Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

LiveDewStream: A stream processing platform for running in-lab distributed deep learning inferences on smartphone clusters at the edge

Cristian Mateos [*], Matías Hirsch, Juan Manuel Toloza, Alejandro Zunino

ISISTAN-UNICEN-CONICET, Tandil, Argentina

A R T I C L E   I N F O

Article history:
Received 23 June 2022
Received in revised form 7 November 2022
Accepted 15 November 2022

Keywords:
Mobile devices, Stream processing, Deep learning, Dew computing, Android

A B S T R A C T

Dew computing, an evolution of Fog computing, aims at fulfilling computing needs, such as deep learning applied to object classification, close to where data is originated and using computing resources that include consumer electronic devices such as smartphones. Simulation tools like DewSim aid the study of resource allocation mechanisms for exploiting clusters of smartphones, however, there is a gap w.r.t software tools that allow to perform similar studies over real Dew computing testbeds. We have developed LiveDewStream, an open source project to model executable tasks derived from data streams to be run on real smartphone clusters. The project offers a key functionality missing in other tools: reproducibility of battery-driven Dew experiments. Our major contribution is to provide the community a common in vivo platform to study best-performing allocation mechanisms under different stream processing scenarios and/or deep learning inference models.

Code metadata

Current code version	V1.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-22-00180
Permanent link to reproducible capsule	https://github.com/matieber/livedewstream
Legal code license	GNU GPL
Code versioning system used	git
Software code languages, tools and services used	Python, Android, shell scripting
Compilation requirements, operating environments and dependencies	The *emanager_server* and *scnrunner* modules run on Linux-based machines using Python 3.7+; Normapp runs on Android 6+ (it is built using the provided Android Studio project). See dependencies and installation instructions at the GitHub repository
If available, link to developer documentation/manual	https://github.com/matieber/livedewstream/doc
Support email for question	matias.hirsch@isistan.unicen.edu.ar

1. Motivation and significance

Fog computing [1] was introduced in 2012 to provide highly-scalable network and computing infrastructures for latency and location-aware (mobile) IoT applications, while augmenting resource-constrained devices with processing/storage resources in their proximity. Several alternatives to realize this idea, including cloudlets, mobile edge computing, micro datacenters, nano

* Corresponding author.
E-mail addresses: cristian.mateos@isistan.unicen.edu.ar (C. Mateos), matias.hirsch@isistan.unicen.edu.ar (M. Hirsch), juanmanuel.toloza@isistan.unicen.edu.ar (J.M. Toloza), alejandro.zunino@isistan.unicen.edu.ar (A. Zunino).

https://doi.org/10.1016/j.softx.2022.101268

datacenters and femto Clouds, have been introduced [2], which aim at processing data/computations using computing resources located at the edge of the network and optionally using remote resources in the distant Cloud when necessary.

Since then, there has been a tremendous growth in the amount of resource-rich devices and hence computational resources available at the closest edge. According to Statista.com, nearly 84% of the current world's population owns a smartphone. Modern smartphones contain, on average, more than a dozen sensors, up to eight cores, and powerful GPUs. Likewise, thousands of purpose-specific sensing devices such as surveillance cameras, smoke detectors, noise detectors, and so on are being deployed across buildings and cities around the globe. This has led to another paradigm shift by which relying on not-so-close Fog servers to support IoT applications consuming and processing the streams of data from such devices might not suffice. This is particularly true considering that today's IoT applications are becoming more commonplace and intelligent, and therefore the timely and efficient execution of increasingly complex tasks and the context-aware processing of larger amounts of data in urban scenarios is needed, which is difficult for centralized Clouds and challenging even for Fog infrastructures.

Dew computing proposes to establish clusters of mobile devices at the very edge [3], as a form of "ubiquitous and opportunistic computing" within data sensing contexts, specially in public places such as transport, classrooms and coffee shops, where many nearby devices are present. Hence, cost-effective platforms for intelligent IoT applications emerge, provided computational resources are managed wisely. Therefore, platforms and tools to study the individual/collective capabilities and limitations of smartphones for intelligent data stream processing are needed.

