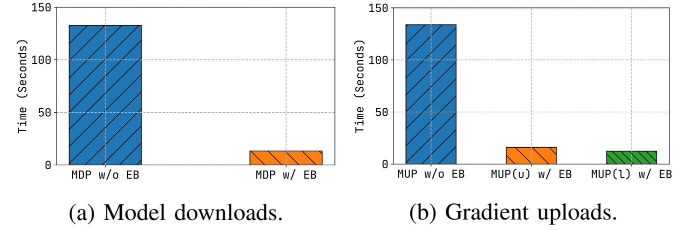
(c) Throughput at the upper stage in model downloads for each job. (d) Throughput at the upper stage in gradient uploads for each job.



(e) Model download completion times in the view of end devices. (f) Gradient upload completion times in the view of end devices.

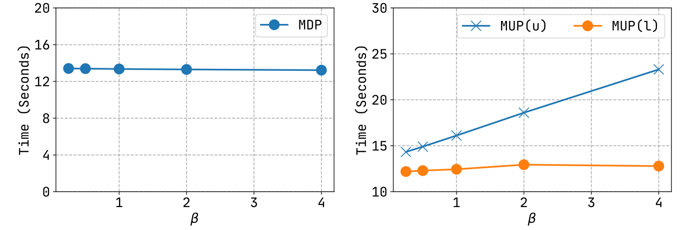
Fig. 11: With α-based job-aware adaptive overbooking, both MDP and MUP transfers could adjust their overbooking rates to make efficient and fair usage of the available bandwidth adaptively, resulting in possible accelerated job completions.

3) Congestion Control with Job-aware Adaptive Overbooking: To verify the effects of job-aware adaptive overbooking schemes proposed in Sections IV-C and V-C, we consider the case where three end devices, ED[1,1], ED[2,1], ED[2,2], belonging to two training jobs, Job_1 and Job_2, compete for the upper-stage path. ED[1,1] starts to download the model or upload the gradients at 0s via an 80Mbps lower-stage connection, while ED[2,1], ED[2,2], begin their model downloads or gradient uploads at 0s, 2s, via 20Mbps lower-stage connections, respectively. From 0s to 2s, both jobs involve only one end device. Without α-based adaptive overbooking, the throughput of Job_2 could only reach about 20Mbps, resulting in about 40Mbps remaining bandwidth at the upper stage, due to the bottleneck effects it suffers at the lower stage. Indeed, such bandwidth resources should be used by Job_1. As Figure 11 shows, with α-based adaptive overlooking, by tuning the value of α, the edge box could enable the flow that belonging to Job_1 to make work-conserving usage of the upper-stage path's capacities, resulting in accelerated completion of model downloads and gradient uploads for Job_1.



(a) Model downloads. (b) Gradient uploads.

Fig. 12: By making usage of the edge box, INP can accelerate model downloads and gradient uploads greatly.



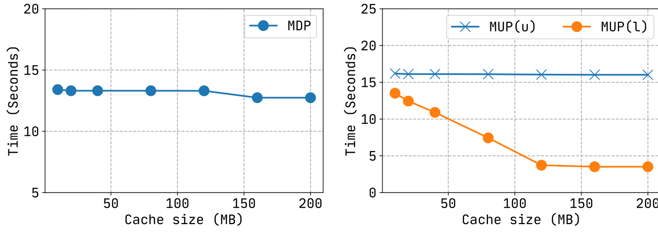
(a) Impacts of β on the completion of model downloads. (b) Impacts of β on the completions of gradient uploads.

Fig. 13: Both MDP and MUP are robust to achieve consistent performance, when β, the level of divergence of the start time of model downloads and gradient uploads, varies.

C. Acceleration Effects of MDP and MUP

1) Case Studies: Figure 12 shows the (average) completion times of both model downloads and gradient uploads under the default setting, with and without the assistance of the edge box, respectively. In the view of end devices, MDP and MUP could reduce the completion time for model downloads and gradient uploads by about 10× and 11×. And in the view of the FL server, the completion time of gradient upload is reduced by about 8×. Regarding the gaps between the completion time of gradient uploads observed by the end devices and the FL server, it mainly stems from the fact that end devices start their uploads asynchronously. This is because an end device treats its upload as finished once it has gotten ACKs from the edge box, while the FL server treats the job as done only when all the data sent by all end devices has been successfully received.

