Sensor Fusion on the Edge: Initial Experiments in the EdgeServe System

Ted Shaowang
University of Chicago
USA
swjz@uchicago.edu

Xi Liang
University of Chicago
USA
xiliang@uchicago.edu

Sanjay Krishnan
University of Chicago
USA
skr@uchicago.edu

ABSTRACT

Due to latency and privacy concerns, we are witnessing the rise of edge computing, where computation is placed close to the point of data collection to facilitate low-latency decision making. However, we believe that a very important class of *sensor fusion* applications, in which data generated in a disaggregated way has to be combined to make a decision, are not well understood in the context of edge computing. The necessary data needs to be in "the right place at the right time", making intra-edge communication a significant bottleneck. In prior work, we proposed an edge-based model serving system, called EdgeServe, that not only manages a machine learning inference service, but also orchestrates data movement between nodes on an edge network. In this paper, we evaluate trade-offs in temporal synchronization between data sources, and present initial experiments that study how different knobs can affect the performance of sensor fusion applications.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; *Sensor networks.*

KEYWORDS

federated inference, edge computing

ACM Reference Format:
Ted Shaowang, Xi Liang, and Sanjay Krishnan. 2022. Sensor Fusion on the Edge: Initial Experiments in the EdgeServe System. In *Big Data in Emergent Distributed Environments (BiDEDE'22), June 12, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3530050.3532924

1 INTRODUCTION

Serving predictions from machine learning models is a crucial part of modern real-time decision systems. The basic idea is to interface trained machine learning models (e.g., a neural network or an SVM) to software clients who can use those predictions (e.g., a fraud detection framework). Current examples include Clipper [4], TensorFlow Serving [9], and InferLine [3], were designed as cloud services. As the uses for machine learning have evolved towards increasingly latency and communication-sensitive applications, such as in control systems, industrial monitoring, and mobile applications, there has been a steady trend towards moving model-serving to resources closer to the point of data collection. We collectively call these computation resources "the edge". The primary focus of model serving on the edge has been to design reduced-size models that can efficiently be deployed on lower-powered devices [5–7, 13].

Simply reducing the computational footprint of each prediction served is only part of the problem in emerging edge applications. *Sensor fusion* is the process of combining sensor data or data derived from disparate sources. For example, in a warehouse analytics example, one might combine data from multiple cameras to track the movement of a package through the warehouse. Or, in a self-driving car, one might bring together inputs from multiple radar sensors, lidar sensors, and cameras to form a single model or image of the environment around a vehicle. In general, all such examples have a common network topology, where data are collected on different endpoints and have to be combined, or "fusion", somewhere in an edge network before a decision can be made. Furthermore, the data must be temporally synchronized when it is combined so that temporally corresponding observations from the different sensors are linked, i.e., the final model integrates data from the same "timestep". This need for synchronized fusion creates a very interesting pair of coupled systems bottlenecks: (1) edge-based sensor fusion tasks can generate a large amount of intra-edge communication creating network contention and saturating bandwidth constraints, (2) jitter in the network can force buffering to ensure that observations are synchronized.

The consequence of (1) and (2) is an increased effective decision latency which is not desirable in edge computing. If one relaxes the need for temporal synchronization, certain types of applications can very quickly degrade in terms of accuracy. Consider the warehouse asset tracking example above. Imagine, that we wanted to triangulate the position of an asset from multiple cameras using a standard stereo correspondence algorithm. In Section 5, we constructed experiments where two cameras connected over an Ethernet network simultaneously tracked a box with a QR code and compared the tracking results to a centralized baseline. We found that temporal synchronization on its own introduced errors of up to 30 pixels in tracking, which would compound the real-world distance for tracking objects relatively far away from the camera. Thus, even over high-bandwidth networks, temporal synchronization is a major challenge, which only becomes worse if the network is more constrained.