We present a software platform to support experimentation with a specific but broad family of such applications, i.e. those using deep learning over data streams. The software allows users, i.e. in-lab Dew researchers, to specify automatic, repeatable batch benchmark plans, while indicating specific smartphone-ready deep learning models and task scheduling algorithms for the cluster. Even when we have already proposed a Dew simulation software [3], our framework represents the first step towards "in vivo" benchmarking of mobile devices for such applications, pretty much like commercial platforms such as BrowserStack and LambdaTest allow users to create in-Cloud device farms with the goal of test automation of web and mobile applications. Given that there is a need for platforms for fairly and comparatively evaluating Edge deep learning performance [4–6], this is the first platform of this kind, particularly aiming at providing a common testbed for studying scheduling of deep learning inference tasks under horizontal execution over mobile device clusters.

The scientific value of this experimentation platform is threefold, namely to allow researchers to (a) realistically characterize and compare smartphone hardware capabilities for executing deep learning codes over arbitrary data streams using multicore microprocessors and GPUs, (b) to experiment with different cluster settings and task scheduling criteria, and (c) to gather smartphone profile data that might be in turn employed to feed back our existing Dew simulators, thus creating a virtuous circle in deriving task schedulers [7]. As we support arbitrary streams and Tensorflow models, in practice, derived knowledge using our platform might impact many disciplines where Dew computing is the killer computing paradigm.

2. Software description

2.1. Software architecture

From an architectural standpoint, the platform is a client–server software system, complemented with a hardware device called Motrol (see Fig. 1(a)). Clients are, on one hand, the mobile devices being exercised, which run a native Android 6+ application (Normapp). Another client is the *scnrunner* module, which parses Dew computing scenario parameters, builds corresponding data streams, derives deep learning-based jobs, decides which node executes which job, and submits jobs to the server and hence indirectly exploiting attached mobile devices.

The server works by assuming an energy managing device called Motrol for which Python-based drivers are provided. Via its support for dynamic energy supply switching, Motrol allows researchers to automatically repeat/reproduce job set executions involving several smartphones configured with a specific battery level. Currently, we support an USB-interfaced Arduino device called Motrol 1.0 [8], and a WiFi-enabled ESP8266-based microcontroller device called Motrol 1.5 (see Fig. 1(b)). A more complex prototype based on the Raspberry Pi 4 Model B, called Motrol 2.0 [9], is under development but it is still not considered in this paper. For example, this model will support USB-charging in addition to AC-charging, a feature missing. In terms of software design, emanager_server exposes several Rest APIs. Please refer to the project's documentation for detailed API specifications in the popular Swagger format (http://swagger.io).

Motrol 1.0 and Motrol 1.5 use electromechanical relays and provide low-level operations to control energy supply for attached mobile devices. We also provide a mock energy manager to operate the whole platform without Motrol, which prompts the user to manually plug/unplug a specific smartphone[1] from the power grid during test execution as needed. Naturally, this plug/unplug behavior, when using Motrol, remains automatic.

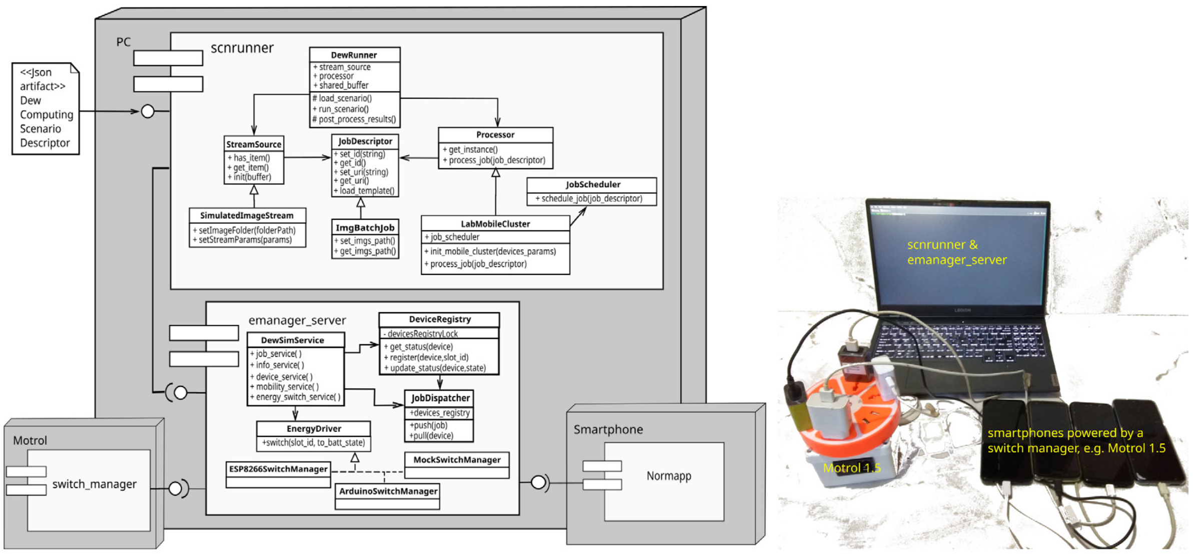
In respect to *scnrunner*, it is a subproject designed to encapsulate logic that facilitates stream-derived jobs modeling and its online execution using real Dew computing testbeds. Modeling stream-derived jobs consists in generating partitions from a continuous data flow, for example, frames captured within a time window, so its processing is treated by the computing infrastructure (e.g., mobile cluster) as an indivisible workload (atomic job) to be solved. By online execution we mean that once an atomic job is available, a scheduling mechanism is activated to delegate its execution to a smartphone within the Dew cluster. For now, jobs perform object recognition and classification operations with deep learning over images. By extending the *stream.py* module, i.e., providing custom implementations to the *has_items* and *get_item* methods of the *StreamSource* class, new stream types can be supported, e.g., audio streams and text streams.

