Contents

1 Introduction

In this report we will present how we have designed and implemented a discrete-event simulator for required system using the next-event approach.

In order to do that, we have strictly followed each step required by typical discrete-event simulation model[1] whose main results are reported in each sections of this document.

2 Goals

Our simulation model's goal is to capture some relevant global and local characteristics, including time-averaged response time, population and throughput for each class of jobs, through the design and implementation of two version of system based on two different routing policies, called *access control algorithms*.

Obviously we will compare results obtained by our simulations when based on different routing policies and we will care about to derive an analytical solution in order to validate that results not before to have properly define a queueing model.

[1] Lawrence M. Leemis, Stephen K. Park, *Discrete-Event Simulation: A First Course* (Pearson; 1 edition January 6, 2006), Algorithm 1.1.1, page 4

3 Conceptual Model

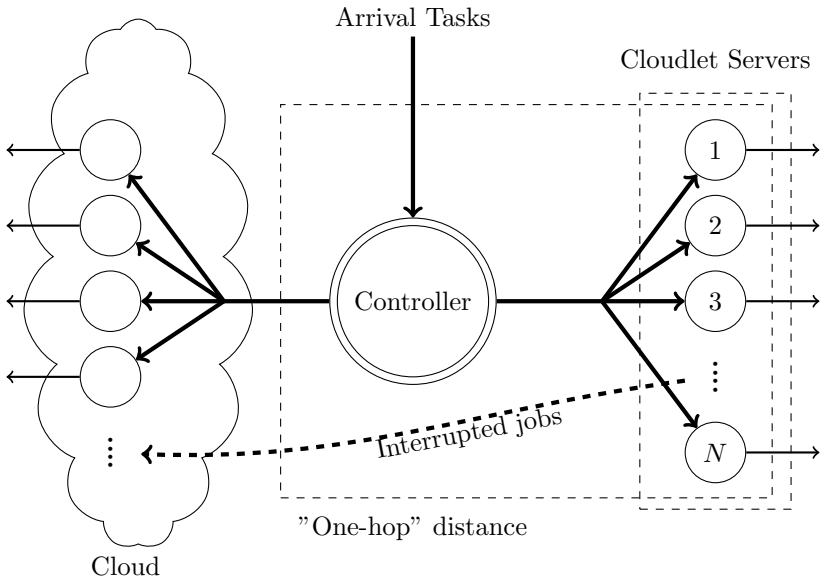


Figure 1: System diagram.

As known, building a system *conceptual model* means to describe it highlighting which state variables are important specifying how they are interrelated.[2]

Our system represents a network of nodes to which some application, running in a set of external mobile devices, send their tasks, called *jobs* or *task*, because of performance or energy saving reasons. Not all jobs sent to system are the same, in fact we distinguish two class type of jobs, that is *class 1* and *class 2* type jobs.

To minimize mean response time experienced by mobile device users, the system is been built as a *two-layer* network in such a way that any arriving job is sent, if possible, to the nearest node to be run.

To be more precise, as you can see from figure 1, our system is made up of a set of nodes including:

Cloudlet node Also called *edge cloud server*, it is a computer cluster located at an "*one-hop*" distance from mobile devices users, that is very close to them, therefore network delay experienced is very low. That cluster is able to guarantee absence of interferences among tasks allocated to it as long as their number does not exceed a given threshold N.

Consequently, from our conceptual model point of view, cloudlet node represents a *fixed-capacity multi-server service node with no queue* which capacity, that is the maximum possible number of jobs in it, is fixed to N.

[2] *Ibid.*

Since jobs inter-arrival times and service times are *exponentially* distributed according to our project specification, a queuing model able to properly represent that node is the M/M/N/N queueing model system[3] although there is a very important different with it: due to adopted routing policies, when cloudlet node is "*full*" and so not capable to run a further job, any arriving job is forwarded somewhere else rather than to cloudlet node, therefore no jobs are really lost.

Cloud node Also called *remote cloud server*, it is another computer cluster which, although it *suffers for greater network delay*, due of its high distance from mobile devices, being virtually made up of unlimited number of servers, has *unlimited resources*, therefore it can to process any number of tasks allocated to it.

Consequently we had modelled this system's node as an *unlimited-capacity multi-server service node with no queue.*

Since, as we previously said, jobs inter-arrival and service times are exponential, due to the presence of an infinite number of server, such a system's node can be modelled by a M/M/∞ queueing model system.[4].

Controller node It represents system's entry point being all arriving jobs sent firstly to it by mobile devices. Its duty is to decide about whether an arrived job should be sent to cloudlet or cloud node according to adopted routing policy.

It is located at the same distance of cloudlet node from mobile devices users, therefore, also in this case, network delay experienced is very low. Although it can run only one job at a time, it has the capability to process each arriving job *instantaneously.*

We have simply modelled it as *single-server service node with no queue.* It's correspondent queuing model system is M/M/1.

Algorithm 1

```
1: function accessControlFunction(arrivalJob)
2:     if (n1 + n2 = N) then
3:         Send arrivalJob on the cloud.
4:     else
5:         Send arrivalJob on the cloudlet.
```

At this point, in order to properly define our conceptual model, we need to specify which *system state variables* are truly important for our goals.

A complete characterization of the our system state is simply provided by the number of each class jobs running both on cloudlet and cloud node at a given instant of time; we will formalize this statement in the next section where we will provide relations and computation methods too.

For now we have to just focus on routing policies. As we previously said, there are two type of access control algorithms used by controller node to make decisions; they are fully described by pseudo-code shown in Algorithm 1 and Algorithm 2. Note that you can use only one access control algorithm type a during a simulation run.

Although they are both *work-conserving scheduling policy*, that is they always perform work on some job when there is a job in the system,[5] there is one difference between them: access

[3] Mor Harchol-Balter, *Performance Modeling and Design of Computer Systems* (Carnegie Mellon University, Pennsylvania 2013), 255-256

[4] *Ibid.*, 271

[5] *Ibid.*, Definition 28.1, page 474

control algorithms 1 does not provide job pre-emption once it starts service on cloudlet node unlike access control algorithms 2, which provides, instead, pre-emption when some condition are achieved requiring also a set-up time to restart interrupted job (note that Algorithm 2 use a further threshold variable to make its decision denoted by S which is less or equal than N).

Algorithm 2

```
function accessControlFunction(arrivalJob)
    if (arrivalJob.isClassOne) then
        if n1 = N then
            Send arrivalJob on the cloud.
        else if n1 + n2 ≤ S then
            Send arrivalJob on the cloudlet.
        else if n2 ≥ 0 then
            Interrupt a class 2 job currently running on cloudlet.
            Send interrupted job to cloud.
            Send arrivalJob on the cloudlet.
        else
            Send arrivalJob on the cloudlet.
    else
        if (n1 + n2 ≥ S) then
            Send arrivalJob on the cloud.
        else
            Send arrivalJob on the cloudlet.
```

4 Specification Model

In this section we will provide a *specification model* of our system in which we will turn all system's states into a collection of mathematical variables together with equations and logic describing how the state variables are interrelated including algorithms for computing their interaction and evolution in time.[6]

In other words, we will provide:

- A set of mathematical state variables that together provide a complete system description.
- A set of system event types.
- A collection of mathematical methods that will take place when each type of event occurs in order to update system state variables.

4.1 System's variables

In order to properly describe all system's state variables, we need to introduce some mathematical notations.

$\tau \in (t_0, t)$ denotes a time's instant of our system simulation clock where t_0 and t represent respectively *start moment* and *final moment* of our simulation; we will use also $c \in \{1, 2\} = C$, representing the class to which a job belongs, and $x \in \{cloudlet, cloud, global\} = X$, used, instead, to refer to a specific system's node or to the whole system.

At this point we can introduce all mathematical variables used in our model:

$$\begin{array}{rcl} n_x^{(c)}(\tau) & = & \text{Number of class } c \text{ jobs currently running at } x \text{ system's node at time } \tau \\ d_x^{(c)}(\tau) & = & \text{Number of class } c \text{ departed jobs from node } x \text{ at time } \tau \\ s_{x,i}^{(c)} & = & \text{Service time of class } c \text{ job } i \text{ served on } x \text{ node} \\ i_{cloudlet}^{(2)}(\tau) & = & \text{Number of class 2 interrupted jobs at time } \tau \text{ which were running on cloudlet} \end{array}$$

We define the *state* of our system at time τ as follows:

$$\omega(\tau) = (\omega_{cloudlet}(\tau), \omega_{cloud}(\tau)) \tag{1}$$

Where $\omega_{cloudlet}(\tau)$ and $\omega_{cloud}(\tau)$ are, respectively, cloudlet and cloud node state at time τ. Specifically:

$$\begin{array}{rcl} \omega_{cloudlet}(\tau) & = & (n_{cloudlet}^{(1)}(\tau), n_{cloudlet}^{(2)}(\tau)) \\ \omega_{cloud}(\tau) & = & (n_{cloud}^{(1)}(\tau), n_{cloud}^{(2)}(\tau)) \end{array} \tag{2}$$

Thus:

$$\omega(\tau) = ((n_{cloudlet}^{(1)}(\tau), n_{cloudlet}^{(2)}(\tau)), (n_{cloud}^{(1)}(\tau), n_{cloud}^{(2)}(\tau)) \tag{3}$$

At this point we can show some equations describing constrains or relations among these variables.