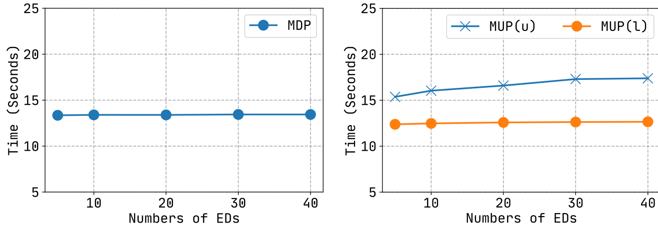
2) Impacts of β: Figure 13 shows the (average) completion times of model downloads and gradient uplaods when β increases from 0.25 to 0.5, to 1, to 2, and to 4. For MDP, the impacts is trivial because cache-creator will cover requests from cache-user, allowing the job to maintain similar throughput in the upper-stage path. For MUP, as β continues to



(a) Impacts of cache size on the completion of model downloads.

(b) Impacts of cache size on the completion of gradient uploads.

Fig. 14: Compared to MDP and MUP(u), the performance of MUP(l) is more sensitive to the available volume of cache.



(a) Impacts of job scale on the completion of model downloads.

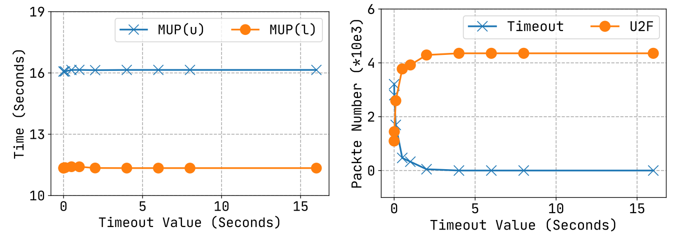
(b) Impacts of job scale on the completion of gradient uploads.

Fig. 15: For model download and gradient upload jobs involving m end devices, with novel in-network cache and aggregation designs, MDP and MUP could reduce the FL server's workload from $O(m)$ to $O(1)$, yielding consistent performance improvements in terms of the average job completion time.

increase, the completion time of the upper-stage path will also increase. This is because the larger β is, the greater dispersion the job start time has, leading to a decrease in aggregated and an increase in the total amount of data transmission.

3) Impacts of Cache Size: Figure 14 shows the impacts of the cache size on the performance of MDP and MUP when its value increases from 10MB to 20MB, to 40MB, to 80MB, to 120MB, to 160MB, and to 200MB. For MUP(l), with the size of the edge box's available cache continues to increase, the average completion time of gradient upload jobs decreases very fast, and woule keep consistent when the size reaches 120MB. Differently, MDP can only achieve maximum acceleration by storing all data in the cache. Otherwise, the edge box has to forward the request to the FL server, which would slow down the transmission. Such results also imply that when the edge box does not have enough cache to maintain all model parameters and gradients, only a limited cache volume (e.g., 10 out of 40 MB) could bring remarkable performance improvements for both gradient uploads and model downloads.

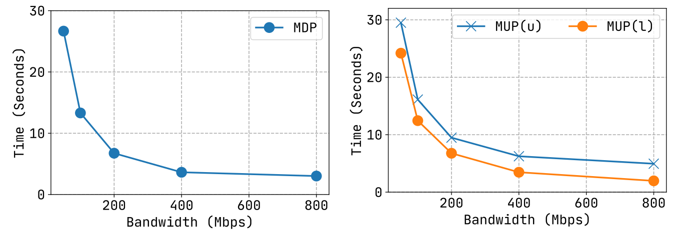
4) Impacts of Job Scale: As Figure 15 shows, both MDP and MUP could achieve consistent performance improvements in terms of the reduction of the completion time, with the increase in the number of involved end devices in each job. This is reasonable. With novel in-network cache and aggregation designs, for a model downloads and gradient uploads task involving m end devices, the edge box could nearly reduce the workload of the FL server from $O(m)$ to $O(1)$, almost eliminating its bottleneck effects. Consistent with the findings



(a) Impacts of timeout value on gradient uploads.

