Towards Testbed as-a-Service: design and implementation of an unattended SoC cluster

Gabriele Proietti Mattia, Roberto Beraldi
Department of Computer, Control and Management Engineering "Antonio Ruberti",
Sapienza University of Rome,
Email: proiettimattia@diag.uniroma1.it, beraldi@diag.uniroma1.it

***Abstract*—The current computing power of single-board computers (SBCs) is relevant, and if this factor is associated with the very low cost of installing and operating such devices, building a cluster is a natural consequence. Very often in fog computing, researchers need to run and test their distributed solutions and algorithms in real hardware and software, rather than by simulations and doing this in a cluster of SBCs is feasible and inexpensive. This paper addresses most of the problems and issues that arise when building a self-contained, remote controllable and unattended cluster of Raspberry Pi that minimizes the physical intervention of a human operator, which enables the notion of *Testbed as-a-Service*. The solution envisioned here is to set up the cluster in a desktop computer case, which needs addressing power management and to allow remote configuration of experiments. Moreover, the paper proposes several guidelines for installing a suitable operating system and software for running any kind of distributed application.**

***Index Terms*—testbed, Raspberry Pi, single board computer, cluster, fog computing.**

I. Introduction

A crucial aspect of designing distributed algorithms for fog or edge computing is to provide results that efficiently run, aside from a simulation, in a real setup with real hardware and software. Most of the time trying to set up such kind of testbed could appear costly and time-consuming, but it is not always the case. Indeed, thanks to the current hardware technologies, there are many types of single-board computers (SBCs) which have a non-negligible computing power and they are available at a very low cost, like for example Raspberry Pi[1]. Building a cluster with this kind of device is reasonable but if we want to build a stable, long-term, and stand-alone solution there are different issues that must be addressed: (i) decide a proper enclosure or chassis which will hold all the components of the cluster, and this must bring with it a suitable solution for managing the power supply of the entire system since it can be unfeasible to have a single power adapter for every single-board PC; (ii) the entire cluster should be managed, under the hardware point of view, like a server rack component which can be easily added or removed for example from a server cabinet; (iii) the ability to power on and off the cluster remotely, and this feature introduces an entire new set of problems that regards software management; (iv) the system is to be unattended or there is a low probability of the intervention of an operator who manually has to remove the single-board PCs from the cluster enclosure and reinstall the operating system, indeed in this kind of low power devices, the software resides on external storage, like a microSD, therefore installing the operating system requires the software to be flashed in the storage support. All these needs are the prerequisite to set up a general cluster that can be used remotely to run experiments, i.e., a concept that we dubbed *Testbed as-a-Service*.

This paper is the result of an experience done to realize the envisioned cluster by using Raspberry Pi SBCs. The main contributions of this work can be summarized as follows:

- delineation of hardware and software requirements for a long-term, unattended and remote controllable solution for implementing a Raspberry Pi cluster;
- design of a power supply board for using a desktop computer power supply (called ATX) for powering up to eight Raspberry Pi boards;
- design of a remote Ethernet switch system for remote controlling the power of the cluster to be associated with the power supply board;
- propose a way to define the testbed configuration and an experiment via JSON configuration files, towards a Testbed-as-a-Service paradigm;
- show the results of the benchmark of a distributed scheduling algorithm installed in the cluster.

The paper is structured as follows. In Section II is presented a summary of other works in literature that involve the design and the implementation of a SBCs cluster, especially regarding Raspberry Pi boards, Section III shows the hardware design of the cluster with a focus on the power supply system. Instead, Section IV presents the software management issues and a proposal of a testbed configuration file architecture. Finally, in Section V we will show the results of a distributed scheduling algorithm benchmark and we will draw conclusions in Section VI.

II. Related Work

There are different approaches in literature for building a Raspberry Pi cluster, indeed the low realization and operational costs make it usable for learning parallel computing [1], [2], run distributed algorithms [3], [4] or just for studying the energy and the computation power system [5], [6], [7].

In particular, [1] and [2] describe a Raspberry Pi cluster that is built for learning purposes. The devices are arranged

[1]https://www.raspberrypi.org/

in a cart and they make use of the MPI messaging protocol for the intercommunication. However, this solution is not self-contained and it does not provide a practical solution to the power management.

Works [8] and [3] show an implementation of a cluster of Raspberry Pi that is assembled with Lego bricks, the former also introduces a software management system called PiCloud and the latter uses the cluster for running a tourism data aggregation application. In both cases, a self-enclosing and an unattended design and structure is not considered.

"Iridis-Pi" is a cluster of 64 Raspberry Pi presented in [9]. The authors performs a benchmark of the cluster trying to derive the total computing power of the entire architecture, but for doing that different issues are addressed, like the power supply, the network capability and a shared and distributed storage.

A series of challenges when building a Raspberry Pi cluster is listed in [10]. The paper, after describing the design and the setup of the cluster, performs a series of benchmarks regarding the total computational power of the system.