Obliviously is true that:

$$n_{global}^{(c)}(\tau) = \sum_{x \in X \setminus \{global\}} n_x^{(c)}(\tau) \quad \forall c \in C, \forall \tau \in (t_0, t) \tag{4}$$

[6] Lawrence M. Leemis, Stephen K. Park, *Discrete-Event Simulation: A First Course* (Pearson; 1 edition January 6, 2006), 186

$$d^{(c)}_{global}(\tau) = \sum_{x \in X \setminus \{global\}} d^{(c)}_x(\tau) \quad \forall c \in C, \forall \tau \in (t_0, t) \tag{5}$$

If our system is based on Algorithm 1, is true that:

$$\sum_{c \in C} n^{(c)}_{cloudlet}(\tau) \leq N \quad \forall \tau \in (t_0, t) \tag{6}$$

When, instead, our system is using Algorithm 2, is verified that:

$$\begin{aligned} \sum_{c \in C} n^{(c)}_{cloudlet}(\tau) \leq S \quad &\Leftrightarrow \quad n^{(2)}_{cloudlet}(\tau) > 0 \quad \forall \tau \in (t_0, t) \\ n^{(1)}_{cloudlet}(\tau) \leq N \quad &\Leftrightarrow \quad n^{(2)}_{cloudlet}(\tau) = 0 \quad \forall \tau \in (t_0, t) \end{aligned} \tag{7}$$

4.2 Statistics computing

During this subsection we will describe how we can compute all system's statistics which we need using mathematical notation explained before.

Time-averaged population In order to compute $E[N_x]^{(c)}$, that is the *time-averaged number of c class jobs in x system's node*, we can do as follows:[7]

$$E[N_x]^{(c)} \quad = \quad \frac{1}{t - t_0} \int_{t_0}^{t} n^{(c)}_x(\tau) d\tau \quad \forall c \in C, \forall x \in X \tag{8}$$

To calculate instead $E[N_x]$, *not-class-based time-averaged number of jobs running in a x system's node*, we can proceed as shown below:

$$\begin{aligned} E[N_x] \quad &= \quad \frac{1}{t - t_0} \cdot \int_{t_0}^{t} \left(n^{(1)}_x(\tau) + n^{(2)}_x(\tau) \right) d\tau \\ &= \quad \frac{1}{t - t_0} \cdot \int_{t_0}^{t} n^{(1)}_x(\tau) d\tau + \frac{1}{t - t_0} \cdot \int_{t_0}^{t} n^{(2)}_x(\tau) d\tau \\ &= \quad E[N_x]^{(1)} + E[N_x]^{(2)} \\ &= \quad \sum_{c \in C} E[N_x]^{(c)} \qquad\qquad\qquad\qquad\qquad\qquad \forall x \in X \end{aligned} \tag{9}$$

Time-average response time Are $E[S_x]^{(c)}$ and $E[T_x]^{(c)}$, respectively, the *time-averaged service time and response time of class c jobs in x system's node.*

Since our system hasn't queues, there is no waiting time or delay experienced by jobs in our system, therefore result that $E[S_x]^{(c)} = E[T_x]^{(c)}$. Said that, we just have to show how to compute $E[S_x]^{(c)}$:[8]

$$E[T_x]^{(c)} \quad = \quad E[S_x]^{(c)} \quad = \quad \frac{1}{d^{(c)}_x(t)} \cdot \sum_{i=0}^{d^{(c)}_x(t)} s^{(c)}_{x,i} \quad \forall c \in C, \forall x \in X \tag{10}$$

[7] *Ibid.*, 19
[8] *Ibid.*, 17

If we aren't interested about per-class metrics, we can compute $E[S_x]$ and $E[T_x]$, respectively *time-averaged service time* and the *time-averaged response time experienced by any class jobs in a x system's node*, we can do as follows:

$$E[T_x] = E[S_x] = \frac{1}{d_x^{(1)}(t) + d_x^{(2)}(t)} \cdot \left(\sum_{i=0}^{d_x^{(1)}(t)} s_{x,i}^{(1)} + \sum_{i=0}^{d_x^{(2)}(t)} s_{x,i}^{(2)} \right) \quad \forall x \in X \tag{11}$$

Throughput *Per-class system's node throughput*, denoted with $X_x(c)$, is the rate of per-class c completions at x system's node, while *system's node throughput*, indicated with X_x, is the rate of any class job completions at x system's node[9].

Generally the throughput is the ratio between the total number of jobs completed and observation time[10], therefore aforesaid metrics are simply computable as follows:

$$X_x(c) = \frac{d_x^{(c)}(t)}{t} \quad \forall c \in C, \forall x \in X \tag{12}$$

$$X_x = \frac{d_x^{(1)} + d_x^{(2)}(t)}{t} \quad \forall x \in X \tag{13}$$

Percentage class 2 job interruptions Is $p_{interruption}^{(2)}$ the percentage class 2 job interruptions previously running on cloudlet. Note that this metric is computable only when our simulation is using access control algorithm 2.

To compute them we can simply do:

$$p_{interruption}^{(2)} = \frac{i_{cloudlet}^{(2)}(t)}{d_{global}^{(2)}(t)} \tag{14}$$

4.3 System's assumptions

To complete to turn out our conceptual model into a proper specification model, we have to make additional assumptions which we have reported below:

- Although initial state variables can have any non-negative integer value[11], we have choosen, as common, to set $n_x^{(c)}(0) = 0$ and $d_x^{(c)}(0) = 0$ setting, in this way, initial system status as *idle*.
- As a consequence of the previous point, first event must be either a class 1 or class 2 job arrival on controller node.
- Terminal state is also idle. Rather than specifying the number of jobs processed, our stopping criteria has been specified in terms of a time τ^* beyond which no new jobs can arrive. This assumption effectively closes the door at time τ^* but allows the system to continue operation until all jobs have been completely served.

  Therefore, the last event must be either a class 1 or class 2 job completion.

[9] Mor Harchol-Balter, *Performance Modeling and Design of Computer Systems* (Carnegie Mellon University, Pennsylvania 2013), 18

[10] *Ibid.*

[11] *Ibid.* 190

4.4 System's events and assumptions

Let's look to all possible *events* which can occurs during our simulations.

Since by definition any event is an occurrence that may change the state of the system[12], obviously our system's state variables can change only when an event occurs.

All event capable to alter our system's state and their effect on state variables are reported in table 1 but pay attention to the fact that all events reported just focus on access control algorithm 1 based system. When you use access control algorithm 2, you have to remember that, when a class 1 job arrival on cloudlet node occurs, a class 2 job may be interrupted if algorithm 2 conditions would be achieved; in this case $n^{(1)}_{\text{cloudlet}}(\tau)$ are then incremented by one while $n^{(2)}_{\text{cloudlet}}(\tau)$ is decremented by one and a class 2 job arrival is scheduled.

Table 1: Access control algorithm 1 system's events

Event name that occurred at time τ	**Event's place**	**Event's effects**
Class 1 job arrival	Cloudlet	$n^{(1)}_{\text{cloudlet}}(\tau)$++
	Cloud	$n^{(1)}_{\text{cloud}}(\tau)$++
	Controller	$n^{(1)}_{\text{global}}(\tau)$++
Class 2 job arrival	Cloudlet	$n^{(2)}_{\text{cloudlet}}(\tau)$++
	Cloud	$n^{(2)}_{\text{cloud}}(\tau)$++
	Controller	$n^{(2)}_{\text{global}}(\tau)$++
Class 1 job departure	Cloudlet	$n^{(1)}_{\text{cloudlet}}(\tau)$−− $n^{(1)}_{\text{global}}(\tau)$−− $d^{(1)}_{\text{cloudlet}}(\tau)$++ $d^{(1)}_{\text{global}}(\tau)$++
	Cloud	$n^{(1)}_{\text{cloud}}(\tau)$−− $n^{(1)}_{\text{global}}(\tau)$−− $d^{(1)}_{\text{cloud}}(\tau)$++ $d^{(1)}_{\text{global}}(\tau)$++
Class 2 job departure	Cloudlet	$n^{(2)}_{\text{cloudlet}}(\tau)$−− $n^{(2)}_{\text{global}}(\tau)$−− $d^{(2)}_{\text{cloudlet}}(\tau)$++ $d^{(2)}_{\text{global}}(\tau)$++
	Cloud	$n^{(2)}_{\text{cloud}}(\tau)$−− $n^{(2)}_{\text{global}}(\tau)$−− $d^{(2)}_{\text{cloud}}(\tau)$++ $d^{(2)}_{\text{global}}(\tau)$++

[12]Lawrence M. Leemis, Stephen K. Park, *Discrete-Event Simulation: A First Course* (Pearson; 1 edition January 6, 2006), 187

5 Computational Model

Let's start now the description of our system *computational model* in which, as known, system's states exist as a collection of data structure, classes and variables that collectively characterize the system and are systematically updated as simulated time evolves.

Our model is been implemented using Java programming languages, which source code is fully available on GitHub[13], famous web-based hosting service for version control using `git`; we remind that LaTeX source code of this report is available too.[14]

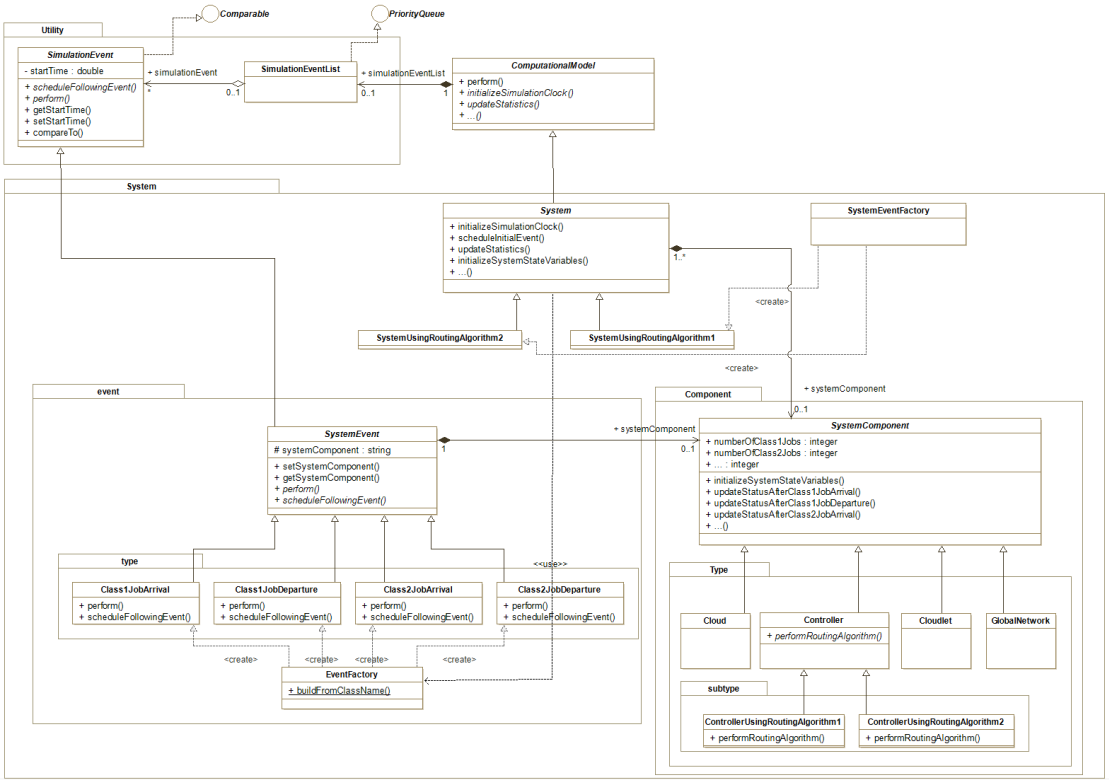


Figure 2: Class diagram showing main classes and methods only.