In conjunction with *emanager_server*, *scnrunner* eases the reproduction of experimental settings representing Dew computing scenarios to study relevant metrics (e.g. throughput, battery utilization, latency) as a result of using different scheduling criteria under real Dew computing environments. An experimental setting is mainly characterized by a workload generation and node states reset. *scnrunner* provides a `run.sh` script that is the entry point to execute a given set of Dew computing scenarios. The script receives a path to a directory containing Dew computing scenarios descriptor files as the only required argument.

2.2. Software functionalities

Once installed in a PC, the server must be started, by optionally indicating the total number of mobile devices to be employed in the benchmark session. By default, this parameter is the maximum number of smartphones (connection slots) supported by the configured energy device (please see `src/emanager_server/serverConfig.json`).

[1] Under this operation mode, smartphones must be connected directly to the power grid



(a) Deployment view   (b) Dew computing in-lab testbed example

Fig. 1. Architectural view of LiveDewStream. The diagram depicts the UML deployment view, and within each node, the UML components and classes modeling the various entities.

The server will then initialize each device involved in the session, by asking the user to plug, one by one, each device via USB to the PC. This allows the server to gain root access to the device to (a) pushing essential configuration such as server IP address and port, and mostly (b) copy and run shell scripts directly on the device that are needed for benchmarks to correctly operate, and (c) remotely install (or update) and run our Android application on the device. It is worth noting that tasks (a) and (b) are performed via the ADB (Android Debug Bridge) tool of the Android SDK, which allows a PC to remotely send commands to a daemon which runs on a mobile device. On the other hand, task (c) is done by using ADB in conjunction with Monkey, a program that emulates streams of mobile user events such as clicks, touches, or gestures, as well as a number of system-level events. Via Monkey, we launch and (if configured to do so in `serverConfig.json`) start the application automatically. Once Normapp is running in the smartphones, the user unplugs each mobile device from the PC and plugs it using its original charger to the configured energy supply hardware (i.e. Motrol 1.0 or Motrol 1.5) or to wall sockets (i.e. Mock). Normapp will periodically poll the server via the DeviceService API for jobs to execute, and submit back the results. Job creation and result summarization is done by *scnrunner*.

2.3. Sample code snippets

Algorithm 1 shows `src/scnrunner/dew_runner.py`, which has the main method of scnrunner. In lines 1–9 there is logic to parse the supplied Dew computing scenario JSON file and to create the corresponding stream and processor objects. At line 8, a shared queue is created (and passed as argument to the `build_stream` factory method) to enable stream and processor entities to communicate using a producer–consumer pattern. The Stream entity, which produces items, runs in a separate thread started upon invocation of `stream.yield_items()` at line 11. Processor entity runs in the main thread consuming items from the shared queue (line 15).