(b) Packet sending events caused by timeout and U2F.

Fig. 16: U2F enables the edge box to efficiently upload partial aggregated data to the FL server without suffering timeout.



(a) Impacts of bandwidth on model downloads.

(b) Impacts of bandwidth on gradient uploads.

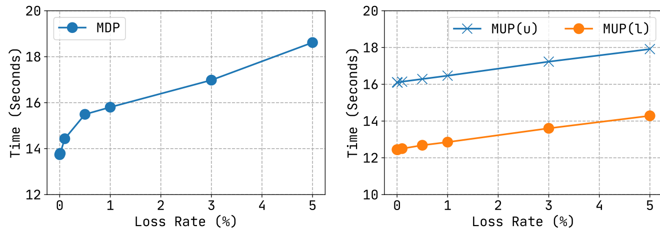
Fig. 17: With the increase in bandwidth, the average completion time of MDP and MUP transfers would decrease.

implied by Figure 13, due to the asynchronism of the start time of end devices belonging to the same job, for gradient uploads, especially in the FL server's view, a larger training scale generally leads to a longer completion time.

5) Impacts of Timeout Value: To study the impacts of the timeout value on the performance of MUP, we now consider that the second ready end device of Job_1 becomes unavailable after uploading gradient for 2s. As Figure 16a shows, while the timeout value increases from 0.001s to 0.01s, to 0.1s, to 0.5s, to 1s, to 2s, to 4s, to 6s, to 8s, and to 16s, this MUP job's completion time almost stays consistent. This is reasonable, as Figure 16b shows, with the growth of the timeout value, an increasing number of packet sending events are triggered by U2F (refer to Section V-D1 for details); once the timeout value is larger than 0.5s, U2F could send the vast majority of the partial aggregated chunks cached in the edge box to the FL server before timeout events occur.

6) Impacts of Bandwidth: Figure 17 shows the average completion times of model downloads and gradient uploads when the link capacity increases from 50Mbps to 800Mbps. As expected, a larger link capacity leads to smaller average completion times. As the bandwidth increases, the degree of the decrease also decays. This is because even when the available bandwidth is large, it would still take several RTTs for both MDP and MUP flows to increase their congestion window sizes to make efficient usage of the bandwidth and to complete their delivery tasks.

7) Impacts of Packet Loss Rate: Last but not least, Figure 18 shows the average completion times of model downloads and gradient uploads when the packet loss rate of links



(a) Impacts of packet loss rate on model downloads.

(b) Impacts of packet loss rate on gradient uploads.

Fig. 18: A higher loss rate leads to larger (i.e.,g slowed) average completion times for both MDP and MUP flows.

between involved entities increases from 0, to 0.0001, to 0.005, to 0.01, to 0.03, to 0.05, following the settings used in [45]. As expected, the increase in the packet loss rate would slow down the completion of both MDP and MUP transfers. Nevertheless, the average flow completion time does not grow very fast, almost linearly with the increase in the loss rate. Extending MDP and MUP to be more robust to random packet loss is interesting which is left as future work.

VIII. Related Work

As Sections II-C and II-D have compared INP with closely related proposals in detail, we now further shortly review other applications of in-network processing (Section VIII-A), other types of communication optimizations (Section VIII-B), and emerging applications of cross-device FL (Section VIII-C).

A. Other Applications of In-Network Processing

Beyond accelerating model synchronizations for distributed machine learning applications like cross-device FL, in-network processing has been widely employed for abundant distributed applications for performance enhancement. For example, proposals like [46], [47] explore the design of accelerating the reconstruction of chunks for EC-based distributed storage systems with in-network processing. NetFEC [48] employs in-network processing to conduct forward error correction (FEC) for reliable media data transmission. ASK [26] implements in-network aggregation of key-value streams as a generic service that distributed big data and high-performance computing applications can benefit from. As pointed out by MTP [8], the support from transport protocols is also important to make efficient and easy usage of in-network processing for these diverse distributed applications, which are open problems. To further simplify the usage of in-network processing, NetRPC [28] tries to integrate in-network processing based acceleration into the RPC framework thus making it easy to use for applications.