In prior work, we proposed an edge-based model serving system, called EdgeServe [12], that not only manages a machine learning inference service but also orchestrates data movement between nodes on an edge network. EdgeServe provides a message broker

BiDEDE'22, June 12, 2022, Philadelphia, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9346-1/22/06…$15.00
https://doi.org/10.1145/3530050.3532924

service that allows producers to push sensor data into a topic, and consumers to listen to data on those topics. This interface is similar to that of ROS [10]. EdgeServe automatically reasons about a network of heterogeneous and disaggregated resources. The system automatically deploys models (consumers of data) to the specified nodes. This workshop paper deep dives into sensor fusion tasks in EdgeServe and illustrates many of the counter-intuitive tradeoffs between latency, communication, and mitigating temporal synchronization errors. The experiments in Section 5, while simple, illustrate 3 key lessons that we hope to use to inform the next iteration of EdgeServe.

- *Lesson 1. Lowering the data resolution can improve accuracy in sensor fusion tasks that are sensitive to temporal synchronization.* Counter-intuitively, degrading the quality of the data collected from the sensors can lower the frequency of incoming data, and thus reduce misalignment caused by variability in processing, communication, and queuing. This actually leads to more accurate final results than if high-resolution data were used. (See discussions in Section 4.2, 4.4 and experiments in Section 5.5.1.)
- *Lesson 2. Placing computation as close to the data source as possible is not always desirable.* It is a natural thought that moving inference closer to the point of data collection is desirable in edge computing. While this approach might significantly reduce latency, it can introduce additional errors due to temporal synchronization. A model can run faster for some data points but slower for other data points, and this variability can cause misalignment between sensors. (See discussions in Section 4.3 and experiments in Section 5.5.2.)
- *Lesson 3. Eager data movement can create points of contention on the message broker.* We propose a lazy approach that simply sends headers to a centralized message broker and data are lazily transferred in a peer-to-peer fashion. While this adds an overhead it cuts down a significant source of variability in queuing if there is contention on the message broker. (See discussions in Section 4.1 and experiments in Section 5.5.3.)

2 EDGESERVE OVERVIEW

The primary goal of EdgeServe is to facilitate machine learning inference on the edge where data sources may have to be *routed through the network before prediction.* Unlike existing model-serving systems, EdgeServe also orchestrates how data moves through the edge network. This architecture is described in prior work [12].

Sensor Fusion and Multi-modal Prediction. A machine learning model is *multi-modal* if it requires data from more than one data source. For example, a model that leverages multiple cameras for tracking a box in a warehouse. Or, a model that integrates information from multiple sensors to detect room occupancy.

Fundamentally, real-time sensor fusion requires temporally corresponding data to be "in the same place at the same time" somewhere in an edge network. EdgeServe is a system designed to address this constraint by ensuring that right data is routed to each model. Figure 1 illustrates the concept of temporal synchronization. Ideally, sensor measurements from exactly the same time point should be paired during multi-modal inference (Figure 1A-B). Since in any real distributed sensing system this is impossible, there are only degrees of synchronization that can be achieved (i.e., how similar are the timestamps of the observations from different sources).

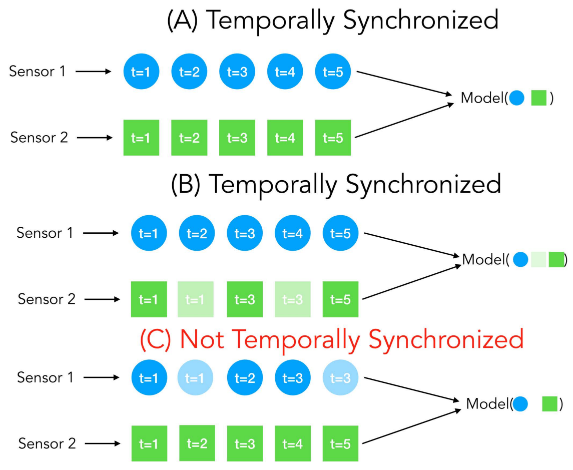

Figure 1: (A) Temporal synchronization means that observations created at the same (or roughly the same) time are paired together during multimodal inference. (B) The same idea can be extended to windows of data if different sensors produce data at different rates. (C) An example of synchronization breaking due to an irregular data processing schedule.

2.1 Basics and Workflow

We assume that each edge *node* is connected to others on a TCP/IP network (either directly or via a switched network). A subset of these nodes are physically connected to data sources (e.g. video cameras, sensors, and other data streams). Every node maintains a globally-synchronized catalog of data streams that are locally collected. In EdgeServe, *models* are functions that are repeatedly applied to fixed windows of data. Every model in EdgeServe has *locality constraints*, which describe where a model's prediction results have to be delivered. For example, one could require that all predictions are delivered to node1. Or, we could require that either node1 or node2 has the required output.