Concluding, [5] studies a Raspberry Pi cluster as a high-performance computing (HPC) cluster, considering the computing and the electric power the work provides performance results regarding the number of cores and the number of computing nodes in a cluster.

III. Hardware

A. Enclosure

The first decision that is needed to take when building the cluster regards a suitable physical structure that is able to hold the essential components, namely the SBCs and the power supply unit. There are many solutions that can satisfy our needs but they are often offered at a very high cost, relatively to the cost of the boards. For this reason, in this work, we tried to re-use a desktop PC case and its power supply unit. Figure 1 shows a preliminary set up of the cluster enclosure. We can observe that the single Raspberry Pis are arranged in (black) cases which have a single screw on the side, then all of these are attached to a rigid plastic structure that fits the standard holes of a Standard-ATX motherboard, which are specified by Intel [11]. The case has also been enriched with two fans for favouring the airflow.

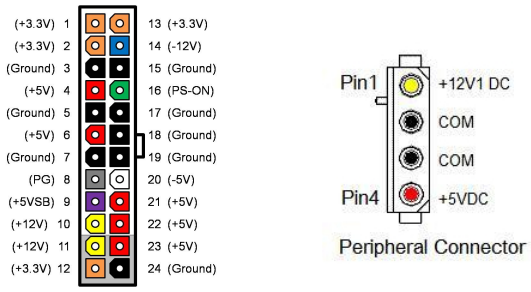
The case that has been used is a standard ATX case but it is not rack-able (i.e. it cannot be arranged in a server rack). For a better space management, there are also available rack-able ATX cases of 4 units that could be easily installed in a server cabinet. However, switching to this kind of case does not alter anything of the work presented here.

B. Power board design

The main issue of using a PC case as an enclosure is the usage of the ATX power supply unit, this because having eight power adapters for eight Raspberry Pis is not a feasible solution, especially in terms of space. A standard ATX power supply has different connectors that are usually attached to the motherboard and to all the peripherals. There are many

Fig. 1: Preliminary set up of the cluster

vendors and types (in our case we used a "Trustech TR-20787", whose specifications are listed in Table I) but in general all of the connectors are cascade arranged in at least in 4 lines, in the following way:



(a) ATX 24pin motherboard connector (b) 4pin peripheral connector

Fig. 2: Desktop PC power supply unit connectors

L1) this line has one 24-pin port (Figure 2a) for the motherboard power;
L2) this line has two 4-pin ports for powering the peripherals (Figure 2b) and two ports for powering SATA disks;
L3) same as L2);
L4) this line has one 4 (or 4+4) pins port powering the CPU.

All of these lines are made up of 20 AWG cables (which corresponds to 0.50mm^2 of section) and they have 20 cores. According to [12], in normal conditions, each of these cables can bring up to 3.5A, considering 5V that is the voltage needed for the Raspberry Pi. Moreover, a Raspberry Pi 4 needs 3A for operating correctly[2], this means that for powering up to 8 Raspberry Pi 4 we need 24A and 7 cables from the +5V rail of the PSU (whose colour is red) and at least 7 cables for the

[2]https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/

Trustech TR-20787						
AC Input	**Voltage** 230Vac		**Current** 6A Max		**Frequency** 50Hz	
Max DC Output	+3.3V 22A	+5V 28A	+12V 28A	-12V 0.8A	-5V 0.6A	+5VSB 2.5A

TABLE I: The specification label on the PSU (Power Supply Unit)

negative pole (whose colour is black). We can gather all the needed cables from the PSU connectors, in particular we can:

- pick 5 red +5V cables and 8 black negative cables from the 24pin connector;
- pick 2 red +5V cables and 2 black negative cables from the two lines in which we have the peripheral 4pin connector.

This configuration would allow powering up 8 Raspberry Pi since the PSU can support at maximum 28A on the +5V rail (Table I).

Figure 3 shows the circuit diagram of "ATX2RPi8", a printed circuit board (PCB) that we designed for powering eight Raspberry Pi using a desktop ATX power supply unit. The red tracks are printed at the front face of the board and the blue ones instead to the rear face. The diagram has been designed with EasyEDA[3] and then submitted to the factory for printing[4]. Aside from gathering the red and black cables as described, also considering the thickness of the tracks proportional to the load, the board also has the following features:

- it distributes the 5V+ current to eight clamps, to which will be attached the cables towards the Raspberry Pis. These cables have been assembled with a Type C connector;
- it offers two fan ports, one at 5V and one at 12V;
- it offers a clamp for the `PS_ON` rail that if connected to ground cause the switching on of the PSU. This clamp will be attached to an ethernet switch for remotely turning on and off the cluster;
- it offers a clamp for the +5VSB rail, that is a line always powered, even if the PSU is turned off. This line is used for powering an ethernet relay;
- it offers an additional clamp for 12V, for general purposes.