B. Related Communication Optimization Schemes for DML

Besides conducting in-network processing for aggregation and cache, there are also abundant other types of communication optimization design for cross-device FL or, even more generic, for DML. For example, several recent works show the possibility of relieving the slowdown effects of network congestion by conducting approximate gradient data transmissions through novel loss-tolerate transmission mechanisms [49], [50], [51], or lossy data compression techniques like quantization, sparsification, and low-rank decomposition [52], [53]. From another direction, proposals like Crew [17] and SelMCast [34] also explore the design of conducting model synchronization among partial rather than all workers to decrease the traffic volume triggered by each round of synchronization, and to deal with training stragglers; Differently, schemes like FedDrop [54] make use of dropout for random model pruning such that the communication and computation loads of each end device could be reduced. As orthogonal to these designs, extending our proposed INP to work with them jointly for communication optimization yields interesting further studies.

Different from INP, proposals like [37], [55] make edge servers act as local FL servers, and configure the frequencies of local aggregation and global aggregation properly; thus the communication loads would be optimized. As discussed in Section II-C, enabling edge servers to provide in-network aggregation and cache as best-effort network acceleration services is more generic and powerful.

C. Emerging Federated Learning Applications

Nowadays, FL has been employed for various application scenarios beyond well-known cases like item recommendations and content suggestions. For example, it can be used to manage the traffic at intersections for better idle durations and fuel consumption [56], control the routing of large-scale mesh networks for congestion avoidance [57], and optimize the various layers of the 6G wireless networks for better end-to-end quality of service and experience [58]. We argue that FL would have more advanced and wider applications and refer the readers to [59] for a more comprehensive survey.

IX. Conclusion and Future Work

In conclusion, we design INP, an In-Network Processing framework, along with the novel Model Download Protocol (MDP) and Model Upload Protocol (MUP), and a suite of resource allocation algorithms for progress synchronization, flow control, congestion control, and cache management, to accelerate the data deliveries involved in large-scale cross-device FL systems. The key of INP is to let *Edge Boxes* (EBs) residing between the selected training *End Devices (EDs)* and the *FL server (FLS)* act as the cache node for model downloads and as the aggregator for gradient uploads. Extensive packet-level performance studies indicate that INP could successfully reduce both the traffic load of the FL server and the needed times of model downloads and gradient uploads.

Regarding future directions, besides a full-fledged study of threats and the design of solutions, we argue that extending INP to support a broader range of application types and diverse network environments is attractive. On the one hand, as is known, in-network processing has many promising applications in domains such as data analysis, security enhancement, and system coordination [60]; however, providing transport protocol support for them is a crucial yet open problem [8], calling for future study. On the other hand, for abundant

intra-datacenter distributed applications benefiting from in-network processing (e.g., DC-DML [22], [30]), their involved aggregators (e.g., P4 switches) generally support only a limited set of simple operations and the available cache memories are scarce [21], [26], [28]. Designing domain-specific transport protocols and schemes to achieve features like *progress synchronization* and *job-scale-aware fair resource allocation* for them using these capacity-limited in-network devices is crucial but quite challenging.

References

[1] S. Luo *et al.*, “Eliminating communication bottlenecks in cross-device federated learning with in-network processing at the edge,” in *Proceedings of IEEE ICC*, 2022, pp. 4601–4606.

[2] C. Niu *et al.*, “Billion-scale federated learning on mobile clients: A submodel design with tunable privacy,” in *Proceedings of the 26th MobiCom*, 2020.

[3] Y. Shi *et al.*, “Communication-efficient edge ai: Algorithms and systems,” *IEEE Communications Surveys and Tutorials*, vol. 22, no. 4, pp. 2167–2191, 2020.

[4] Y. Zhan *et al.*, “A learning-based incentive mechanism for federated learning,” *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 6360–6368, 2020.

[5] P. Kairouz *et al.*, “Advances and open problems in federated learning,” *Found. Trends Mach. Learn.*, vol. 14, no. 1–2, pp. 1–210, jun 2021.

[6] K. Bonawitz *et al.*, “Towards federated learning at scale: System design,” *Proceedings of Machine Learning and Systems*, vol. 1, pp. 374–388, 2019.

[7] S. J. Reddi *et al.*, “Adaptive federated optimization,” in *International Conference on Learning Representations*, 2021.