5.1 Events implementation

In order to properly describe how our implementation works we need to understand how all system's events, seen in previous section, are represented and managed.

As you can see in the class diagram 2, a *generic* system's event has been implemented and represented using an *abstract* Java class called `SimulationEvent` (located into `nexteventsimulation.utility` Java package) which has two main features:

- Has only one `double` type field, called `StartTime`, which is used to hold the instant of time according to which an event occurs. Setter and getter methods are provided in order to manage this field.

[13] Source code available on `https://github.com/AndreaG93/PMCSN-Project`
[14] See `https://github.com/AndreaG93/PMCSN-Project-Report`

- It declares two very important abstract methods which, as we will can see later, are called by next-event simulation algorithm implemented in `ComputationalModel` class; these methods are called `perform()`, whose purpose is to update system's state variables when an event occurs, and `scheduleFollowingEvent()`, which is used, instead, to inserts into an *event list* events that follows the current one.

  Although is obvious that exact methods behaviour depends strictly by which concrete class implements them, is however very important to point out that `SimulationEvent` class does *not* represent system-specific events. In fact, to ensure low coupling and high cohesion among classes, neither `SimulationEvent` or `ComputationalModel` class need to know any specific system information.

System-specific event are instead represented by instances of `SystemEvent` class (located into `nexteventsimulation.computationalmodel.model.system.event` package) whose code is partially shown in Listing 1.

Listing 1: `SystemEvent` class implementation.

```
1  public abstract class SystemEvent extends SimulationEvent {
2
3      protected SystemComponent systemComponent;
4
5      public void setSystemComponent(SystemComponent systemComponent) {
6          this.systemComponent = systemComponent;
7      }
8
9      public SystemComponent getSystemComponent() {
10         return systemComponent;
11     }
12     ...
```

As you can see from reported snipped, `SystemEvent` class has a `SystemComponent` type field, called `systemComponent`, which is used to hold a reference to that system's node where a generic event occurs; in this way any `SystemEvent` instance is able to update state variables of the component of which it has the reference, without altering others parts of system.

However, in order to do that, required abstract method implementation is needed. In our implementation there are four concrete `SystemEvent` type classes used to represents all system's events:

- `Class1JobArrival` class which represents a class 1 job arrival.
- `Class2JobArrival` class which models a class 2 job arrival.
- `Class1JobDeparture` class is used to model a class 1 job departure.
- `Class2JobDeparture` class for class 2 jobs departures.
- `PreviouslyInterruptedClass2JobArrival` this class (not displayed in class diagram shown in Figure 2) represents a class 2 jobs arrival which is been previously interrupted on cloudlet node.
- `PreviouslyInterruptedClass2JobDeparture` like previous one but focused on class 2 jobs departure.

Any instance of above classes, having knowledge about methods published by `SystemComponent` class, can update states of any component of our system. In this way we can achieve an high degree of cohesion among classes and low coupling taking advantage of polymorphism pattern; in fact, as we can see also from shown class diagram, `ComputationalModel` class has to know only about `SimulationEvent` class and, thorough invocation of its methods, being able to update our system ignoring all specific information about our system.

5.2 Event list

As known, an *event list*, also called *calendar*, represents a data structure used to keep track of scheduled time of occurrence for the next possible event.

For reasons of efficiency, our event list is been implemented as a *min-heap binary tree*, that is a *priority queue* in which all events are ordered by time, in such a way that the most imminent event is found on the tree's root. This data structure is able to provide $O(log_2(n))$ time for the en-queuing and dequeuing operations and constant time for root element retrieval.[15]

Fortunately a Java implementation is already available in JDK through built-in class called `PriorityQueue<E>` which provides all method that we need, like `poll`, for root element retrieval, `add` and `remove`, used respectively for en-queuing and dequeuing operations.

However, in order to that data structure works properly, is required that a total ordering among all possible event exists. To provide it, we need that all `SimulationEvent` type instances implement an interface called `Comparable` providing an implementation, shown in listing 2, of a method called `compareTo`.[16] In this way, is very easy to order all `SimulationEvent` type instances based on their start event time, from the youngest to the oldest.

Listing 2: Snippet of `SimulationEvent` class implementation

```
1 @Override
2 public int compareTo(SimulationEvent o) {
3         return Double.compare(this.startTime, o.startTime);
4 }
```

5.3 Simulation clock

As we know a *simulation clock* is necessary to keep track of the current value of simulated time during simulation.

In our system's implementation, our simulation clock is represented by a `SimulationClock` type instance, whose code is partially reported in Listing 3.

Simulation clock state is represented by two `double` type fields: the first, called `currentEventTime`, is used to holding current simulated time value, while the second, called `nextEventTime`, is used instead to keep track of next event simulated time value; obviously, proper setter and getter methods are provided in order to manage them.

From software architecture point of view, believing that multiple `SimulationClock` instance are not necessary for our purposes, we have choosen to design this class using the *singleton* pattern.

Listing 3: `SimulationClock` class implementation.

```
1 public class SimulationClock {
```

[15] Camil Demetrescu, Irene Finocchi, Giuseppe F. Italiano - *Algorimti e strutture dati* - second edition, McGraw-Hill, pag. 200

[16] See `https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/Comparable.html#compareTo(T)`

```
2
3       private static SimulationClock instance = null;
4
5       private double currentEventTime;
6       private double nextEventTime;
7
8       private SimulationClock(){
9       }
10
11      public static SimulationClock getInstance() {
12          if (instance == null)
13              instance = new SimulationClock();
14          return instance;
15      }
16      ...
```

5.4 Next-event simulation logic

The most important code is included into `ComputationalModel` class because it contains main simulation logic.

To start a next-event simulation is very easy because we just need to invoke an its method called `perform` which contains the algorithm[17] used to perform a simulation based on next-event approach, whose Java implementation is reported in Listing 4. Our algorithm consists of the following steps:

Simulation initialization As phase name suggests, during this step system state variables, simulation clock, event list and everything is needed for simulation is initialized compatibly with what we have described in previous section, during specification model analysis.

Remember that `ComputationalModel` is an abstract class which knows anything about our specific system, therefore, obviously, system initialization phase differs according to which concrete class implements that class. Anyhow, as you can see from code snippet reported below, this task is performed invoking `initializeSimulation()` method; that invocation represents a short-cut used to call a large set of abstract methods for initialization purposes, like `initializeSystemStateVariables`, `initializeSimulationClock` and `scheduleInitialEvent`.

Process current event Although we believe the code is self explanatory, we sum up this phase as follows: until event list is not empty (line 6), the most imminent possible event is extracted and removed from event list (line 8); then simulation clock is advanced to this event's scheduled time of occurrence (line 12) and finally system state is updated (line 14).

Schedule following event Until a fixed simulation time τ^* in not exceeded, a new event is generated by current event and consequently placed into event list. Subsequently simulation clock state is updated in order to compute some system metrics.

Statistic update While simulation advance from one event time to the next, through an invocation to an abstract method called `updateStatistics`, is possible to update some variables for system's statistic metrics computation.

[17] Lawrence M. Leemis, Stephen K. Park, *Discrete-Event Simulation: A First Course* (Pearson; 1 edition January 6, 2006), Algorithm 5.1.1, page 189

Simulation result generation During this final step, all collected data are elaborated using various approaches, producing some statistic metrics including graphics like histograms or scatter plots. We will examine involved classes later.

Listing 4: Snippet of `perform` method

```
public void perform() {

        initializeSimulation();

        while (!this.simulationEventList.isEmpty()) {

            SimulationEvent actualEvent = this.simulationEventList.poll()
                ;

            if (actualEvent != null) {

                SimulationClock.getInstance().setCurrentEventTime(
                    actualEvent.getStartTime());

                actualEvent.perform();
                actualEvent.scheduleFollowingEvent();

                SimulationEvent nextEvent = this.simulationEventList.peek
                    ();

                if (nextEvent != null)
                    SimulationClock.getInstance().setNextEventTime(
                        nextEvent.getStartTime());
            }
            updateStatistics();
        }

        BatchMeansRegister.getInstance().computeStatisticsAndWriteData();
        ScatterPlotRegister.getInstance().writingOutputData();
        HistogramsRegister.getInstance().computeStatisticsAndWriteData();

        manageCurrentSimulationResult();
    }
```

5.5 System network implementation

As we have seen in previous subsection, `ComputationalModel` is used to represent a very *generic* form of system (or queues network) because it includes only the code, seen previously, used to perform a next-event simulation.

In fact, we have used another abstract class to represent our specific queues network; that class, which also provide the implementation of all previously mentioned required methods in order to perform our simulation, is called `System`.

Although a `System` type instance represents our specific queues network, it however do *not* contain any system state variables and do *not* perform any statistic metric computation about them; its main purpose is to hold a reference to any system's node, represented by `SystemComponent` objects, in order to manage them.