Unlike existing model-serving systems that work asynchronously, EdgeServe works in a push-based model, where the arrival of each new data batch (defined by the user's inference task) triggers re-evaluation. These tasks subscribe to a message-broker service, which informs each node about new data. Each model can consume one or more sources of data and yield a new stream (a prediction). Since there is a global message-broker service, the output of models can be streamed to other models as well. Models that consume multiple streams of data induce additional locality constraints, where data streams from multiple nodes may have to be aggregated in a central place. For example, an activity recognition model that requires video, audio, and network data must aggregate all of the data in a single place somewhere in the network. At a high level, EdgeServe combines a publication-subscription system to facilitate communication between multiple model-serving nodes on a network. To the best of our knowledge, such a system does not exist in part due to the challenges in routing, scheduling, and placement. The key system goal of EdgeServe is to provide a centralized control plane to find placement, routing decisions, and model partitioning decisions that satisfy locality and hardware constraints.

2.2 Metrics

The main performance metric that we care about is *timeliness*, which is the time delay between data arrival and the prediction results arriving at the appropriate node in the edge network (independent of exactly where and how EdgeServe chose to execute that process).

Time-to-Decision (Timeliness). Unlike existing model serving systems where predictions are triggered by client requests, EdgeServe will continuously process predictions over streams of incoming data. Therefore, the concept of "latency" is a little more complicated in this setting. Accordingly, we define a new metric called timeliness, which is the gap between the time at which the data arrived and when a prediction was issued. The start time is defined as the time point at which all of the relevant data for a particular prediction is available somewhere on the network, and the end time is the time at which the prediction is issued and communicated to the appropriate edge node that can use the prediction.

The focus on model-serving makes the design of EdgeServe particularly interesting, because optimization decisions that improve the timeliness of predictions may affect their accuracy:

(1) *Prediction Accuracy (Accuracy).* We also care about the accuracy of the predictions that are made, or the gap between the prediction of a class or a continuous label and (hypothetical) ground-truth.
(2) *Robustness to Failure (Robustness).* Finally, it is also important to consider robustness to network and node failures. These failures can affect both the placement of computation and the availability of source data. Robustness is measured in terms of the number and type of edge nodes that can be lost while still issuing a prediction.

All three of these metrics have both systems and machine learning implications. For example, there are systems solutions to improving timeliness through batching and locality, but there are also machine learning solutions where different model types have latency characteristics. Similarly, systems techniques like replication can help tolerate failures, but robust machine learning techniques can also allow for issuing predictions even if some of the features are lost.

2.3 Implementation

To better address the aforementioned challenges in routing, scheduling, and placement, we design our system with the following principles in mind.

2.3.1 Declarative specification of the system. We provide a user interface for the user to configure the system in a declarative manner. Users can specify the topology of the network, e.g., which nodes are data sources, which nodes are edge nodes and how are they connected; the tasks that are to be processed, e.g., which data sources should be combined and which model should be used for processing; and the objectives of the system, e.g., should the system aim for better timeliness or better accuracy.

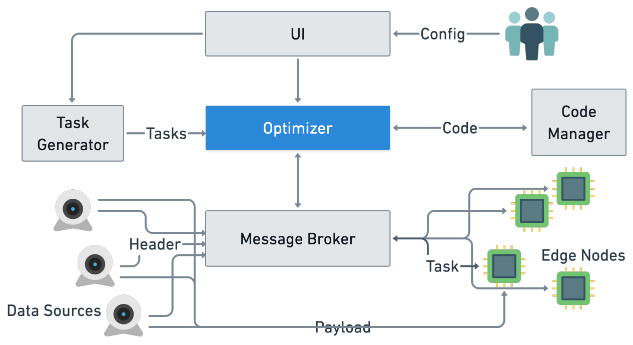

Figure 2: Architecture of the EdgeServe system.

2.3.2 Declarative serving and processing. The challenging decision-making for the underlying routing, scheduling, and placement decisions should be hidden from the user and automatically determined by an optimizer given user-specified network topology, task specification, and objective.