[8] B. E. Stephens *et al.*, “Tcp is harmful to in-network computing: Designing a message transport protocol (mtp),” in *Proceedings of the 20th HotNets*. New York, NY, USA: ACM, 2021, pp. 61–68.

[9] L. Liu *et al.*, “Client-edge-cloud hierarchical federated learning,” in *Proceedings of the IEEE ICC*, 2020.

[10] R. Stoenescu *et al.*, “In-net: In-network processing for the masses,” in *Proceedings of the 10th EuroSys*, 2015.

[11] A. Panda *et al.*, “Netbricks: Taking the v out of NFV,” in *Proceedings of the 12th OSDI*. USENIX Association, Nov. 2016, pp. 203–216.

[12] Y.-Y. Shih *et al.*, “An nfv-based service framework for iot applications in edge computing environments,” *IEEE Trans. Netw. Service Manag.*, vol. 16, no. 4, pp. 1419–1434, 2019.

[13] P. Jin *et al.*, “Latency-aware vnf chain deployment with efficient resource reuse at network edge,” in *Proceedings of IEEE INFOCOM*, 2020, pp. 267–276.

[14] L. Zhang *et al.*, “Named data networking,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 66–73, Jul. 2014.

[15] C. Yang *et al.*, “Flash: Heterogeneity-aware federated learning at scale,” *IEEE Transactions on Mobile Computing*, vol. 23, no. 1, pp. 483–500, 2024.

[16] L. Mai *et al.*, “Kungfu: Making training in distributed machine learning adaptive,” in *Proceedings of the 14th OSDI*, 2020, pp. 937–954.

[17] S. Luo *et al.*, “Efficient cross-cloud partial reduce with crew,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 11, pp. 2224–2238, 2024.

[18] X. Li *et al.*, “On the convergence of fedavg on non-iid data,” in *International Conference on Learning Representations*, 2020.

[19] Z. Charles *et al.*, “On large-cohort training for federated learning,” in *Advances in Neural Information Processing Systems*, vol. 34. Curran Associates, Inc., 2021, pp. 20 461–20 475.

[20] M. Ye *et al.*, “Heterogeneous federated learning: State-of-the-art and research challenges,” *ACM Computing Surveys*, vol. 56, no. 3, Oct. 2023.

[21] A. Sapio *et al.*, “Scaling distributed machine learning with In-Network aggregation,” in *Proceedings of the 18th NSDI*. USENIX Association, Apr. 2021, pp. 785–808.

[22] C. Lao *et al.*, “ATP: In-network aggregation for multi-tenant learning,” in *Proceedings of the 18th NSDI*. USENIX Association, Apr. 2021.

[23] Z. Li *et al.*, “A2tp: Aggregator-aware in-network aggregation for multi-tenant learning,” in *Proceedings of the 18th EuroSys*. New York, NY, USA: ACM, 2023, pp. 639–653.

[24] Z. Li *et al.*, “Pa-atp: Progress-aware transmission protocol for in-network aggregation,” in *Proceedings of ICNP*, 2023, pp. 1–11.

[25] D. De Sensi *et al.*, “Canary: Congestion-aware in-network allreduce using dynamic trees,” *Future Gener. Comput. Syst.*, vol. 152, no. C, pp. 70–82, Mar. 2024.

[26] Y. He *et al.*, “A generic service to provide in-network aggregation for key-value streams,” in *Proceedings of the 28th ASPLOS*. New York, NY, USA: ACM, 2023, pp. 33–47.

[27] S. Liu *et al.*, “In-network aggregation with transport transparency for distributed training,” in *Proceedings of the 28th ASPLOS*. New York, NY, USA: ACM, 2023, pp. 376–391.

[28] B. Zhao, W. Wu, and W. Xu, “NetRPC: Enabling In-Network computation in remote procedure calls,” in *Proceedings of the 20th NSDI 23*. Boston, MA: USENIX Association, Apr. 2023, pp. 199–217.

[29] N. Gebara, M. Ghobadi, and P. Costa, “In-network aggregation for shared machine learning clusters,” in *Proceedings of Machine Learning and Systems*, vol. 3, 2021, pp. 829–844.