In fact the state of a `System` type instance is made up of four `SystemComponent` type fields which are used to hold a reference to all system's components: cloud, cloudlet and controller. An extra `SystemComponent` type field is used as a convenient short-cut to represent the global system.

Its other responsibilities are:

- Allocation and management of all `SystemComponent` type objects.
- Methods implementation providing for simulation clock and system nodes state variables initialization.
- Event scheduling management through some methods invocations like `scheduleEventOnCloud`, `scheduleEventOnCloulet` etc.
- Class 2 job interruption management through the so called `removeCloudletClass2JobDeparture` method.

As you can see from class diagram reported in Figure 2, there are two `System`'s concrete subclass called `SystemUsingRoutingAlgorithm1` and `SystemUsingRoutingAlgorithm2`; as their names suggest, they are used respectively to represents a system based on access control algorithm 1 and 2.

Regarding class 2 job interruption management, note that `removeCloudletClass2JobDeparture` is invoked when conditions, established by Algorithm 2, are achieved. When that method is running, a randomly choosen event, representing a cloudlet class 2 departure event, is simply removed from event list.

Now we can finally take a look to `SystemComponent` abstract class which, as previously said, is used to represent any system's node; there are, in fact, its four `SystemComponent` subclasses used to represent any of them: `Cloud`, `Cloudlet`, `Controller` and `GlobalNetwork`.

Main purpose of any `SystemComponent` type instance is to hold a reference to any relevant system's node specific state variables, like actual per-class job population or the number of departed jobs from that specific node which represents; a set of fields, used to compute time-average statistics are also included.

A full list of all class fields representing `SystemComponent` type instance state is shown in Listing 5.

Listing 5: Fields of `SystemComponent` abstract class.

```
1  protected double class1AverageArrivalRate;
2  protected double class2AverageArrivalRate;
3  protected double class1AverageServiceRate;
4  protected double class2AverageServiceRate;
5
6  private int numberOfClass1Jobs;
7  private int numberOfClass2Jobs;
8  private int numberOfClass1DepartedJobs;
9  private int numberOfClass2DepartedJobs;
10
11 private double areaNumberOfClass1Jobs = 0.0;
12 private double areaNumberOfClass2Jobs = 0.0;
13 private double areaServiceTimeClass1Jobs = 0.0;
14 private double areaServiceTimeClass2Jobs = 0.0;
15
16 private double areaServiceTime = 0.0;
```

```
17 private double areaNumberOfJobs = 0.0;
18 ...
```

`SystemComponent` class has other very important responsibilities including:

- System state variables update management every time an event occurs; for example, when a class one job arrives on a certain system's component, `updateStatusAfterClass1JobArrival()` method can be invoked in order to update the number of class one jobs actually present on that component.
- Inter-arrival and service times computation, necessary for event scheduling. As you can see also from Listing 5, `SystemComponent` class has some fields holding component-specific mean rates in order to compute them; these field are initialize in constructor methods of each concrete `SystemComponent` class.
- Method providing in order to model component's behaviour when an event occurs. Note that these methods are abstract and have to be implemented by each `SystemComponent` class in order to reflect the exact specific component behaviour. These method are directly invoked by any `SystemEvent` type instances when they are extracted from event list.

`Cloud`, `Cloudlet`, `Controller` and `GlobalNetwork` class contains the code which declare what to do when an event occurs; we believe that code contain in them is self explanatory therefore refer to it for more informations.

However is very important to focus on `Controller` abstract class which main purpose is, obviously, job routing policy performing. In order to do perform its task, whenever an arrival occurs, a method, called `performRoutingAlgorithm` is invoked; since there are two access control algorithms, we have decide to use two subclasses, called `ControllerUsingRoutingAlgorithm1` and `ControllerUsingRoutingAlgorithm2`, which contains respectively the implementations of access control algorithms 1 and 2, both reported in Listing 6 and 7.

Listing 6: `performRoutingAlgorithm` method implementation concerning access control algorithm 1.

```
1  protected void performRoutingAlgorithm(SystemEvent event) {
2
3     int n1 = this.system.getNumberOfClass1JobOnCloudlet();
4     int n2 = this.system.getNumberOfClass2JobOnCloudlet();
5
6     if ((n1 + n2) == this.system.getThreshold())
7        this.system.scheduleEventOnCloud(event, 0);
8     else
9        this.system.scheduleEventOnCloudlet(event, 0);
10 }
```

Listing 7: `performRoutingAlgorithm` method implementation concerning access control algorithm 2.

```
1  protected void performRoutingAlgorithm(SystemEvent event) {
2
3      int n1 = this.system.getNumberOfClass1JobOnCloudlet();
4      int n2 = this.system.getNumberOfClass2JobOnCloudlet();
5
6          if (event instanceof Class1JobArrival) {
```

```
7
8               if (n1 == this.system.getThreshold())
9                   this.system.scheduleEventOnCloud(event, 0);
10              else if (n1 + n2 < this.system.getThreshold())
11                  this.system.scheduleEventOnCloudlet(event, 0);
12              else if (n2 > 0) {
13
14                  double runningCloudletTimeOfInterruptedJob = this.system.
                        removeCloudletClass2JobDeparture();
15
16                  this.system.scheduleEventOnCloudlet(event, 0);
17
18                  double setupTime = RandomNumberGenerator.getInstance().
                        getExponential(5, 0.8);
19
20                  this.system.scheduleEventOnCloud(SystemEventFactory.
                        buildPreviouslyInterruptedClass2JobArrival(setupTime +
                         runningCloudletTimeOfInterruptedJob), setupTime);
21
22                  this.numberOfInterruptedClass2Jobs++;
23
24              } else
25                  this.system.scheduleEventOnCloudlet(event, 0);
26
27
28          } else {
29
30              this.numberOfTotalClass2Jobs++;
31
32              if ((n1 + n2) >= this.system.getThreshold())
33                  this.system.scheduleEventOnCloud(event, 0);
34              else {
35                  this.numberOfClass2JobsForwardedToCloudlet++;
36                  this.system.scheduleEventOnCloudlet(event, 0);
37              }
38          }
39      }
```

5.6 Class 2 job interruption management

Due to our system design, class 2 job interruption management is exactly the same to normal class 2 jobs handling, although there are very little differences concerning only statistic metric computation.

As previously said, to represent an interrupted class 2 job we have used two classes denoted `PreviouslyInterruptedClass2JobArrival` and `PreviouslyInterruptedClass2JobDeparture`, which are very similar to `Class2JobArrival` and `Class2JobDeparture`.

However, unlike `Class2JobArrival` class, the `PreviouslyInterruptedClass2JobArrival` class has a very important `double` field, called `delay`, which is used to hold a time value representing the sum of *how long it has been running on cloudlet before interruption* and of *setup time*; this field is obviously necessary for statistic metric computation like time-average response time.

As you can see from Listing 7, when a class 2 job has to be interrupted on cloudlet, a method, called `removeCloudletClass2JobDeparture` is invoked. That method, in addition to delete a `Class2JobDeparture` occurrence on cloudlet note from event list, returns a `double` value representing how long the interrupted job has been running so far on cloudlet; to compute them, we can simply to subtract the value of current simulation time with time's instance corresponding to execution start time of the job on cloudlet node. Note that every `Class2JobDeparture` type instances hold a `double` value called `jobExecutionStartTime` representing execution start time of that job.

There are no other conceptual differences respect to `Class2JobArrival`; the effects of the invocation of `perform` method using a `PreviouslyInterruptedClass2JobArrival` type instance has the same effect of using `Class2JobArrival` type instance because same update methods are invoked although in the first case other fields are updated in order to compute statistical metrics. To convince yourself of this, please refer to the code which is very self explanatory.

5.7 Output analysis and graphics generation

In order to properly show all collected data and computed statistics, a very large set of graphics, including scatter plots, histograms and others, are automatically generated using various Java scripts; the total number of generated graphics is 92!

All Java classes, containing logic used to produce all aforesaid graphics, are located into `outputanaylisis` Java package where the most important is `Statistics` class whose purpose is to generate various statistics, like mean or standard deviation, from `double` type raw input data.

In that class two very important methods are present: the first, called `computeMeanAndStandardDeviation` and shown in Listing 8, contains an implementation of *Welford's algorithm* in order to compute mean of input data while the second, called `getConfidenceInterval`, is used to compute confidence interval estimation.[18]

Listing 8: `computeMeanAndStandardDeviation` method implementation

```
1  private void computeMeanAndStandardDeviation() {
2
3          int n = 0;
4          double sum = 0.0;
5          double diff;
6          this.mean = 0.0;
7
8          for (double value : this.randomValuesList) {
9              n++;
10             diff = value - this.mean;
11             sum += Math.pow(diff, 2) * ((n - 1.0) / n);
12             this.mean += diff / n;
13
14             if (value > max)
15                 max = value;
16             else if (value < min)
17                 min = value;
18         }
19
20         this.standardDeviation = Math.sqrt(sum / n);
```

[18] Lawrence M. Leemis, Stephen K. Park, *Discrete-Event Simulation: A First Course* (Pearson; 1 edition January 6, 2006), Algorithm 8.1.1 - pag 354

```
21      }
```

Data output graphics generation is the duty BatchMeans, ScatterPlot, Histograms, EnsembleStatistic . Each existing instance of above classes is used to collect one type data, like job population or throughput, during a simulation in order to generate output graphics.

From software engineering point of view, all these classes are designed using the *registry pattern*[19] because, besides being easy to implement, it guaranties to retrieve a reference to a specific graphics object in a very simple way using its name only, reducing at the same time coupling among classes.

We prefer to omit a detail descriptions of Java classes involved into MATLAB graphics generation because they are just scripts and therefore not relevant from conceptual point of view.

[19] See `https://martinfowler.com/eaaCatalog/registry.html`

6 Analytical solution

In this section we will develop an analytical solution to validate all results obtained through our simulations; in order to do that, we will compare analytic and simulation results in section 8.

6.1 System based on access control Algorithm 1

Let's start with the analysis of the system when its routing policy is based on Algorithm 1.