3. Illustrative example

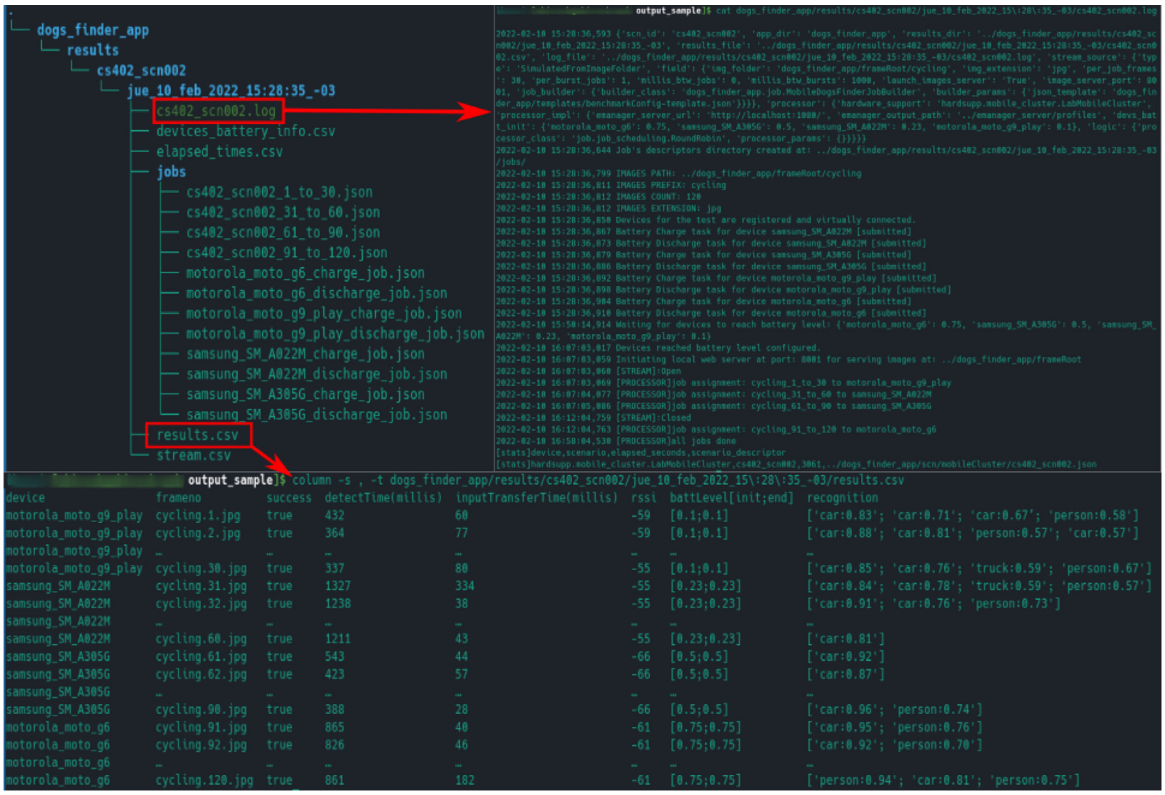
In the GitHub repository, we provide a sample scenario configuration file (`/doc/cs402_scn002.json`). The `src/scnrunner/run.sh` script accepts as an argument a directory path with at least one of these JSON formatted files. The `scn_id` field identifies the scenario. In the file, this field used to name the results directory and an output log file, all created by the platform and where it stores scenario execution related information. The JSON file also contains configuration parameters of the stream entity mentioned in Section 2.3. For instance, `img_folder` indicates the path where input images are located within the filesystem, `per_job_frames`, `per_burst_jobs`, `millis_btw_jobs` and `millis_btw_bursts` are parameters used to shape the stream speed and load introduced to the system. In the provided example, jobs are composed of 30 consecutive frames and these jobs are generated every one second (`millis_btw_bursts`). Job generation finishes when consuming all images of `img_folder`. Images are served to Normapp through a simple HTTP server which can be accessed through the specified port. The configuration also indicates the Python class loaded by the platform to create the next job item every time the `get_item()` method is invoked.

The processor key is where parameters of the processor entity are configured. The `hardsupp.mobile_cluster.LabMobileCluster` class is loaded by the platform to perform nodes initialization and jobs scheduling. The class uses the emanager_server Rest API. Nodes in this case are four smartphones whose model names and initial battery level are indicated under `devs_batt_init`. Battery level is a two-decimal number in the range [0-1]. This is, in the configuration, battery level is a number with at most two decimals, being 0 equals to 0% of battery (in the mobile device). The same applies to battery level equals to 1 (it represents 100%). Other examples are 0.04, 0.55 and 0.85 (meaning 4%, 55% and 85%). Besides, "-1" is interpreted as "any battery level". Desired scheduling logic is configured by providing the path using dot notation to the class implementing a processor logic, in this case, `job.job_scheduling.RoundRobin`. Paths are relative to where `dew_runner.py` is within the project.