Table II shows the Bill of Materials (BOM) of the board, it can be used for ordering the components from the supplier[5]. The components are: the 24pin female connector, two fan ports, two 4pin peripheral power connectors and eleven generic bipole clamps. The PCB printing and the components cost for a single board, including the shipping fees, is about 10$.

[3]https://easyeda.com/
[4]https://jlcpcb.com/
[5]https://lcsc.com

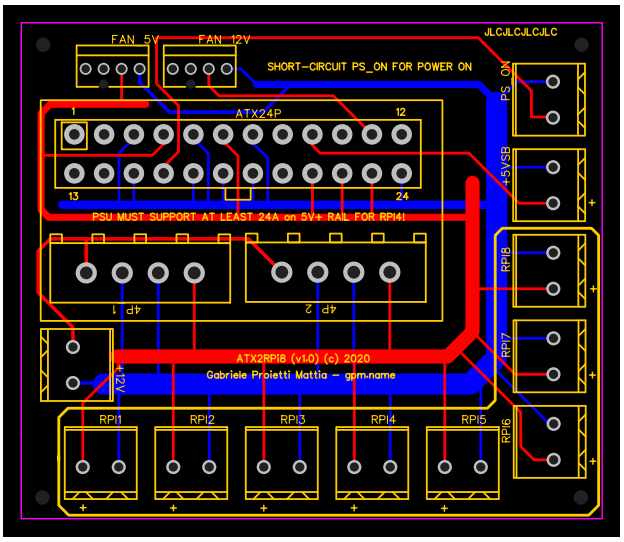


Fig. 3: Power supply board (ATX2RPi) (82x70mm)

C. Power control board

A relevant feature that an unattended cluster needs to have, especially if composed by Raspberry Pi, is the ability to remotely switching on and off the power in case there is the necessity of force restart the devices. This capability is included in the design of ATX2RPi8, but it must be completed with an ethernet or wireless relay which must be connected to the `+5VSB` port for being powered and to the `PS_ON` port for switching on and off the PSU.

In our set-up, we used a "HW-584 Web_Relay_Con V2.0" that is a relay control board and it can manage up to 16 channels. We only used one channel that has been connected to a high/low-level trigger relay, and the relay has been connected to the `PS_ON` port of the ATX2RPi8 board. The controller exposes a web dashboard from which we can switch on and off the relay and therefore the entire cluster.

D. Final Setup

Figure 4 shows the final configuration of the cluster that is currently operational. As we can observe, the cluster with eight Raspberry Pis 4B (with a quad-core Cortex-A72 CPU, 4GB of RAM and gigabit port) have been attached to a WiFi router (ASUS RT-AX88U, that supports up the 802.11ax standard) which allows experiments that make use of the wireless network, for example, smartphones or IoT. Then another two external Raspberry Pis 3B (with a quad-core Cortex-A53 CPU, 1GB of RAM and gigabit ethernet capped to 300Mbps due to the internal design of the RPi) have been added: the former is used as traffic generator which can simulate a background or noise traffic to the other SBCs, the latter is instead an entry point, since the cluster lives in a department network, it is necessary to have an SSH or VPN entry point from which we can have the control of all the components of the cluster, in this way we expose only a single node to the department's internal network and the cluster traffic remains confined in the subnetwork. In particular, this Raspberry Pi has been equipped

ID	Name	Designator	Quantity	Manufacturer Part	Manufacturer	Supplier	Supplier Part
1	CONN-TH_39281243	ATX24P	1	39281243	MOLEX	LCSC	C114088
2	CONN-TH_47053-1000	FAN_12V,FAN_5V	2	47053-1000	MOLEX	LCSC	C240840
3	CONN-TH_350211-1	4P_1,4P_2	2	350211-1	TE Connectivity	LCSC	C305826
4	CONN-TH_2P-P5.00_WJ500V-5.08-2P	RPI*	11	WJ500V-5.08-2P-14-00A	ReliaPro	LCSC	C8465

TABLE II: BOM for ATX2RPi(8) board

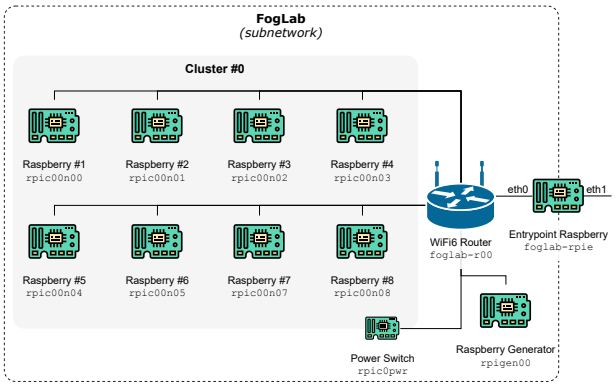


Fig. 4: Final set up of the cluster

with a USB ethernet (eth1), beyond the embedded ethernet port (eth0) in this way the RPi can be reachable from both the subnetworks.