In order to compute all parameters and metrics of interest associated with that kind of system, we need to focus on cloudlet node behaviour: since we know that, due to its limited resources, it can accept jobs until their number does not exceed a given threshold N, is crucial to firstly compute the *fraction of jobs that are forwarded to cloudlet and to cloud*; in other word, from analytical point of view, our intent is to obtain the probability according to which the sum of job of each class in cloudlet node is equal to that threshold which can permit us to calculate fraction of jobs that are forwarded to each system's node.

To determine this probability, we have modelled cloudlet node as a *continuous-time Markov chain* (CTMC) whose graphical representation is shown in Figure 3; as you can see, each chain's state is denoted with (n_1, n_2), where n_1 and n_2 represent respectively the number of class 1 and class 2 jobs.

6.1.1 Balance equation

After have made CTMC modelling, we can compute *limiting probabilities* $\pi_{(n_1,n_2)}$, namely the probability according to which the chain is in a certain state, say j, independently of the starting state, say i, via *balance* (or *stationary*) *equations*, in which we can equate the rate at which the system leaves state j with the rate at which the system enters state j [20], remembering that limiting probabilities sum to 1, that is $\sum_{j=0}^{\infty} \pi_j = 1$. Those balance equations are shown in table 2.

Table 2: Balance equations.

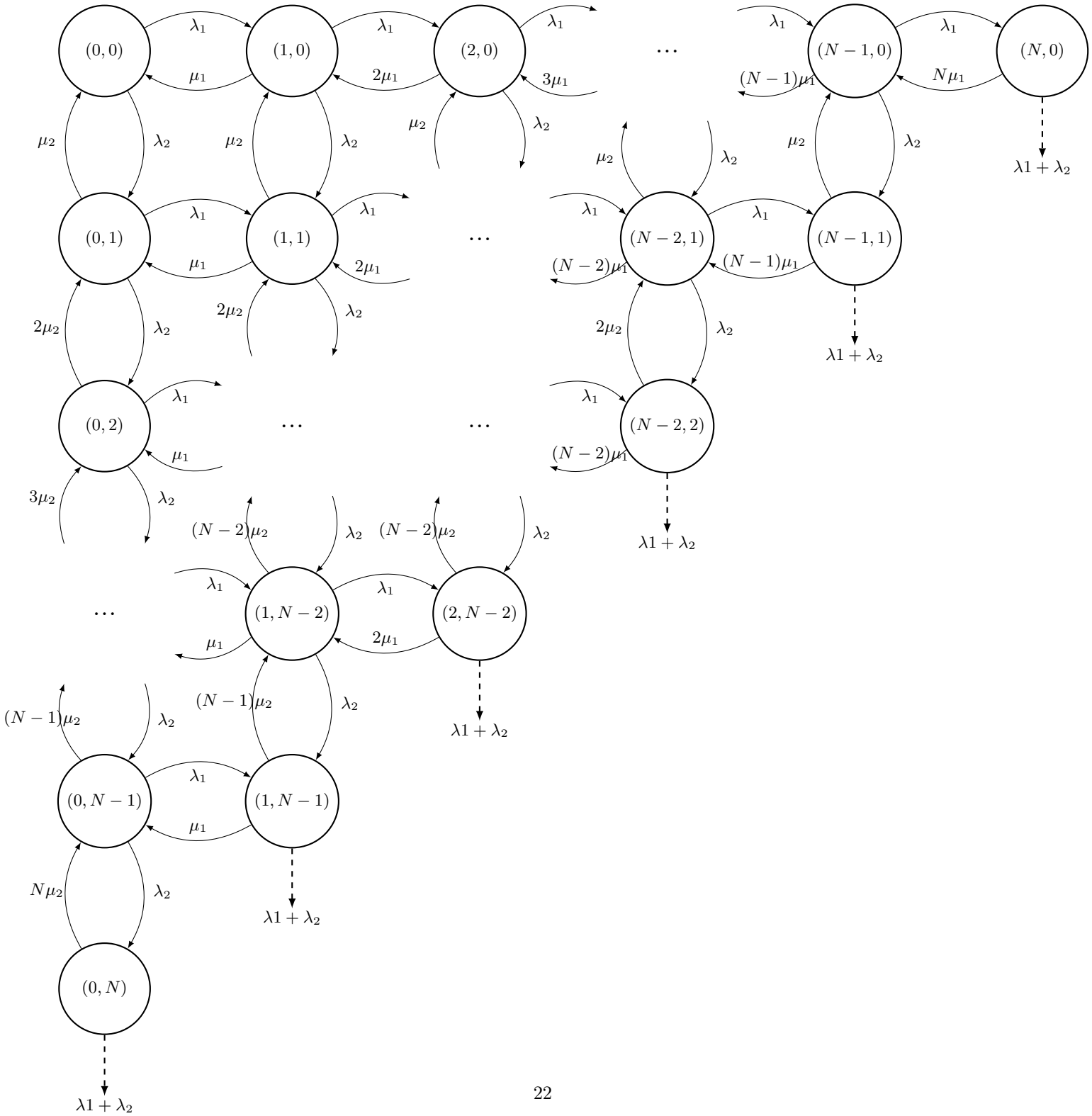
$(\lambda_1 + \lambda_2)\pi_{(0,0)}$	$=$	$\mu_1\pi_{(1,0)} + \mu_2\pi_{(0,1)}$	
$(\lambda_1 + \lambda_2 + n_1\mu_1)\pi_{(n_1,0)}$	$=$	$\lambda_1\pi_{(n_1-1,0)} + \mu_1(n_1+1)\pi_{(n_1+1,0)} + \mu_2\pi_{(n_1,1)}$	$\forall n_1 \in \mathbb{N} \cap [1, N-1]$
$(\lambda_1 + \lambda_2 + n_2\mu_2)\pi_{(0,n_2)}$	$=$	$\lambda_2\pi_{(0,n_2-1)} + \mu_1\pi_{(1,n_2)} + \mu_2(n_2+1)\pi_{(0,n_2+1)}$	$\forall n_2 \in \mathbb{N} \cap [1, N-1]$
$\mu_1 N\pi_{(N,0)}$	$=$	$\lambda_1\pi_{(N-1,0)}$	
$\mu_2 N\pi_{(0,N)}$	$=$	$\lambda_2\pi_{(0,N-1)}$	
$(n_1\mu_1 + n_2\mu_2)\pi_{(n_1,n_2)}$	$=$	$\lambda_1\pi_{(n_1-1,n_2)} + \lambda_2\pi_{(n_1,n_2-1)}$	$\forall n_1, n_2 \in \mathbb{N} \cap [1, N-1] \mid n_1 + n_2 = N$
$(\lambda_1 + \lambda_2 + n_1\mu_1 + n_2\mu_2)\pi_{(n_1,n_2)}$	$=$	$\lambda_1\pi_{(n_1-1,n_2)} + \lambda_2\pi_{(n_1,n_2-1)} + \mu_1(n_1+1)\pi_{(n_1+1,n_2)} + \mu_2(n_2+1)\pi_{(n_1,n_2+1)}$	$\forall n_1, n_2 \in \mathbb{N} \cap [1, N-1] \mid n_1 + n_2 < N$

Regarding balance equations resolution we have develop

A Markov chain for which the limiting probabilities exist is said to be stationary or in steady state if the initial state is chosen according to the stationary probabilities. (page. 138)

[20] Mor Harchol-Balter - *Performance Modeling and Design of Computer Systems* - Carnegie Mellon University, Pennsylvania, page 237

Figure 3: Access control algorithm 1 based cloudlet's CTMC.



6.1.2 Probabilities computing

Having found the stationary probabilities, we can now find $\Pi_{\text{SendToCloud}}$, that is the *probability that an arriving job on controller has to be forwarded to cloud.* Observe that the class to which an arrival job belongs to is not important because, according to access control Algorithm 1, jobs of both classes have same probability to be sent to cloud. To be more precise, $\Pi_{\textbf{SendToCloud}}$ is the probability that an arrival job find that the number of jobs present in cloudlet has exceeded threshold N, which occurs when $n_1 + n_2 = N$. Formally:

$$\begin{aligned} \Pi_{\text{SendToCloud}} &= P\{\text{An arrival job on controller sees } N \text{ jobs in cloudlet}\} \\ &= \text{Limiting probability that there are } N \text{ jobs in system} \\ &= \sum_{\substack{n_1,n_2 \in \mathbb{N} \cap [0,N] \\ n1+n2=N}} \pi_{(n_1,n_2)} \end{aligned} \tag{15}$$

At this point we can easily compute $\Pi_{\textbf{SendToCloudlet}}$, which instead represents the **probability according to which an arriving job on controller has to be accepted on cloudlet** and it is same for both job classes too.

$$\begin{aligned} \Pi_{\text{SendToCloudlet}} &= P\{\text{An arrival job on controller sees less than } N \text{ jobs in cloudlet}\} \\ &= 1 - P\{\text{An arrival job on controller sees } N \text{ jobs in cloudlet}\} \\ &= 1 - \Pi_{\text{SendToCloud}} \end{aligned} \tag{16}$$

6.1.3 Average arrival rates

Is $\lambda_x^{(c)}$ the *total arrival rate into a system's component x of class c job.* Applying previous results, using an appropriate equation[21], we can now compute per-class average arrival rates as follow:

$$\begin{aligned} \lambda_{\text{cloud}}^{(1)} &= \lambda_1 \cdot \Pi_{\text{SendToCloud}} \\ \lambda_{\text{cloud}}^{(2)} &= \lambda_2 \cdot \Pi_{\text{SendToCloud}} \\ \lambda_{\text{cloudlet}}^{(1)} &= \lambda_1 \cdot \Pi_{\text{SendToCloudlet}} \\ \lambda_{\text{cloudlet}}^{(2)} &= \lambda_2 \cdot \Pi_{\text{SendToCloudlet}} \end{aligned} \tag{17}$$

Then we get λ_i, that is **total arrival rate to system's component** i, by summing the per-class rates as follows:

$$\begin{aligned} \lambda_{\text{cloud}} &= \lambda_{\text{cloud}}^{(1)} + \lambda_{\text{cloud}}^{(2)} \\ \lambda_{\text{cloudlet}} &= \lambda_{\text{cloudlet}}^{(1)} + \lambda_{\text{cloudlet}}^{(2)} \end{aligned} \tag{18}$$

[21] *Cfr. Ibid.*, page 315, equation (18.1)

6.1.4 Average population

Is $E[N_x]^{(c)}$ the time-average population of class c jobs into a system's node x.