[30] S. Luo *et al.*, “Releasing the power of in-network aggregation with aggregator-aware routing optimization,” *IEEE/ACM Transactions on Networking*, vol. 32, no. 5, pp. 4488–4502, 2024.

[31] S. Luo *et al.*, “Fast parameter synchronization for distributed learning with selective multicast,” in *Proceedings of IEEE ICC*, 2022, pp. 4775–4780.

[32] S. Luo *et al.*, “Efficient file dissemination in data center networks with priority-based adaptive multicast,” *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 6, pp. 1161–1175, 2020.

[33] S. Luo *et al.*, “Efficient multisource data delivery in edge cloud with rateless parallel push,” *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 10 495–10 510, 2020.

[34] S. Luo *et al.*, “Efficient parameter synchronization for peer-to-peer distributed learning with selective multicast,” *IEEE Transactions on Services Computing*, vol. 18, no. 1, pp. 156–168, 2025.

[35] R. Silva *et al.*, “In-network computing—challenges and opportunities,” *Internet Technology Letters*, vol. 7, no. 3, p. e487, 2024.

[36] J. Liu *et al.*, “Venn: Resource management across federated learning jobs,” arXiv:2312.08298, 2023.

[37] L. Luo *et al.*, “Communication-efficient federated learning with adaptive aggregation for heterogeneous client-edge-cloud network,” *IEEE Transactions on Services Computing*, vol. 17, no. 6, pp. 3241–3255, 2024.

[38] X. Li, Y. Zhao, and C. Qiao, “Rcsr: Robust client selection and replacement in federated learning,” in *Proceedings of ICPADS*, 2023, pp. 1577–1584.

[39] M. Polese *et al.*, “A survey on recent advances in transport layer protocols,” *IEEE Communications Surveys and Tutorials*, vol. 21, no. 4, pp. 3584–3608, 2019.

[40] S. Vahidian *et al.*, “Rethinking data heterogeneity in federated learning: Introducing a new notion and standard benchmarks,” *IEEE Transactions on Artificial Intelligence*, vol. 5, no. 3, pp. 1386–1397, 2024.

[41] N. Rodríguez-Barroso *et al.*, “Survey on federated learning threats: Concepts, taxonomy on attacks and defences, experimental study and challenges,” *Information Fusion*, vol. 90, pp. 148–173, 2023.

[42] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 7th ed. Pearson, 2017.

[43] H. Krawczyk, M. Bellare, and R. Canetti, “Rfc2104: Hmac: Keyed-hashing for message authentication,” USA, 1997.

[44] C. Zhang *et al.*, “Batchcrypt: Efficient homomorphic encryption for cross-silo federated learning,” in *Proceedings of USENIX ATC*, Jul. 2020, pp. 493–506.

[45] Z. Chen *et al.*, “Unifl: Enabling loss-tolerant transmission in federated learning,” in *Proceedings of the 8th APNet*. ACM, 2024, pp. 163–168.

[46] Y. Qiao *et al.*, “Netec: Accelerating erasure coding reconstruction with in-network aggregation,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 10, pp. 2571–2583, 2022.

[47] J. Xia *et al.*, “Parallelized in-network aggregation for failure repair in erasure-coded storage systems,” *IEEE/ACM Transactions on Networking*, vol. 32, no. 4, pp. 2888–2903, 2024.

[48] Y. Qiao, H. Zhang, and J. Wang, “Netfec: In-network fec encoding acceleration for latency-sensitive multimedia applications,” in *Proceedings of IEEE INFOCOM*, 2024, pp. 2348–2357.

[49] H. Zhou *et al.*, “Dgt: A contribution-aware differential gradient transmission mechanism for distributed machine learning,” *Future Generation Computer Systems*, vol. 121, pp. 35–47, 2021.

[50] S. Luo *et al.*, “Meeting coflow deadlines in data center networks with policy-based selective completion,” *IEEE/ACM Transactions on Networking*, vol. 31, no. 1, pp. 178–191, 2023.

[51] H. Wang *et al.*, “Towards Domain-Specific network transport for distributed DNN training,” in *Proceedings of the 21st NSDI*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 1421–1443.