2.3.3 Efficient communication. A central message broker is used to relay messages among the optimizer, the data source, and the edge nodes. Unlike other systems that transfer the entire data payload using a message broker, in our system, only the header of the data is distributed to the edge nodes and the payload is lazily accessed on demand by the edge nodes that are responsible for processing the payload.

2.3.4 Extensible tasks. A task generator is built to use some of the user input to generate tasks that contain a unified interface used by the edge nodes. This can decouple the optimizer from the user configuration which makes it easy to support new types of tasks. A code manager is also integrated to serve the same purpose, by exposing APIs for edge nodes to access the right code or model used by a task, the code can be white-boxed (Python code) or even black-boxed (e.g., a Docker container).

2.4 Example Execution

Let's consider the warehouse asset tracking example in the introduction. There are three *streams* of data: audio, video, and network traffic. Audio and video are collected on node1 (an Intel Video Processing Embedded System) and network traffic is collected on node2 (a programmable wireless access point). We have a *model* which is a neural network that requires all three data sources to predict ongoing activities in the warehouse. To issue such predictions, the system could create a data flow (via publication and subscription) that repeatedly transfers raw data from node2 to node1, and host a comprehensive model on node1. Alternatively, it could also featurize the network traffic data locally on node2 and only transfer pre-processed features to node1. node1 applies a pooling method to issue a prediction based on features from multiple sources. This allows the user to combine the sources of data for a richer prediction, as well as leverage the specialized prediction hardware on both nodes.

3 RELATED WORK

Current machine learning model serving systems including Clipper [4], TensorFlow Serving [9], and InferLine [3] all assume that the user has manually programmed all necessary data movement. Recent systems have begun to realize the underappreciated problem of data movement and communication-intensive aspects of modern AI applications, but have yet to address the trade-offs in temporal synchronization between different data sources when they do not arrive at the same time. For example, Hoplite [14] generates data transfer schedules specifically for asynchronous collective communication (e.g., broadcast, reduce) operations in a task-based framework. [8, 11]

Traditional relational stream processing systems, e.g., [2], have very strict requirements for temporal synchronization where they model such an operation as a temporal join. These systems will buffer data, indefinitely if needed, to ensure that corresponding observations are properly linked. While desirable for relation query processing, this approach is excessive in machine learning applications which have to tolerate some level of inaccuracy anyways. Moreover, multi-modal machine learning inference usually involves data sources generated at different rates. In this setting, a looser level of synchronization would be beneficial to the system and improve performance.

In the context of sensing, ROS (Robot Operating System) [10] is an open-source framework designed for robotics research. It incorporates an algorithm called `ApproximateTime` that tries to match messages coming on different topics at different timestamps. This algorithm can drop messages on certain topics if they arrive too frequently, but does not use any message more than once. In other words, if one sensor sends data very infrequently, the algorithm will have to wait and drop messages from all other sensors until it sees a new message from the low-frequency sensor to issue a match. On top of that, such a wait can be harmful to end-to-end timeliness and accuracy, especially in a synchronization-sensitive scenario where high-frequency information is lost. If we assume that temporal correlation exists in sensor data, an alternative to this algorithm would be to reuse the last known value from the low-frequency sensor and match it with other sensors when they are ready.

4 TRADE-OFFS IN TEMPORAL SYNCHRONIZATION

In this section, we evaluate variable control of temporal synchronization in EdgeServe by illustrating how different knobs might affect timeliness and synchronization errors. Our key question here is, how do we loose our synchronization constraints to an extent that we are still able to make accurate and timely inferences?

4.1 Knob 1. Time Interval for Message Matching

The message broker has the privilege of retaining a message until it receives matching messages from other sensors, leaving room for a matching algorithm like the one in ROS. The effect of such forced synchronization on the message broker side heavily depends on a reasonable time interval between predictions. If the time interval is too short, such synchronization can be ineffective without improving accuracy; if the time interval is too long, potential long waits can stretch end-to-end timeliness while harming accuracy due to lost high-frequency information in between. Finding such a task-specific time interval can be challenging and tedious for humans and we need a system to automate this process. In order for such a system like ROS to implement a matching algorithm, it also has to eagerly queue up messages on the message broker side, which can face contention when such messages are large in size. Experiments in Sec. 5.5.3 show how such contention affects latency. We show that a lazy method of message passing can significantly mitigate the effects of contention.