We can use previously computed stationary probabilities to get average population for cloudlet; it's enough to sum each state's limiting probability multiplied by corresponding number of job as follows:

$$\begin{aligned} E[N^{(1)}_{\text{cloudlet}}] &= \sum_{(n_1,n_2)\in M} n_1 \cdot \pi_{(n_1,n_2)} \\ E[N^{(2)}_{\text{cloudlet}}] &= \sum_{(n_1,n_2)\in M} n_2 \cdot \pi_{(n_1,n_2)} \\ E[N_{\text{cloudlet}}] &= E[N^{(1)}_{\text{cloudlet}}] + E[N^{(2)}_{\text{cloudlet}}] \\ &= \sum_{(n_1,n_2)\in M} (n_1+n_2) \cdot \pi_{(n_1,n_2)} \end{aligned} \tag{19}$$

Since we have modelled cloud component as a $M/M/\infty$ system, in which there is no job's waiting time due to presence of an infinite number of servers, we can simply to apply **Little's Law**[22] in order to get cloud's average population as shown in Equation **??**.

$$\begin{aligned} E[N^{(1)}_{\text{cloud}}] &= \lambda^{(1)}_{\text{cloud}} \cdot E[T^{(1)}_{\text{cloud}}] \\ &= \lambda^{(1)}_{\text{cloud}} \cdot (E[T^{(1)}_{Q_{\text{cloud}}}] + E[S^{(1)}_{\text{cloud}}]) \\ &= \lambda^{(1)}_{\text{cloud}} \cdot (\tfrac{1}{\mu^{(1)}_{\text{cloud}}}) \\ &= \frac{\lambda^{(1)}_{\text{cloud}}}{\mu^{(1)}_{\text{cloud}}} \end{aligned} \tag{20}$$

Similarly:

$$E[N^{(2)}_{\text{cloud}}] = \frac{\lambda^{(2)}_{\text{cloud}}}{\mu^{(2)}_{\text{cloud}}} \tag{21}$$

$$E[N_{\text{cloud}}] = E[N^{(1)}_{\text{cloud}}] + E[N^{(2)}_{\text{cloud}}] \tag{22}$$

Finally we can get global average job populations as follows:

$$\begin{aligned} E[N^{(1)}] &= E[N^{(1)}_{\text{cloudlet}}] + E[N^{(1)}_{\text{cloud}}] \\ E[N^{(2)}] &= E[N^{(2)}_{\text{cloudlet}}] + E[N^{(2)}_{\text{cloud}}] \\ E[N] &= E[N^{(1)}] + E[N^{(2)}] \end{aligned} \tag{23}$$

[22] *Ibid.*, pag. 95, theorem (6.1)

6.1.5 Average response time

Is $E[T_x^{(c)}]$ the *mean response time experienced by a class c jobs into a component x*.
In order to properly compute said metric for each system's component observe that:

- Knowing per-class average job population and per-class mean arrival rates, we can easily compute $E[T_x^{(c)}]$ using Little's Law.
- Since our system haven't queues, no waiting time is experienced by jobs, therefore is true that $E[T_x^{(c)}]$ is also equal to $E[S_x^{(c)}]$, that is *mean service time experienced by a class c jobs into a component x*, which is equal to $1/\mu_x^{(c)}$, where $\mu_x^{(c)}$ means average service rate at which a class c jobs into a component x is served.

Therefore we can get these metric as follows.

$$\begin{aligned} E[T_{\text{cloudlet}}^{(1)}] &= \frac{E[N_{\text{cloudlet}}^{(1)}]}{\lambda_{\text{cloudlet}}^{(1)}} \\ &= E[S_{\text{cloudlet}}^{(1)}] \\ &= \frac{1}{\mu_{\text{cloudlet}}^{(1)}} \end{aligned} \tag{24}$$

$$\begin{aligned} E[T_{\text{cloudlet}}^{(2)}] &= \frac{E[N_{\text{cloudlet}}^{(2)}]}{\lambda_{\text{cloudlet}}^{(2)}} \\ &= E[S_{\text{cloudlet}}^{(2)}] \\ &= \frac{1}{\mu_{\text{cloudlet}}^{(2)}} \end{aligned} \tag{25}$$

$$\begin{aligned} E[T_{\text{cloud}}^{(1)}] &= \frac{E[N_{\text{cloud}}^{(1)}]}{\lambda_{\text{cloud}}^{(1)}} \\ &= E[S_{\text{cloud}}^{(1)}] \\ &= \frac{1}{\mu_{\text{cloud}}^{(1)}} \end{aligned} \tag{26}$$

$$\begin{aligned} E[T_{\text{cloud}}^{(2)}] &= \frac{E[N_{\text{cloud}}^{(2)}]}{\lambda_{\text{cloud}}^{(2)}} \\ &= E[S_{\text{cloud}}^{(2)}] \\ &= \frac{1}{\mu_{\text{cloud}}^{(2)}} \end{aligned} \tag{27}$$

At this point we can get global per-class mean response times as follows:

$$\begin{aligned} E[T^{(1)}] &= E[T^{(1)}_{\text{cloudlet}}] \cdot P\{\text{An arrival class 1 job is sent to cloudlet}\} \\ &\quad + E[T^{(1)}_{\text{cloud}}] \cdot P\{\text{An arrival class 1 job is sent to cloud}\} \\ &= E[T^{(1)}_{\text{cloudlet}}] \cdot \Pi_{\text{SendToCloudlet}} + E[T^{(1)}_{\text{cloud}}] \cdot \Pi_{\text{SendToCloud}} \end{aligned} \tag{28}$$

Similarly:

$$\begin{aligned} E[T^{(2)}] &= E[T^{(2)}_{\text{cloudlet}}] \cdot P\{\text{An arrival class 2 job is sent to cloudlet}\} \\ &\quad + E[T^{(2)}_{\text{cloud}}] \cdot P\{\text{An arrival class 2 job is sent to cloud}\} \\ &= E[T^{(2)}_{\text{cloudlet}}] \cdot \Pi_{\text{SendToCloudlet}} + E[T^{(2)}_{\text{cloud}}] \cdot \Pi_{\text{SendToCloud}} \end{aligned} \tag{29}$$

Finally, using obtained results, we can get global mean response times as shown below:

$$E[T] = E[T^{(1)}] \cdot \frac{\lambda_1}{\lambda_1 + \lambda_2} + E[T^{(2)}] \cdot \frac{\lambda_2}{\lambda_1 + \lambda_2} \tag{30}$$

6.1.6 Throughput

To determine system's throughput let's prove if our system is stable. As we know, every queueing system in which its mean arrival rate is less than its mean service rate is known as a stable system; formally, are λ and μ our system's mean arrival rate and mean service rate respectively, if $\lambda \leq \mu$, given system is stable. Cloudlet subsystem is clearly *not* stable due to its limited resources compared to its workload; we have already seen that exists a not null probability according to which an arriving job on controller sees N jobs in cloudlet which implying its forward to cloud subsystem. Since it haven't a queue, cloudlet stability condition is achieved when $\Pi_{\text{SendToCloud}}$ is null but, based on current system parameters, is not the case. Regarding cloud subsystem, no matter how high we make λ_{cloud} because it is made up of an infinite number of server for which completion rate is still bounded by the arrival rate. Accordingly to previous considerations we can conclude that **our system is stable** due to stability or its cloud subsystem therefore we can get system throughput as follows:

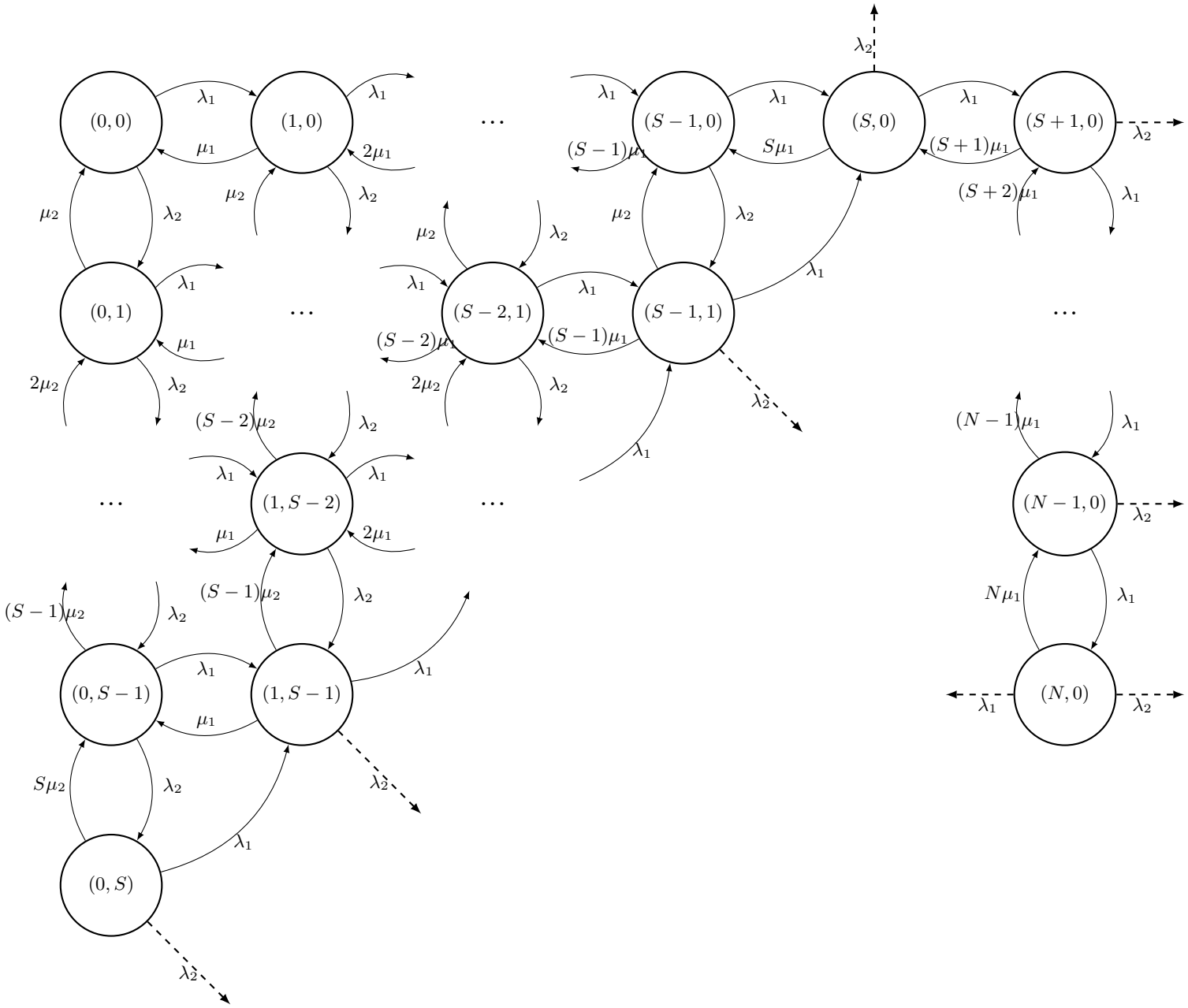
$$X = \lambda = \lambda_1 + \lambda_2 \tag{31}$$