Fig. 2 shows, on the top left, a sample directory tree of files generated by the platform after running a test. The scn_id

Algorithm 1: Parsing/creating the stream and processor entities declared in scenario descriptor file.

```
if __name__ == '__main__':

    args = parser.parse_args()
    with open(args.scenarioDescriptor, "r") as scnfile:
        scn_data = json.load(scnfile)

    processor = pb.build_processor(scn_data["processor"])
    jobs_queue = queue.Queue(0)
    stream = sb.build_stream(scn_data["scn_id"], scn_data["stream_source"], jobs_queue)

    stream.yield_items()
    test_end = False
   while not test_end:
        try:
           processor.process_job(jobs_queue.get())
        except queue.Empty:
            if stream.is_closed() and processor.all_jobs_completed():
                test_end = True
```



```
.
└── dogs_finder_app
    └── results
        └── cs402_scn002
            └── jue_10_feb_2022_15:28:35_-03
                ├── cs402_scn002.log
                ├── devices_battery_info.csv
                ├── elapsed_times.csv
                ├── jobs
                │   ├── cs402_scn002_1_to_30.json
                │   ├── cs402_scn002_31_to_60.json
                │   ├── cs402_scn002_61_to_90.json
                │   ├── cs402_scn002_91_to_120.json
                │   ├── motorola_moto_g6_charge_job.json
                │   ├── motorola_moto_g6_discharge_job.json
                │   ├── motorola_moto_g9_play_charge_job.json
                │   ├── motorola_moto_g9_play_discharge_job.json
                │   ├── samsung_SM_A022M_charge_job.json
                │   ├── samsung_SM_A022M_discharge_job.json
                │   ├── samsung_SM_A305G_charge_job.json
                │   └── samsung_SM_A305G_discharge_job.json
                ├── results.csv
                └── stream.csv
```