4.2 Knob 2. Down-sampling

Down-sampling, or temporally sampling the data from each sensor, can drastically improve end-to-end timeliness by reducing data transfer when communication is costly for the task. In down-sampling, there is a rate-limit on how much data each data source produces. The consumer can still predict at a faster rate by using a last known observation from a source if needed. Not only does down-sampling reduce computation and communication, it also increases the time between messages which allows the system to tolerate more variability (and thus better temporal synchronization). So, the basic tradeoff is whether the loss in accuracy due to down-sampling is worth the better synchronization. Interestingly enough, the answer seems to be "sometimes". Experiments in Sec. 5.5.1 show how down-sampling affects latency and accuracy.

4.3 Knob 3. Compute Placement

It seems a good idea to place computation closer to the point of data collection, at least from the perspective of improving end-to-end timeliness for communication-intensive tasks. However, in real-world use cases, the latency of such computation varies across nodes and can be very hard to predict in advance. Such variability in latency can become a new source of synchronization problem and result in lower accuracy. Experiments in Sec. 5.5.2 show how compute placement affects latency and accuracy.

4.4 Knob 4. Lossy Compression

Similar to down-sampling, lossy compression also affects synchronization. Compression can also significantly reduce file size and ease the burden of communication between nodes, improving end-to-end timeliness for communication-intensive tasks. Smaller network payloads are less susceptible to network jitter. Furthermore, compression requires some non-trivial computation and acts as a rate-limit on the data source side, much in the same way down-sampling does. Again, if the consumer wishes to predict faster than this rate-limited source the last-known observation can be used. Experiments in Sec. 5.5.1 show how lossy compression affects latency and accuracy.

5 EXPERIMENTS: MULTI-CAMERA TRACKING

Our goal for this paper is to simulate a synchronization sensitive task and demonstrate the trade-off between latency, accuracy and communication. In order to achieve this goal, we set up a QR code detector where two webcams capture the same QR code from different positions. We move the physical position of the QR code along a horizontal axis and observe the QR code positions detected by both cameras. We compare the trajectory of positions to a centralized baseline where both cameras collect data on the same node to evaluate accuracy.

5.1 Hardware Setup

Our hardware setup consists of two 1080p webcams and four Intel Skylake NUC computers, each equipped with an Intel Core i3-6100U CPU, 16 GB memory and M.2 SSD. In a centralized setting (Sec. 5.4), only one NUC computer is used and it is connected to both webcams. In a distributed setting (Sec. 5.5), all four NUC computers are used: two of them are connected to two webcams respectively serving as data source nodes, one of them serves as a message broker and the other NUC serves as the compute node taking input from data source nodes. All four NUCs are interconnected via 100Mbps Ethernet.

5.2 Software Setup

We use Apache Pulsar [1] as the message broker to transfer messages between NUC computers. For small messages such as a 2D array, we transfer them directly via Pulsar. For larger files such as images, we create FTP paths for them and transfer those paths in messages for the compute node to download, saving traffic on the message broker side. For QR code detection, we use an OpenCV resolution with two CNN-based Caffe models: an object detection model to detect the QR code with a bounding box and a super-resolution model to zoom in the QR code when it is small. Videos are collected in advance to ensure reproducibility and we simulate real-time streaming of these videos. All results present the average values of 3 experiments. All videos used in the following experiments are of 1920x1080 resolution, 5 seconds long at 30 FPS unless otherwise specified. The size of QR code is about 200x200.

5.3 Metrics and Ground Truth

In this experiment, we define accuracy as pixel-level 'error', which is the difference between ground truth trajectory and the experiments in both x and y axes in the unit of absolute pixels, averaged over all frames. The smaller the absolute number of 'error', the more accurate the target experiment is. The ground truth of such offset is defined as the offset between two videos in a centralized setup without compression (Table 1(a)-(d)). If there is no QR code detected from a certain frame in the target experiment, we use the last known QR code position for that camera. For down-sample experiments, we up-sample the missing frames with the last known frame when measuring accuracy as well. We also define latency as the time period from the timestamp when the first piece of data is transferred until the timestamp when the prediction for the last piece of data is issued. Since all of our videos have the same length of 5 seconds, this metric is a proxy for "timeliness" defined before.