To get per-subsystems throughput we proceed as shown below:

$$X_{\text{cloud}} = \lambda_{\text{cloud}} \tag{32}$$

$$X_{\text{cloudlet}} = X - X_{\text{cloud}} \tag{33}$$

Figure 4: Cloudlet system component modeled using a CTMC



6.2 System based on access control Algorithm 2

Based on our analysis, performance achieved by our system when its routing policy is based on second algorithm, reported in Algorithm 2, are very different from previous case.

The main feature of this kind of system is that a class 2 job, running on cloudlet, may be stopped through its execution on system's cloudlet node and then subsequently resumed in system's cloud node; is important to specify that adopted routing policy is *not* work-conserving because it re-runs interrupted jobs losing work previously done. Moreover, when a class 2 job is interrupted and sent on the system's cloud node, a setup time, denoted whit s_{setup}, has to be considered to restart the task on the cloud.

Most equations described and used in section 6.1 are still valid also for this system, although there are differences about numerical values of each metrics; therefore in this section we focus only to differences compared to previous case.

6.2.1 Balance equation

Regarding balance equation necessary for stationary probabilities computation, since there are no differences about mathematical notation used or significant conceptual differences compared to previously case, we limit ourselves to report them in table 3.

Table 3: Balance equations.

$(\lambda_1+\lambda_2)\pi_{(0,0)}$	$=$	$\mu_1\pi_{(1,0)}+\mu_2\pi_{(0,1)}$	
$(\lambda_1+\lambda_2+n_1\mu_1)\pi_{(n_1,0)}$	$=$	$\lambda_1\pi_{(n_1-1,0)}+\mu_1(n_1+1)\pi_{(n_1+1,0)}+\mu_2\pi_{(n_1,1)}$	$\forall n_1 \in \mathbb{N}\cap[1,S-1]$
$(\lambda_1+\lambda_2+n_2\mu_2)\pi_{(0,n_2)}$	$=$	$\lambda_2\pi_{(0,n_2-1)}+\mu_1\pi_{(1,n_2)}+\mu_2(n_2+1)\pi_{(0,n_2+1)}$	$\forall n_2 \in \mathbb{N}\cap[1,S-1]$
$(S\mu_1+\lambda_1)\pi_{(S,0)}$	$=$	$\lambda_1\pi_{(S-1,0)}+\lambda_1\pi_{(S-1,1)}+(S+1)\mu_1\pi_{(S+1,0)}$	
$(\lambda_1+S\mu_2)\pi_{(0,S)}$	$=$	$\lambda_2\pi_{(0,S-1)}$	
$(n_1\mu_1+n_2\mu_2+\lambda_1)\pi_{(n_1,n_2)}$	$=$	$\lambda_1\pi_{(n_1-1,n_2+1)}+\lambda_1\pi_{(n_1-1,n_2)}+\lambda_2\pi_{(n_1,n_2-1)}$	$\forall n_1,n_2 \in \mathbb{N}\cap[1,S-1] \mid n_1+n_2=S$
$(\lambda_1+\lambda_2+n_1\mu_1+n_2\mu_2)\pi_{(n_1,n_2)}$	$=$	$\lambda_1\pi_{(n_1-1,n_2)}+\lambda_2\pi_{(n_1,n_2-1)}+\mu_1(n_1+1)\pi_{(n_1+1,n_2)}$ $+\mu_2(n_2+1)\pi_{(n_1,n_2+1)}$	$\forall n_1,n_2 \in \mathbb{N}\cap[1,S-1] \mid n_1+n_2<S$
$(n_1\mu_1+\lambda_1)\pi_{(n_1,0)}$	$=$	$\mu_1(n_1+1)\pi_{(n_1+1,0)}+\lambda_1\pi_{(n_1-1,0)}$	$\forall n_1 \in \mathbb{N} \mid S+1\leq n_1\leq N-1$
$N\mu_1\pi_{(N,0)}$	$=$	$\lambda_1\pi_{(N-1,0)}$	

6.2.2 Probabilities computing

Let's start clarifying all differences, compared to previous case, about routing probabilities necessary for analytical results computations.

Is $\Pi^{(c)}_{\text{SendToCloud}}$ the probability according to which an arriving class c job on controller has to be forwarded to system's cloud node. Observing both CTMC and routing policy algorithm is very easy to understand that:

$$\begin{aligned}
\Pi^{(1)}_{\text{SendToCloud}} &= P\{\text{An arrival class 1 job on controller sees } N \text{ class 1 jobs in cloudlet.}\} \\
&= P\{n_1 = N\} \\
&= \pi_{(N,0)}
\end{aligned} \tag{34}$$

$$\begin{aligned}
\Pi^{(2)}_{\text{SendToCloud}} &= P\left\{ \begin{array}{l} \text{An arrival class 2 job on controller sees that number of} \\ \text{jobs in cloudlet exceed or is equal to a given threshold } S. \end{array} \right\} \\
&= P\{n_1 + n_2 \geq S\} \\
&= \sum_{\substack{0 \leq n_1 \leq S \\ 0 \leq n_2 \leq S \\ n_1 + n_2 = S}} \pi_{(n_1,n_2)} + \sum_{n_1 = S+1}^{N} \pi_{(n_1,0)} \\
&= \sum_{\substack{0 \leq n_1 \leq N \\ 0 \leq n_2 \leq S \\ n_1 + n_2 \geq S}} \pi_{(n_1,n_2)}
\end{aligned} \tag{35}$$

Is $\Pi^{(c)}_{\text{SendToCloudlet}}$ the probability that an arriving class c job on controller has to be forwarded to system's cloudlet node.

In order to compute these probabilities we can proceed as shown below:

$$\begin{aligned}
\Pi^{(1)}_{\text{SendToCloudlet}} &= 1 - \Pi^{(1)}_{\text{SendToCloud}} \\
\Pi^{(2)}_{\text{SendToCloudlet}} &= 1 - \Pi^{(2)}_{\text{SendToCloud}}
\end{aligned} \tag{36}$$

To compute $\Pi^{(2)}_{\text{interruption}}$, that is the probability according to which a class 2 job, running on cloudlet, have to be stopped and forwarded to system's cloud node, is necessary know firstly $\Pi_{\text{interruptible}}$, the probability that system's cloudlet node is in a state in which any future class 1 job arrival is able to cause a class 2 job interruption. Formally:

$$
\begin{aligned}
\Pi_{\text{interruptible}} &= P\left\{ \begin{array}{l} \text{System's cloudlet node is in a state in which any future class} \\ \text{1 job arrival is able to cause a class 2 job interruption.} \end{array} \right\} \\
&= P\{n_1 + n_2 = S, n_2 > 0\} \\
&= \sum_{\substack{0 \le n_1 \le S-1 \\ 1 \le n_2 \le S \\ n_1 + n_2 = S}} \pi(n_1, n_2)
\end{aligned}
\tag{37}
$$

At this point we can formally define $\Pi^{(2)}_{\text{interruption}}$ as follows:

$$
\begin{aligned}
\Pi^{(2)}_{\text{interruption}} &= P\left\{ \begin{array}{l} \text{A class 1 job is forwarded to system's cloudlet node when} \\ \text{it is in a state in which any class 1 job arrival is able to} \\ \text{cause a class 2 job interruption.} \end{array} \right\} \\
&= P\left\{ \left(\begin{array}{l} \text{System's cloudlet node is in a} \\ \text{state in which any future class} \\ \text{1 job arrival is able to cause a} \\ \text{class 2 job interruption.} \end{array} \right) \cap \left(\begin{array}{l} \text{A class 1 job is forwarded} \\ \text{to system's cloudlet node.} \end{array} \right) \right\} \\
&= \Pi_{\text{interruptible}} \cdot \Pi^{(1)}_{\text{SendToCloudlet}}
\end{aligned}
\tag{38}
$$

6.2.3 Average arrival rates

Finally, applying previous results, we can now compute per-class average arrival rates as follow of our system:

$$
\begin{aligned}
\lambda^{(1)}_{\text{cloud}} &= \lambda_1 \cdot \Pi^{(1)}_{\text{SendToCloud}} \\
\lambda^{(2)}_{\text{cloud}} &= \lambda_2 \cdot \Pi^{(2)}_{\text{SendToCloud}} \\
\lambda^{(1)}_{\text{cloudlet}} &= \lambda_1 \cdot \Pi^{(1)}_{\text{SendToCloudlet}} \\
\lambda^{(2)}_{\text{cloudlet}} &= \lambda_2 \cdot \Pi^{(2)}_{\text{SendToCloudlet}}
\end{aligned}
\tag{39}
$$

Then we can get λ_x, that is total arrival rate to system's component x, proceeding as follows:

$$
\begin{aligned}
\lambda_{\text{cloud}} &= \lambda^{(1)}_{\text{cloud}} + \lambda^{(2)}_{\text{cloud}} + \lambda^{(2)}_{\text{cloudlet}} \cdot \Pi^{(2)}_{\text{interruption}} \\
\lambda_{\text{cloudlet}} &= \lambda^{(1)}_{\text{cloudlet}} + \lambda^{(2)}_{\text{cloudlet}}
\end{aligned}
\tag{40}
$$

6.2.4 Other metrics

In order to compute $E[N_x]^{(c)}$ and $E[T_x]^{(c)}$, respectively averaged population and response time of class c jobs running in a x system's node, is possible to use same equations described in previous section because, although stationary probabilities and arrival rates have different numeric values, due to different routing policy, there are no conceptual differences about computation of those metrics.