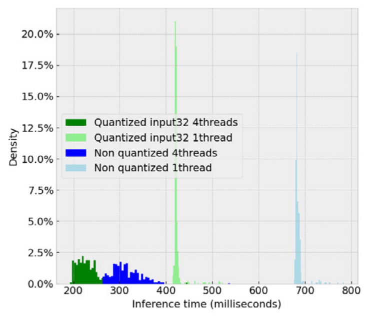
```
[redacted] output_sample]$ cat dogs_finder_app/results/cs402_scn002/jue_10_feb_2022_15\:28\:35_-03/cs402_scn002.log

2022-02-10 15:28:36,593 {'scn_id': 'cs402_scn002', 'app_dir': 'dogs_finder_app', 'results_dir': '../dogs_finder_app/results/cs402_sc
n002/jue_10_feb_2022_15:28:35_-03', 'results_file': '../dogs_finder_app/results/cs402_scn002/jue_10_feb_2022_15:28:35_-03/cs402_scn0
02.csv', 'log_file': '../dogs_finder_app/results/cs402_scn002/jue_10_feb_2022_15:28:35_-03/cs402_scn002.log', 'stream_source': {'typ
e': 'SimulatedFromImageFolder', 'field': {'img_folder': 'dogs_finder_app/frameRoot/cycling', 'img_extension': 'jpg', 'per_job_frames
': 30, 'per_burst_jobs': 1, 'millis_btw_jobs': 0, 'millis_btw_bursts': 1000, 'launch_images_server': 'True', 'image_server_port': 80
01, 'job_builder': {'builder_class': 'dogs_finder_app.job.MobileDogsFinderJobBuilder', 'builder_params': {'json_template': 'dogs_fin
der_app/templates/benchmarkConfig-template.json'}}}}, 'processor': {'hardware_support': 'hardsupp.mobile_cluster.LabMobileCluster',
'processor_impl': {'emanager_server_url': 'http://localhost:1080/', 'emanager_output_path': '../emanager_server/profiles', 'devs_bat
t_init': {'motorola_moto_g6': 0.75, 'samsung_SM_A305G': 0.5, 'samsung_SM_A022M': 0.23, 'motorola_moto_g9_play': 0.1}, 'logic': {'pro
cessor_class': 'job.job_scheduling.RoundRobin', 'processor_params': {}}}}}
2022-02-10 15:28:36,644 Job's descriptors directory created at: ../dogs_finder_app/results/cs402_scn002/jue_10_feb_2022_15:28:35_-03
/jobs/
2022-02-10 15:28:36,799 IMAGES PATH: ../dogs_finder_app/frameRoot/cycling
2022-02-10 15:28:36,811 IMAGES PREFIX: cycling
2022-02-10 15:28:36,812 IMAGES COUNT: 120
2022-02-10 15:28:36,812 IMAGES EXTENSION: jpg
2022-02-10 15:28:36,850 Devices for the test are registered and virtually connected.
2022-02-10 15:28:36,867 Battery Charge task for device samsung_SM_A022M [submitted]
2022-02-10 15:28:36,873 Battery Discharge task for device samsung_SM_A022M [submitted]
2022-02-10 15:28:36,879 Battery Charge task for device samsung_SM_A305G [submitted]
2022-02-10 15:28:36,886 Battery Discharge task for device samsung_SM_A305G [submitted]
2022-02-10 15:28:36,892 Battery Charge task for device motorola_moto_g9_play [submitted]
2022-02-10 15:28:36,898 Battery Discharge task for device motorola_moto_g9_play [submitted]
2022-02-10 15:28:36,904 Battery Charge task for device motorola_moto_g6 [submitted]
2022-02-10 15:28:36,910 Battery Discharge task for device motorola_moto_g6 [submitted]
2022-02-10 15:50:14,914 Waiting for devices to reach battery level: {'motorola_moto_g6': 0.75, 'samsung_SM_A305G': 0.5, 'samsung_SM_
A022M': 0.23, 'motorola_moto_g9_play': 0.1}
2022-02-10 16:07:03,017 Devices reached battery level configured.
2022-02-10 16:07:03,059 Initiating local web server at port: 8001 for serving images at: ../dogs_finder_app/frameRoot
2022-02-10 16:07:03,068 [STREAM]:Open
2022-02-10 16:07:03,069 [PROCESSOR]job assignment: cycling_1_to_30 to motorola_moto_g9_play
2022-02-10 16:07:04,077 [PROCESSOR]job assignment: cycling_31_to_60 to samsung_SM_A022M
2022-02-10 16:07:05,086 [PROCESSOR]job assignment: cycling_61_to_90 to samsung_SM_A305G
2022-02-10 16:12:04,759 [STREAM]:Closed
2022-02-10 16:12:04,763 [PROCESSOR]job assignment: cycling_91_to_120 to motorola_moto_g6
2022-02-10 16:58:04,530 [PROCESSOR]all jobs done
[stats]device,scenario,elapsed_seconds,scenario_descriptor
[stats]hardsupp.mobile_cluster.LabMobileCluster,cs402_scn002,3061,../dogs_finder_app/scn/mobileCluster/cs402_scn002.json
```

```
[redacted] output_sample]$ column -s , -t dogs_finder_app/results/cs402_scn002/jue_10_feb_2022_15\:28\:35_-03/results.csv
device                frameno          success  detectTime(millis)  inputTransferTime(millis)  rssi  battLevel[init;end]  recognition
motorola_moto_g9_play cycling.1.jpg    true     432                 60                         -59   [0.1;0.1]            ['car:0.83'; 'car:0.71'; 'car:0.67'; 'person:0.58']
motorola_moto_g9_play cycling.2.jpg    true     364                 77                         -59   [0.1;0.1]            ['car:0.88'; 'car:0.81'; 'person:0.57'; 'car:0.57']
motorola_moto_g9_play …                …        …                   …                          …     …                    …
motorola_moto_g9_play cycling.30.jpg   true     337                 80                         -55   [0.1;0.1]            ['car:0.85'; 'car:0.76'; 'truck:0.59'; 'person:0.67']
samsung_SM_A022M      cycling.31.jpg   true     1327                334                        -55   [0.23;0.23]          ['car:0.84'; 'car:0.78'; 'truck:0.59'; 'person:0.57']
samsung_SM_A022M      cycling.32.jpg   true     1238                38                         -55   [0.23;0.23]          ['car:0.91'; 'car:0.76'; 'person:0.73']
samsung_SM_A022M      …                …        …                   …                          …     …                    …
samsung_SM_A022M      cycling.60.jpg   true     1211                43                         -55   [0.23;0.23]          ['car:0.81']
samsung_SM_A305G      cycling.61.jpg   true     543                 44                         -66   [0.5;0.5]            ['car:0.92']
samsung_SM_A305G      cycling.62.jpg   true     423                 57                         -66   [0.5;0.5]            ['car:0.87']
samsung_SM_A305G      …                …        …                   …                          …     …                    …
samsung_SM_A305G      cycling.90.jpg   true     388                 28                         -66   [0.5;0.5]            ['car:0.96'; 'person:0.74']
motorola_moto_g6      cycling.91.jpg   true     865                 40                         -61   [0.75;0.75]          ['car:0.95'; 'person:0.76']
motorola_moto_g6      cycling.92.jpg   true     826                 46                         -61   [0.75;0.75]          ['car:0.92'; 'person:0.70']
motorola_moto_g6      …                …        …                   …                          …     …                    …
motorola_moto_g6      cycling.120.jpg  true     861                 182                        -61   [0.75;0.75]          ['person:0.94'; 'car:0.81'; 'person:0.75']
```