	Fast movement	Slow movement
a) Memory	9.92s	10.52s
b) Disk (uncompressed)	13.53s (+3.61s, 36%)	13.75s (+3.23s, 31%)
c) Disk (jpeg)	29.76s (+19.84s, 200%)	31.16s (+20.64s, 196%)

Table 1: Latency from a centralized compute: (a)-(c) involve the same task with different level of disk access. Numbers and percentages in parentheses are relative to (a).

JPEG accuracy	Fast movement	Slow movement
x-axis	0.5931px	0.0050px
y-axis	0.3868px	0.0022px

Table 2: Errors introduced by JPEG compression.

5.4 Centralized Compute

As a baseline, we consider a scenario where the computation is centralized. There is no communication or synchronization issue in this case. Therefore, we treat the result from this run as ground truth and running time as a baseline. We demonstrate the intrinsic characteristics between fast and slow movements of the QR code and explore the latency component of disk I/O to get a better understanding of the task.

Table 1 shows the latency from centralized multi-camera tracking example: (a) both streams are captured and passed to EdgeServe in memory; (b) the camera streams are stored to disk in an uncompressed format and incrementally retrieved by EdgeServe; and (c) the camera streams are stored to disk in a JPEG format and incrementally retrieved by EdgeServe.

First, we look at the differences between fast movement and slow movement columns. When we move the QR code too fast, quite a few frames are too blurry for the detector to recognize anything so the decoding step is skipped. Therefore, it takes a little shorter time to finish the computation in fast movement cases. Second, we compare Table 1(b) against (a) to measure the latency of disk I/O. Specifically, reading and writing image files from/to disk takes about the same amount of time and they add up to about 30-40% of compute time. We see from Table 1(c) that disk I/O with JPEG runs significantly longer because extra time is spent on compression and decompression.

It should be noted that images in BMP format are lossless and the size of each BMP file is about 6 MB. JPEG compression is lossy but could significantly reduce file size to 200-300 KB. The accuracy of JPEG compression is shown in Table 2. In both axes, we see an average error of less than 1 pixel, which is nearly perfect. Fast movement is worse because some blurry frames that are recognizable as BMP files are no longer recognizable after JPEG compression.

5.5 Distributed Edge Cluster

As described in Sec. 5.1, we construct an Edge cluster and do the same QR code detection task, where data is collected on different nodes. We build a queue for each data source at the message broker and they are aggregated to make a prediction as soon as there is new data coming in from any data source. A data point may be reused to make a joint prediction if it is still the latest from an infrequent data

	Size	Time	x-axis error	y-axis error
BMP (30 FPS)	1.7 GB	3m 11s (2m 41s)	22.8402px	4.1812px
BMP (10 FPS)	593 MB	1m 4s (53.8s)	7.3501px	1.7942px
BMP (5 FPS)	297 MB	32.1s (27.1s)	29.4458px	1.5753px
JPEG (30 FPS)	72 MB	49.8s (8.3s)	4.0714px	0.4754px
JPEG (10 FPS)	24 MB	16.6s (2.9s)	6.6651px	2.7245px
JPEG (5 FPS)	12 MB	8.2s (1.6s)	29.4797px	1.6021px

Table 3: BMP, JPEG and down-sampling comparison. Numbers in parentheses are time spent purely on data download, measured by `wget`.

Endpoint-Placement	Time	x-axis error	y-axis error
Original 30 FPS	5.79s	22.4691px	2.7398px
Down-sampled to 10 FPS	5.93s	24.7646px	5.6231px
Down-sampled to 5 FPS	5.55s	29.5273px	1.7011px

Table 4: Latency and accuracy for Endpoint-Placement.

source. Jitter in the network, variability in processing times, and queuing delays can introduce extra errors. We compare these errors to the sub-pixel errors introduced by lossy compression above.