IV. Software

A crucial feature that a cluster of Raspberry Pi should have, regards the ability of easily deploy and un-deploy software when it is needed. This process could be the easiest and cleanest as possible, we cannot allow an uncontrolled installation of libraries, dependencies and configurations. Therefore the use of some container management system is essential. We chose to install within each Raspberry Pi board a software distribution that already has included Docker and cloud-init that is a utility used in cloud contexts for auto-configuring nodes at boot. This distribution is open-source and it is called Hypriot OS[6]. But we did not use the software as it is, we adapted it to the following guidelines that we defined to make the cluster unattended.

1) in order to limit a human operator intervention we need to drastically reduce the possibility of OS corruption, for this reason the root partition should be read-only while all the persistent data (e.g. the Docker images and containers data) should be moved to a new writable partition;
2) the Raspberry Pi should reboot automatically if the system hangs or freezes (for example due to a kernel panic) therefore the watchdog kernel module (which is natively supported by hardware backend in RPi) must be enabled;
3) every node must auto-configure itself at boot, in particular every node must be reachable with SSH without manually typing username and password, this for facilitating any set-up or benchmarking script.

These code changes are published as open-source[7]. The final OS has been built and installed to all the Raspberry Pis and configured to have fixed IPs under a WiFi6 router which completed our cluster deployment.

A. Testbed-as-a-Service

For executing experiments, we installed in the cluster a FaaS scheduling framework called P2PFaaS [13] that allows implementing distributed scheduling algorithms for the FaaS job unit model. Since the framework is fully configurable via REST API, we can envision a JSON configuration file which sets up the testbed environment, like the needed nodes, the topology, the chosen scheduling algorithms and other parameters. This JSON, which can be accompanied by another configuration file which regards the specific experiment parameters, should be taken as input to a hypothetical master node which is in charge to actuate the passed configuration constraints.

Listing 1 shows the full description of a possible JSON file for describing the testbed environment. As we can see, the capabilities that are offered by the configuration regards:

1) the definition of the `infrastructure`, namely the number of nodes to start and their specific topology which is expressed in terms of neighbors nodes;
2) the configuration of the `scheduler` service envisioned as a Docker container; therefore we need to pass the address of the Docker image, the name of the scheduler that will be used (to choose among a set of schedulers implemented in the framework), the arguments of the scheduler and other basic parameters like the maximum number of parallel jobs that can be executed and the maximum job queue length;
3) the configuration of the `discovery` service, which is again a Docker container in charge of making nodes aware of their neighbors; here we could configure, for instance, the Docker image and the delay between the heartbeats;
4) the configuration of the `functions` that will be made available for testing, again envisioned as Docker containers; therefore we need to specify the Docker image address of the function, a name, the API address that will be call-able by clients and a set of specific fixed deployment arguments, if needed. The list of functions is expressed as a JSON array of JSON objects.

Our cluster is implementing a Fog environment, therefore we emphasize that the scheduler and the discovery service (and therefore the P2PFaaS framework) with the specified functions

[6]https://blog.hypriot.com/downloads/

[7]https://github.com/rpi-cluster

Listing 1 Testbed JSON configuration file

```
{
  "infrastructure": {
    "nr_nodes": "3",
    "topology": {
      "0": ["1","2"],
      "1": ["0","2"],
      "2": ["0","1"]
    }
  },
  "scheduler": {
    "image": "https://...",
    "name": "SchedulerIdentificator",
    "args": ["arg1", "arg2", "arg3" ],
    "max_parallel_jobs": 4,
    "max_queue_length": 2,
  },
  "discovery": {
    "image": "https://...",
    "heartbeat": "30s"
  },
  "functions": [
    {
      "name": "My Service",
      "api": "my_service",
      "image": "https://...",
      "args": ["arg1", "arg2", "arg3"]
    },
    {
      "name": "My Service #2",
      "api": "my_service_2",
      "image": "https://...",
      "args": ["arg1", "arg2", "arg3"]
    },
  ]
}
```

will be spawned in every node, since every node will be able to execute that functions by calling the respective API addresses and to schedule the execution in other neighbors nodes.

Listing 2 represents the configuration file for running an experiment. As in the previous case, this file should be passed to a hypothetical master node that is in charge of executing parallel flows of REST API calls to all the nodes in the cluster. The configuration file should allow to properly set:

- the `api` address of the function to test (that has previously configured in Listing 1);
- the `payload` path to associate to every REST API call;
- the path (`log_path`) to the log directory where the test results can be collected;
- the job `arrivals` configuration, that comprehends the rate (requests/s, also referred as λ) to every specific node and the distribution according to which the requests will be generated, for example as a Poisson distribution; for the sake of simplicity, we assume a fixed distribution but we could also envision to use here a distribution that comes from real user traffic, properly defined in a plain text file;
- the total number of request after that the experiment can stop, with field `max_num_requests`.