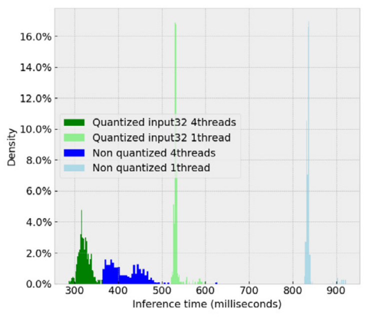
Fig. 2. Output sample: Directory tree generated, events log and results file.

specified in the scenario file and the system date are used to identify and create a directory (dogs_finder_app/results/-cs402_scn002) where to locate all output files generated by a test. Such directory includes a plain text log containing chronologically-ordered events (see Fig. 2 top right). Depending on which entity is logging information, entries of the event log are tagged with [STREAM] or [PROCESSOR]. There is also scenario descriptor file information to identify which log corresponds to which input parameters. Fig. 2 bottom part shows a sample results.csv file, which is built by merging individual results sent by mobile devices to the server. It tells which images were sent to which device, how much time the device spent on downloading data and performing the inference over the image. It also records RSSI indicator, battery level at the time of downloading images and finishing inferences, and the output of the recognition algorithm itself.

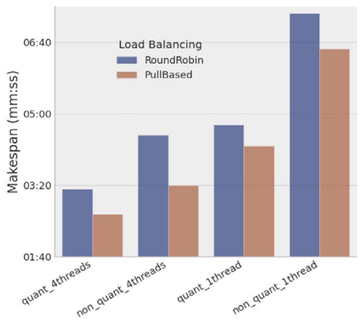
To illustrate some of the software output that allow researchers to study the feasibility of Dew computing scenarios, next we include running examples of several Dew computing scenario configuration files (dogs_finder_app/scn/). These aim at studying the performance of different versions of the same Tensorflow Lite model on two mobile devices using different tasks load balancing mechanisms. The resulting plots were obtained by processing the results.csv file of each Dew computing scenario. Figs. 3(a) and 3(b) show the inference times of different versions of the same Tensorflow Lite model (YOLOv4) when running on a Xiaomi Redmi Note 7 and a Samsung A30 smartphone, respectively. We can see that the lowest inference time is achieved using quantization and multithreading. However, multithreading produces more dispersed times than single threading model execution. Fig. 3(c) shows the performance of all model-thread combinations for a Dew computing scenario



(a) Inference time on Xiaomi RN 7



(b) Inference time on Samsung A30



(c) Load Balancing performance

Fig. 3. YOLOv4 model performance.

where two smartphones cooperate in making inferences over a stream of 1000 images given at a rate of 30 FPS. Consistently with previous results, the quantized version of YOLOv4 using multithreading achieves the lowest makespan. Moreover, a pull-based load balancing mechanism beats Round Robin irrespective of the model version.

4. Impact and closing remarks

Due to the difficulty and time-consuming nature of experimentation in the Distributed Computing research community, simulation is an accepted practice. This is demonstrated by the heavy adoption of simulators for Cloud and Fog computing in the literature. Prominent examples are CloudSim (2011) [10] and iFogSim (2017) [11], whose seminal papers have reached around 5,500 and 1,200 citations respectively according to Google Scholar. The software projects behind these simulators have also made an impact in the abovementioned research community. For instance, CloudSim has generated dozens of complementary projects (e.g. WorkflowCloudSim) and its GitHub repo (github.com/Cloudslab/cloudsim) has over 400 forks. In Dew computing, simulation is also a practiced evaluation methodology for studying mobile resource scavenging heuristics [7]. Then, we have proposed DewSim (2020) [3], a trace-based simulation software that uses battery traces to map mobile devices resource utilization levels (e.g. microprocessor) with battery behavior. Despite having many miles ahead to travel yet, DewSim plus its supporting platform [8] has already shown impact in the scientific literature, as it is the support for many Dew schedulers [7,12], which have been published in reputed journals.