5.5.1 Effect of Down-sampling and Compression on Accuracy. Counter-intuitively, degrading the quality of the data collected from the sensors can lead to better results in the distributed setting. We compare compression and effects of down-sampling in Table 3. We find that down-sampling can almost linearly improve latency because it directly reduces data transfer. It also forces synchronization on the data source side to improve accuracy until it reaches a sweet spot, after which we lose so much information between frames that accuracy starts to decrease. JPEG compression has a similar effect of reducing data transfer significantly, which also improves latency compared to BMP counterparts. Since JPEG compression takes a while (as shown in Table 1), it acts as a rate limit at the data source, similar to the down-sampling method, which also forces synchronization. The accuracy 'sweet spot' for JPEG is reached at the original sampling rate and further down-sampling would only decrease accuracy.

5.5.2 Compute Placement and Synchronization Errors. It is a natural thought that moving inference closer to the point of data collection is desirable in edge computing; this is not always the case in multimodal prediction. Instead of transferring frames over the network, we could also run the model at data source nodes and only transfer coordinates over the network. This method makes use of model parallelism and spare resources on data source nodes to save communication between data source nodes and compute node(s). Such an "Endpoint-Placement" method is perfectly suitable for our use case because we have light models but heavy communication.

	Rate limit (up /down)	Time
Lazy (ours)	No limit	3m 10s
Lazy (ours)	1 Mbps / 1 Mbps	3m 12s
Eager (similar to ROS)	No limit	3m 16s
Eager (similar to ROS)	20 Mbps / 20 Mbps	21m 32s

Table 5: Latency with network bandwidth limits.

Table 4 shows that this method improves latency by 33x compared to BMP (30 FPS) in Table 3.

However, Endpoint-Placement methods can be a source of misalignment as well, especially when a model runs faster for some data points but slower for other data points. The variability in model inference latency across nodes can add up to become a synchronization problem and reduces accuracy. In Table 4, we see higher synchronization errors because of this variability.

5.5.3 Effect of Queuing Strategy. In Sec. 5.2 we described how we use FTP to transfer large data directly from data source to compute node and only pass pointers to data over the message broker. This is especially beneficial when the message broker is busy with network requests. We simulate a congestion scenario where the network bandwidth at the message broker is limited and measure the end-to-end latency of our system versus a ROS-like system that transfers raw data through a centralized broker. Table 5 shows that our system is tolerant to slow network bandwidth while transferring raw frames can be extremely slow when the network is congested.

6 CONCLUSION

In this paper, we discuss trade-offs in temporal synchronization by showing initial experiments of a multi-camera tracking example in our EdgeServe system. We find that synchronization errors and timeliness metrics depend on multiple knobs, as discussed in Section 4. We believe this topic is underappreciated and we hope our findings are useful for future similar systems.

REFERENCES

[1] Apache. 2022. Apache Pulsar. https://pulsar.apache.org/ (visited on March 27, 2022).

[2] Sirish Chandrasekaran and Michael J Franklin. 2003. PSoup: a system for streaming queries over streaming data. *The VLDB Journal* 12, 2 (2003), 140–156.

[3] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 477–491. https://doi.org/10.1145/3419111.3421285

[4] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, Aditya Akella and Jon Howell (Eds.). USENIX Association, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[5] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings.*

[6] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodik, Leana Golubchik, Minlan Yu, Paramvir Bahl, and Matthai Philipose. 2018. Videoedge: Processing camera streams using hierarchical clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC).* IEEE, 115–131.

[7] Sumit Maheshwari, Wuyang Zhang, Ivan Seskar, Yanyong Zhang, and Dipankar Raychaudhuri. 2019. EdgeDrive: Supporting advanced driver assistance systems

using mobile edge clouds networks. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 1–6.

[8] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 561–577.

[9] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. In *Workshop on ML Systems at NIPS 2017*.

[10] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.

[11] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, Vol. 130. Citeseer, 136.

[12] Ted Shaowang, Nilesh Jain, Dennis D Matthews, and Sanjay Krishnan. 2021. Declarative data serving: the future of machine learning inference on the edge. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2555–2562.

[13] Xiao Zeng, Biyi Fang, Haichen Shen, and Mi Zhang. 2020. Distream: scaling live video analytics with workload-adaptive distributed edge intelligence. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*. 409–421.

[14] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. 2021. Hoplite: efficient and fault-tolerant collective communication for task-based distributed systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 641–656.