By having defined both the configuration files for the testbed itself and the experiment to carry out, we envision that the testbed usage can follow a Testbed-as-a-Service paradigm.

Listing 2 Testbed experiment JSON configuration file

```
{
  "api": "/my_service",
  "payload": "/path/to/payload",
  "log_path": "/path/to/log",
  "arrivals": {
    "distribution": "poisson",
    "rate": 1.0,
    "rates": {
      "0": 1.0,
      "1": 2.0,
      "2": 1.5,
    },
    "max_num_request": 2000,
  },
}
```

V. Experiments and Results

A. Experiments

We performed two experiments by using the presented cluster:

1) the first experiment has been run configuring the testbed to use a scheduler that requested no cooperation between nodes: upon a job arrival the job is always executed locally in the node, if there are available resources, otherwise it is rejected. Then we chose an arrivals scheme with no distribution behind, thus with fixed jobs inter-arrival time. This has been done for understanding the computational power of a single node in the cluster and to properly choose a reasonable arrival rate for the next experiment;
2) the second experiment is based on a power-of random choice cooperation algorithm for scheduling (called `PowerOfN` within the framework), in particular, the one presented in [13]. We executed a benchmark for eight different payload sizes.

For both the experiments, we tested the same function, namely a face detection task based on the PiCo algorithm[8]. The same image has been used for every job request: a 640x480 JPG with exactly four faces.

B. Results

1) Experiment 1: With this experiment, we collected the response time of 2000 requests sent in series. We obtained an average of $0.696s$ ($\sigma = 0.01848$) and the distribution depicted in Figure 5. This means that the service rate μ for a single node in the cluster is about 1.43 jobs/s, but this must be multiplied by the four processing cores of the CPU, therefore we estimated a total service rate of $1.43 \times 4 = 5.72$ jobs/s. This value can be considered as a good estimation because we assume that the kernel scheduler is fair and therefore four parallel jobs will be approximately scheduled to four different cores for the most of the execution time. This is also the reason why we set the P2PFaaS framework to be allowed to execute only four jobs in parallel ($K = 4$) in each node and in each benchmark.

[8]https://github.com/esimov/pigo

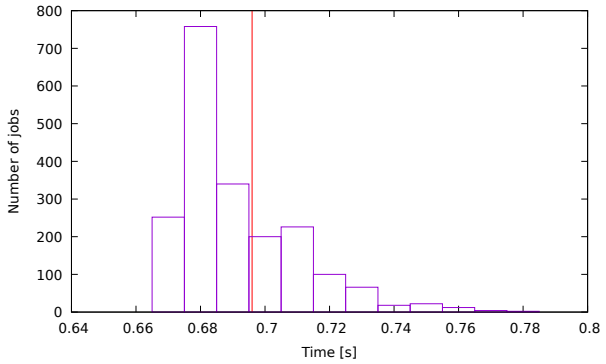


Fig. 5: Job duration distribution for a single Raspberry Pi 4B with no cooperation scheme

2) Experiment 2: Listing 3 shows the testbed configuration file for this experiment. As we can observe, the topology has been declared as a `fullyConnected` graph, and we set the `PowerOfN` scheduler which takes as input: (i) the fanout, that is the number of random probed nodes, set to one; (ii) the threshold (Θ), that is the limit above which cooperation is started, set to two; (iii) if the job that cannot be executed is discarded and not put in a waiting queue, set to true; (iv) the maximum number of hops that job can perform before being executed, set to one. Then we also set the maximum number of parallel jobs to four, as four is the number of cores of the Raspberry Pi 4B, and we set to use no additional queue, since `max_queue_length` is set to zero. As far as regard the function, the parameters `input_mode` and `output_mode` that refers to the input and output type of the function are set to `image`, therefore the input will be a binary file representing a JPG image and the output a binary file that is the same image with the highlighted faces.

Metrics collected and analyzed during the benchmarks are of two types. The first set of metrics regards the jobs' execution, and it has been collected by using the data reported by P2PFaaS. We have:

- *drop rate* (P_B), the percentage of jobs that have been rejected because they find the node to which they have been assigned at full load ($K = 4$);
- *total delay* (d_t), the total elapsed time for completing the request as seen by the client, in our case the Raspberry Pi Generator;
- *probing delay* (d_p), the total elapsed time for asking to another node its current load: it comprises the time for transmitting the request and for receiving the response;
- *forwarding delay* (d_f), that is the total time required for transmitting the job (only a few bytes of metadata) and its payload to another node;
- τ_e that is the time between the decision to forward a job and the effective job arrival in the remote node and it is defined as $d_f + d_p/2$.