[52] W. Han *et al.*, "Beyond throughput and compression ratios: Towards high end-to-end utility of gradient compression," in *Proceedings of the 23rd HotNets*, ser. HotNets '24. ACM, 2024, pp. 186–194.

[53] X. Liu *et al.*, "Approximate gradient synchronization with aqgb," in *Proceedings of the 6th APNet*. New York, NY, USA: ACM, 2023, pp. 101–102.

[54] D. Wen, K.-J. Jeon, and K. Huang, "Federated dropout—a simple approach for enabling federated learning on resource constrained devices," *IEEE Wireless Communications Letters*, vol. 11, no. 5, pp. 923–927, 2022.

[55] C. Feng *et al.*, "Mobility-aware cluster federated learning in hierarchical wireless networks," *IEEE Transactions on Wireless Communications*, vol. 21, no. 10, pp. 8441–8458, 2022.

[56] A. Chougule *et al.*, "A novel framework for traffic congestion management at intersections using federated learning and vertical partitioning," *IEEE Transactions on Consumer Electronics*, vol. 70, no. 1, pp. 1725–1735, 2024.

[57] Y. Watanabe, Y. Kawamoto, and N. Kato, "A novel routing control method using federated learning in large-scale wireless mesh networks," *IEEE Transactions on Wireless Communications*, vol. 22, no. 12, pp. 9291–9300, 2023.

[58] F. Tang *et al.*, "Survey on machine learning for intelligent end-to-end communication toward 6g: From network access, routing to traffic control and streaming adaption," *IEEE Communications Surveys and Tutorials*, vol. 23, no. 3, pp. 1578–1598, 2021.

[59] E. T. Martínez Beltrán *et al.*, "Decentralized federated learning: Fundamentals, state of the art, frameworks, trends, and challenges," *IEEE Communications Surveys and Tutorials*, vol. 25, no. 4, pp. 2983–3013, 2023.

[60] S. Kianpisheh and T. Taleb, "A survey on in-network computing: Programmable data plane and technology specific applications," *IEEE Communications Surveys and Tutorials*, vol. 25, no. 1, pp. 701–761, 2023.

Shouxi Luo (Member, IEEE) received the bachelor's degree in communication engineering and the Ph.D. degree in communication and information systems from the University of Electronic Science and Technology of China, China, in 2011 and 2016, respectively. He is currently an Associate Professor with Southwest Jiaotong University. His research interests include data center networks, software-defined networking, and networked systems.

Peidong Zhang received the bachelor's degree in computer science and technology from Henan University, China, in 2022. Currently, he is pursuing the master's degree in electronic information at Southwest Jiaotong University. His research interests include in-network computing and communication system optimization.

Xin Song is currently pursuing the master's degree in computer technology at Southwest Jiaotong University. His research interests include distributed deep learning and networked systems.

Pingzhi Fan (Fellow, IEEE) received the M.Sc. degree in computer science from Southwest Jiaotong University, China, in 1987, and the Ph.D. degree in electronic engineering from Hull University, U.K., in 1994. He is currently a Presidential Professor with Southwest Jiaotong University. His research interests include high mobility wireless communications, massive random-access techniques, etc. He is a fellow of IEEE, IET, CIE, and CIC.

Huanlai Xing (Member, IEEE) received the B. Eng. degree in communications engineering from Southwest Jiaotong University, China, in 2006, the M. Eng. degree in electromagnetic fields and wavelength technology from the Beijing University of Posts and Telecommunications, China, in 2009, and his Ph.D. degree in computer science from the University of Nottingham, U.K., in 2013. Currently, he is an Associate Professor with Southwest Jiaotong University. His research interests include mobile edge computing, evolutionary computation, etc.

Long Luo received the M.S. and Ph.D. in communication and information systems from the University of Electronic Science and Technology of China, China, in 2015 and 2020, respectively. She is currently an Associate Professor with the University of Electronic Science and Technology of China. Her research interests include networking and distributed systems, etc.

Hongfang Yu (Senior Member, IEEE) received the Ph.D. degree in communication and information systems from the University of Electronic Science and Technology of China, China, in 2006. She is currently a Professor with the University of Electronic Science and Technology of China. Her research interests include SDN/NFV, data center networks, networking for AI systems, and network security, etc.