Generally speaking, simulations assume that all the involved elemental processes are known and correct, only then built simulators are fully trustworthy. In practice, however, the relentless developing nature of Cloud, Fog and Dew gradually introduce new fundamental knowledge that challenge the assumptions made by built simulators. This means (a) new versions of specific software modules have to be built, or (b) integral software modifications must be carried out in order to comply to certain validation experiments. For example, a trace-based file transfer model has been proposed for CloudSim [13], and the sub-optimal accuracy of the simulator as a whole when modeling resource heterogeneity has been evidenced [14].

The software proposed in this paper provides a testbed for deriving such fundamental knowledge, so DewSim can be correctly evolved, and proper validation experiments can be performed. Not to mention that, as the platform targets deep learning codes over data streams on mobile devices clusters, this specific yet general-enough application scenario, where Dew computing seems to be the killer paradigm, might increment our current user base.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

We thank Eng. Pablo Sanabria (https://www.linkedin.com/in/tailmon/?originalSubdomain=cl) from Chile for his assistance regarding running Tensorflow Lite models on Android. We also acknowledge the financial support by CONICET (grant no. 11220170100490CO) and ANPCyT(grant no. PICT-2018-03323).

References

[1] Bonomi Flavio, Milito Rodolfo, Zhu Jiang, Addepalli Sateesh. Fog computing and its role in the internet of things. In: Proceedings of the first edition of the MCC workshop on mobile cloud computing. 2012, p. 13–6.

[2] Aazam Mohammad, Zeadally Sherali, Harras Khaled A. Offloading in fog computing for iot: Review, enabling technologies, and research opportunities. Future Gener Comput Syst 2018;87:278–89.

[3] Hirsch Matías, Mateos Cristian, Rodriguez Juan Manuel, Zunino Alejandro. Dewsim: A trace-driven toolkit for simulating mobile device clusters in dew computing environments. Softw - Pract Exp 2020;50(5):688–718.

[4] Wang Xiaofei, Han Yiwen, Leung Victor CM, Niyato Dusit, Yan Xueqiang, Chen Xu. Convergence of edge computing and deep learning: A comprehensive survey. IEEE Commun Surv Tutor 2020;22(2):869–904.

[5] Chen Jiasi, Ran Xukan. Deep learning with edge computing: A review. Proc IEEE 2019;107(8):1655–74.

[6] Murshed MG Sarwar, Murphy Christopher, Hou Daqing, Khan Nazar, Ananthanarayanan Ganesh, Hussain Faraz. Machine learning at the network edge: A survey. ACM Comput Surv 2021;54(8):1–37.

[7] Hirsch Matías, Mateos Cristian, Zunino Alejandro, Majchrzak Tim A, Grønli Tor-Morten, Kaindl Hermann. A task execution scheme for dew computing with state-of-the-art smartphones. Electronics 2021;10(16):2006.

[8] Hirsch Matías, Mateos Cristian, Zunino Alejandro, Toloza Juan. A platform for automating battery-driven batch benchmarking and profiling of android-based mobile devices. Simul Model Pract Theory 2021;109:102266.

[9] Mateos Cristian, Hirsch Matías, Toloza Juan, Zunino Alejandro. Motrol 2.0: A dew-oriented hardware/software platform for batch-benchmarking smartphones. In: 2021 IEEE 45th annual computers, software, and applications conference. IEEE; 2021, p. 1772–7.

[10] Calheiros Rodrigo N, Ranjan Rajiv, Beloglazov Anton, Rose César AF De, Buyya Rajkumar. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw - Pract Exp 2011;41(1):23–50.

[11] Gupta Harshit, Dastjerdi Amir Vahid, Ghosh Soumya K, Buyya Rajkumar. Ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. Softw Pract Exp 2017;47(9):1275–96.

[12] Hirsch Matías, Rodríguez Juan Manuel, Mateos Cristian, Zunino Alejandro. A two-phase energy-aware scheduling approach for cpu-intensive jobs in mobile grids. J Grid Comput 2017;15(1):55–80.

[13] Chai Anchen, Bazm Mohammad-Mahdi, Camarasu-Pop Sorina, Glatard Tristan, Benoit-Cattin Hugues, Suter Frédéric. Modeling distributed platforms from application traces for realistic file transfer simulation. In: 2017 17th IEEE/ACM international symposium on cluster, cloud and grid computing. IEEE; 2017, p. 54–63.

[14] Zakarya Muhammad, Gillam Lee. Modelling resource heterogeneities in cloud simulations and quantifying their accuracy. Simul Model Pract Theory 2019;94:43–65.