The second set of metrics regards the nodes operating system and have been collected using Telegraf[9] and InfluxDB[10]. They comprise:

- *network activity*, the bytes received and sent by the network adapter every second;
- *CPU load*, the CPU time used by the system, the user and for serving the interrupt requests (Soft IRQs);
- *system load*, the average load of the system as reported by Linux in the last 1, 5 and 15 minutes.

Listing 3 Testbed JSON configuration file for Experiment 2.

```
{
  "infrastructure": {
    "nr_nodes": "6",
    "topology": "fullyConnected"
  },
  "scheduler": {
    "image": "p2pfaas/scheduler",
    "name": "PowerOfN",
    "args": [1, 2, true, 1],
    "max_parallel_jobs": 4,
    "max_queue_length": 0
  },
  "discovery": {
    "image": "p2pfaas/discovery",
    "heartbeat": "30s"
  },
  "functions": [
    {
      "name": "Pigo Face Detector",
      "api": "pigo-face-detector",
      "image": "esimov/pigo-openfaas",
      "args": {
        "input_mode": "image",
        "output_mode": "image",
        "write_timeout": 100,
        "read_timeout": 100
      }
    }
  ]
}
```

In this experiment, we tested different payload sizes but we used the same image in order to not change the computational time required to detect the faces. We indeed appended spare bits at the end of the payload to reach desired payload sizes. The following results focus to nine different payloads, from 50kB ($\tau_e = 0.021$s), the original size of the image, to 800kB ($\tau_e = 0.135$s) the maximum payload supported by the Raspberry Pi Generator (we experimented that a payload greater than this limit saturates the network adapter queue). Each test for each payload involved 2000 requests sent with Poisson distribution and has been repeated five times. Confidence intervals that are shown have been computed as $\bar{X} \pm t_{\frac{\alpha}{2},n-1} \frac{S}{\sqrt{n}}$ (where $\bar{X}$ is the sample mean, S the sample variance, and t the Student-t distribution) with $\alpha = 0.05$. All the sample means of the experiments are reported in table III. Listing 4 shows the JSON configuration file for this experiment, notice that we need a configuration for each payload to use, for this reason the payload path is set as `payload-Xkb.jpg`.

[9]https://github.com/influxdata/telegraf
[10]https://github.com/influxdata/influxdb

Listing 4 Testbed experiment JSON configuration file for Experiment 2.

```
{
  "api": "/pigo-face-detector",
  "payload": "./payload-Xkb.jpg",
  "log_path": "./log",
  "arrivals": {
    "distribution": "poisson",
    "rate": 5.5,
    "max_num_requests": 2000,
  }
}
```

kB	P_B	d_t	d_f	d_p	τ_e
50	0.5517	0.7545	0.0174	0.0073	0.0211
100	0.5572	0.7751	0.0284	0.0075	0.0321
200	0.5736	0.8175	0.0455	0.0075	0.0492
300	0.5958	0.8740	0.0591	0.0078	0.0630
400	0.6046	0.9159	0.0744	0.0074	0.0781
500	0.6163	0.9486	0.0867	0.0072	0.0902
600	0.6344	1.0158	0.0998	0.0074	0.1035
700	0.6452	1.0712	0.1132	0.0070	0.1167
800	0.6553	1.1401	0.1314	0.0070	0.1349

TABLE III: Summary of experimental results ($\Theta = 2$, $\lambda = 5.50$) as a function of the payload size (kB), all times are expressed in seconds

Figures 6 and 7 show the average drop rate and the delay when the threshold $\Theta = 2$ and the job arrival rate $\lambda = 5.50$ jobs/s as a function of τ_e. We can observe that the increase of the job payload, and thus of the network delay that exists between the forwarding decision and the actual arrival of the job to the destination node, causes a twofold effect: (i) an increase of the percentage of jobs that are dropped and (ii) a growth of the total delay. In particular, when we deal with the original image of about 50kB the drop rate is 55.16%, the total delay is 754ms while $\tau_e = 21$ms; when raising the payload up to 800 kB we observed an additional 10% in the drop rate, an increase of the total delay to 1.14s and of τ_e to 135ms.

The probing delay remains independent from the job payload, and it remains steady to about 7ms, and this reasonable since the probing does not use any relevant data transmission.

The linear relationship between the drop rate and the delay is reasonable and it shows the impact of the uncorrelation between the moment in which the node takes the forwarding decision and the moment in which the job actually arrives in the remote node

Figures 8, 9, and 10 shows Raspberry Pi 4B node OS statistics for a single benchmark from payload 50kB to 800kB with 2000 requests for payload size and using $\Theta = 2$ and $\lambda = 5.50$ images/s. In all of these figures, the x-axis represents the time elapsed during the benchmark, and we can notice a periodic fall of the y-axis values. Indeed, when switching the payload size, we observe a 60s gap during which the system is idle: this is voluntarily done to let the OS free all the resources and also to insert a clear recognition mark for the beginning each different test in the chart. Analyzing these results, we can observe the behaviour of network activity when the payload

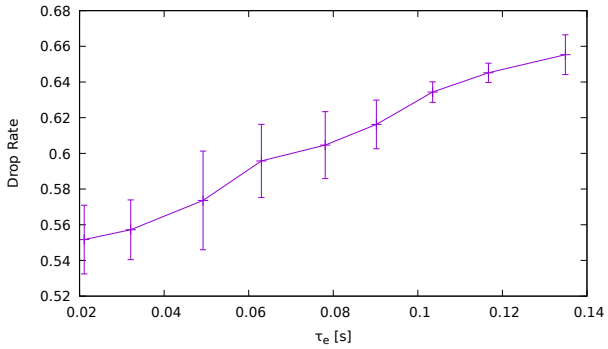


Fig. 6: Effect of τ_e on P_B ($\Theta = 2$, $\lambda = 5.50$)

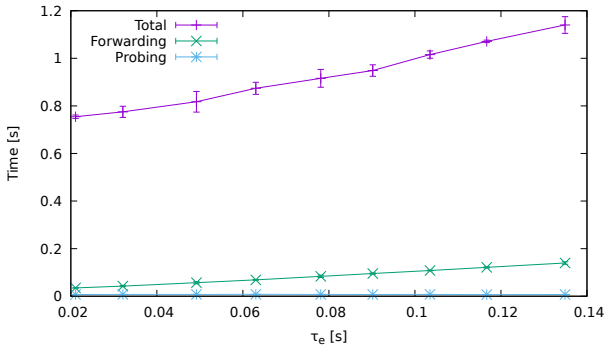


Fig. 7: Effect of τ_e on d_t, d_f, d_p ($\Theta = 2$, $\lambda = 5.50$)

increases, and we can note that the bytes received and sent grow linearly with different slopes: this is justified by the fact that the bytes sent from the node regard only (i) the payload jobs that are forwarded and (ii) the response payload to the traffic generator (that is always the 50kB with the face highlighted independently from the payload of the request). Since we increase the payload while leaving constant the job arrival rate, the bytes sent rate also increases. In particular, we start with and an average of 0.5MB/s when the payload is 50kB to 1.5MB/s when the payload is 800kB. Focusing on the bytes received rate, we observe (iii) the payload of the jobs sent by the traffic generator and the (iv) payload of the job forwarded by other nodes: we start with 0.5MB/s when the payload is 50kB to 5MB/s when the payload is 800kB. The total network traffic estimation is reasonable considering nodes and router network capabilities.

We can conclude the analysis by observing the CPU usage (Figure 9) and the system load (Figure 10). In particular, we can note how the CPU usage in "user" space, which includes the job processing, remains constant during the experiment, while the IRQ processing and the system usage increases. This effect is explained because, by increasing the payload size, we require more packets to be received and to be sent: for this reason, there is more work to be done by the kernel with respect to the job processing. The system load, instead, reflects the overall load of the system, and due to the increase of

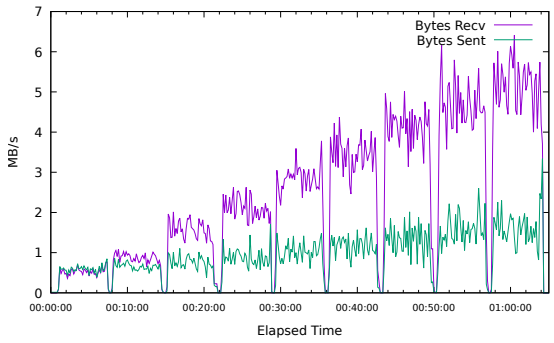


Fig. 8: Network activity for one Raspberry Pi 4B varying the payload size

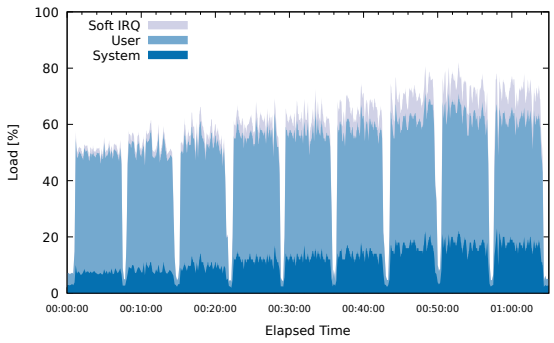


Fig. 9: CPU usage for one Raspberry Pi 4B varying the payload size every 2000 requests (about 10 minutes)

the system and IRQ processing, it grows with the payload size. These last results again confirm the consistency of our experiments.

VI. Conclusion

In this paper we showed a long-term solution for building a cluster of Raspberry Pi that is self-enclosed and tries to minimize the intervention of a human operator. We designed a power supply and a control board system for using a desktop PC case (ATX2RPi8), we presented a set of software guidelines, a Testbed-as-a-service configuration file architecture and finally we showed an usage scenario of the cluster, namely the implementation and the benchmarking of a distributed scheduling algorithm in a very close to real fog computing deployment. However, there are some aspects that should be further investigated, in our cluster deployment, for example, nodes configuration is static and cannot change dynamically. Indeed, it will be needed to consider the possibility to provide a centralized configuration solution for managing the single SBCs operating system parameters (like adding new users or SSH keys), for switching on/off desired nodes, or to set up a particular network topology for running specific benchmarks of other algorithms by using the JSON configuration files proposal that we presented in this work.

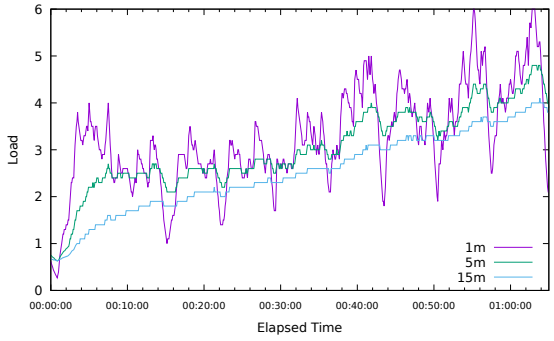


Fig. 10: System load for one Raspberry Pi 4B varying the payload size

References

[1] K. Doucet and J. Zhang, "Learning cluster computing by creating a raspberry pi cluster," in *Proceedings of the SouthEast Conference*, ser. ACM SE '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 191–194. [Online]. Available: https://doi.org/10.1145/3077286.3077324

[2] K. Doucet and J. Zhang, "The creation of a low-cost raspberry pi cluster for teaching," in *Proceedings of the Western Canadian Conference on Computing Education*, ser. WCCCE '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3314994.3325088

[3] M. d'Amore, R. Baggio, and E. Valdani, "A practical approach to big data in tourism: A low cost raspberry pi cluster," in *Information and Communication Technologies in Tourism 2015*, I. Tussyadiah and A. Inversini, Eds. Cham: Springer International Publishing, 2015, pp. 169–181.

[4] J. Saffran, G. Garcia, M. A. Souza, P. H. Penna, M. Castro, L. F. W. Góes, and H. C. Freitas, "A low-cost energy-efficient raspberry pi cluster for data mining algorithms," in *Euro-Par 2016: Parallel Processing Workshops*, F. Desprez, P.-F. Dutot, C. Kaklamanis, L. Marchal, K. Molitorisz, L. Ricci, V. Scarano, M. A. Vega-Rodríguez, A. L. Varbanescu, S. Hunold, S. L. Scott, S. Lankes, and J. Weidendorfer, Eds. Cham: Springer International Publishing, 2017, pp. 788–799.

[5] A. Mappuji, N. Effendy, M. Mustaghfirin, F. Sondok, R. P. Yuniar, and S. P. Pangesti, "Study of raspberry pi 2 quad-core cortex-a7 cpu cluster as a mini supercomputer," in *2016 8th International Conference on Information Technology and Electrical Engineering (ICITEE)*, 2016, pp. 1–4.

[6] P. Abrahamsson, S. Helmer, N. Phaphoom, L. Nicolodi, N. Preda, L. Miori, M. Angriman, J. Rikkilä, X. Wang, K. Hamily, and S. Bugoloni, "Affordable and energy-efficient cloud computing clusters: The bolzano raspberry pi cloud cluster experiment," in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 2, 2013, pp. 170–175.

[7] M. F. Cloutier, C. Paradis, and V. M. Weaver, "A raspberry pi cluster instrumented for fine-grained power measurement," *Electronics*, vol. 5, no. 4, p. 61, 2016.

[8] F. P. Tso, D. R. White, S. Jouet, J. Singer, and D. P. Pezaros, "The glasgow raspberry pi cloud: A scale model for cloud computing infrastructures," in *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*, 2013, pp. 108–112.

[9] S. J. Cox, J. T. Cox, R. P. Boardman, S. J. Johnston, M. Scott, and N. S. O'Brien, "Iridis-pi: a low-cost, compact demonstration cluster," *Cluster Computing*, vol. 17, no. 2, pp. 349–358, 2014. [Online]. Available: https://doi.org/10.1007/s10586-013-0282-7

[10] E. Wilcox, P. Jhunjhunwala, K. Gopavaram, and J. Herrera, "Pi-crust: a raspberry pi cluster implementation," Technical report, Texas A&M University, Tech. Rep., 2015.

[11] Intel, "Atx motherboard specification, v2.2," Tech. Rep. [Online]. Available: https://web.archive.org/web/20120725150314/http://www.formfactors.org/developer/specs/atx2_2.pdf

[12] E. ToolBox. (2003) Awg wire gauges current ratings. [Online]. Available: https://www.engineeringtoolbox.com/wire-gauges-d_419.html

[13] R. Beraldi and G. Proietti Mattia, "Power of random choices made efficient for fog computing," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2020.