Analytical methods used to compute system throughput are pretty the same as those used in previous section. The only difference lies in cloudlet class 2 throughput $X^{(2)}_{\text{cloudlet}}$ computation, computable using equation shown below:

$$X^{(2)}_{\text{cloudlet}} = \lambda^{(2)}_{\text{cloudlet}} - \lambda^{(2)}_{\text{cloudlet}} \cdot \Pi^{(2)}_{\text{interruption}} \tag{41}$$

Regarding instead $E[T]^{(2)}$, that is global averaged response time experienced by class 2 jobs, there are, instead, some conceptual differences due to jobs interruptions. Formally:

$$\begin{aligned} E[T]^{(2)} = {} & \Pi^{(2)}_{\text{SendToCloud}} \cdot E[T^{(2)}_{\text{cloud}}] + \\ & \Pi^{(2)}_{\text{SendToCloudlet}} \cdot \left(\Pi^{(2)}_{\text{interruption}} \cdot (E[S_{setup}] + E[T^{(2)}_{\text{cloud}}]) + (1 - \Pi^{(2)}_{\text{interruption}}) \cdot E[T^{(2)}_{\text{cloudlet}}] \right) \end{aligned} \tag{42}$$

Finally, in order to compute average response time relating to interrupted class 2 jobs only, denoted with $E[T]^{(2)}_{\text{interrupted}}$, we can simply proceed as shown below:

$$E[T]^{(2)}_{\text{interrupted}} = E[S_{\text{setup}}] + E[T^{(2)}_{\text{cloud}}] + \alpha \cdot E[T^{(2)}_{\text{cloudlet}}] \qquad \forall \alpha \in \mathbb{R} \mid 0 \leq \alpha \leq 1 \tag{43}$$

In order to compute the fraction of time during which interrupted class 2 job was running on the cloudlet, we have used an attenuation factor α which value, determined experimentally, is equal to 0.649.

6.3 Summary of analytical results

All analytical results are been computed using a combination of Java and MATLAB[23] scripts.

In fact, all balance equations, previously shown in tables 2 and 3, are firstly automatically generated using simple Java scripts[24] able to built some MATLAB scripts as output[25] subsequently executed using MATLAB environment.

Is very important to precise that all equation systems are been resolved using a MATLAB function called `solve`,[26] which returns all systems solutions as *symbolic variables* or `sym` objects.[27]

A symbolic variable can hold an expression instead of just a numeric value allowing an higher precision than hardware numbers so is possible to compute more decimal places; obliviously the trade off is that symbolic computations need more memory and are slower than hardware computation.

Practically all analytical solutions has been represented by MATLAB as fractions some of which are made up of very big numbers because which has been impossible to fit in this page; just think to symbolic variable holding $\Pi^{(2)}_{\text{SendToCloud}}$ solution which has a 658 digits number as numerator!

In order to represent analytical solutions we have used two MATLAB functions called digits and `digits`[28] and `vpa`[29]; the first function sets the precision used by `vpa` to d significant decimal digits while the second is used to evaluate symbolic input to at least d significant digits. Using these functions was possible to list analytical results below.

[23] See. `https://www.mathworks.com/products/matlab.html`

[24] See `CTMCResolverScriptGenerator.java`, `ResolverUsingRoutingAlgorithm1.java` and `ResolverUsingRoutingAlgorithm2.java` files.

[25] See `MATLAB_ALG1_CTMC_ResolverScript.m` and `MATLAB_ALG1_CTMC_ResolverScript.m` files

[26] See `https://www.mathworks.com/help/symbolic/solve.html`

[27] See `https://it.mathworks.com/help/symbolic/sym.html`

[28] See `https://it.mathworks.com/help/symbolic/digits.html`

[29] See `https://it.mathworks.com/help/symbolic/vpa.html`

Table 4: Analytical results of Algorithm 1 based system. (Part 1)

Variable	Approximate value	Symbolic variable representation
$\Pi_{SendToCloud}$	0.413925392	$\frac{8785943692625664651725111763842474139347534179687500000}{21225911349808585752097139398361184558580062213704427 4729}$
$\Pi_{SendToCloudlet}$	0.586074607	$\frac{1243996765718292110037202763451871041923252803401692747 29}{21225911349808585752097139398361184558580062213704427 4729}$
$\lambda^{(1)}$, $X^{(1)}$	4	
$\lambda^{(2)}$, $X^{(2)}$	6.25	
λ, X	10.25	
$\lambda^{(1)}_{cloudlet}$, $X^{(1)}_{cloudlet}$	2.344298428	$\frac{2639445382678245}{1125899906842624}$
$\lambda^{(2)}_{cloudlet}$, $X^{(2)}_{cloudlet}$	3.662966295	$\frac{65986134566956125}{18014398509481984}$
$\lambda_{cloudlet}$, $X_{cloudlet}$	6.007264723	$\frac{108217260689808045}{18014398509481984}$
$\lambda^{(1)}_{cloud}$, $X^{(1)}_{cloud}$	1.655701571	$\frac{1864154244692251}{1125899906842624}$
$\lambda^{(2)}_{cloud}$, $X^{(2)}_{cloud}$	2.587033704	$\frac{46603856117306275}{18014398509481984}$
λ_{cloud}, X_{cloud}	4.242735276	$\frac{76430324032382291}{18014398509481984}$
$E[N^{(1)}_{cloudlet}]$	5.204009618	$\frac{1104598468206022474496767594370330477471953336910445636480}{21225911349808585752097139398361184558580062213704427 4729}$
$E[N^{(2)}_{cloudlet}]$	13.55210838	$\frac{2876558510953183527335332277006068951749878481537618845000}{21225911349808585752097139398361184558580062213704427 4729}$

Table 5: Analytical results of Algorithm 1 based system. (Part 2)

Variable	Approximate value	Symbolic variable representation
$E[N_{cloudlet}]$	18.75611799	$\frac{3981156979159206001832099871376399429221831818448064481480}{21225911349808585752097139398361184558580062213704427 4729}$
$E[N^{(1)}_{cloud}]$	6.622806284	$\frac{1864154244692251}{281474976710656}$
$E[N^{(2)}_{cloud}]$	11.759244113	$\frac{1165096402932656875}{99079191802150912}$
$E[N_{cloud}]$	18.382050397	$\frac{1821278697064329227}{99079191802150912}$
$E[N^{(1)}]$	11.826815902	$\frac{13315810923168857}{1125899906842624}$
$E[N^{(2)}]$	25.311352493	$\frac{2507828348524290043}{99079191802150912}$
$E[N]$	37.138168396	$\frac{3679619709763149459}{99079191802150912}$
$E[T^{(1)}_{cloudlet}]$, $E[S^{(1)}_{cloudlet}]$	$2.\overline{2}$	$\frac{20}{9}$
$E[T^{(2)}_{cloudlet}]$, $E[S^{(2)}_{cloudlet}]$	$3.\overline{7037}$	$\frac{100}{27}$
$E[T_{cloudlet}]$, $E[S_{cloudlet}]$	3.125564588	$\frac{3460}{1107}$
$E[T^{(1)}_{cloud}]$, $E[S^{(1)}_{cloud}]$	4	
$E[T^{(2)}_{cloud}]$, $E[S^{(2)}_{cloud}]$	$4.\overline{54}$	$\frac{50}{11}$

Table 6: Analytical results of Algorithm 1 based system. (Part 3)

Variable	**Approximate value**	**Symbolic variable representation**
$E[T_{cloud}]$, $E[S_{cloud}]$	4.332594235	$\frac{1954}{451}$
$E[T^{(1)}]$, $E[S^{(1)}]$	2.958089587	$\frac{2497884592968457}{844424930131968}$
$E[T^{(2)}]$, $E[S^{(2)}]$	4.052125751	$\frac{903333025213434725}{222928181554839552}$
$E[T]$, $E[S]$	3.625184809	$\frac{33134390151034630493}{9140055443748421632}$

Table 7: Analytical results of Algorithm 2 based system. (Part 1)

Variable	Approximate value	Symbolic variable representation
$\Pi^{(1)}_{SendToCloud}$	0.0005377649	$\frac{703687441776640000000000000000}{1308540831314824230819390435296941}$
$\Pi^{(2)}_{SendToCloud}$	0.356859509	To big to be represented!
$\Pi^{(1)}_{SendToCloudlet}$	0.999462235	$\frac{1307837143873047590819390435296941}{1308540831314824230819390435296941}$
$\Pi^{(2)}_{SendToCloudlet}$	0.643140490	To big to be represented!
DA VEDERE $\Pi^{(2)}_{Interrupted}$	0.35705151957534072695649962275511	To big to be represented!
$\lambda^{(1)}$, $X^{(1)}$	4	
$\lambda^{(2)}$, $X^{(2)}$	6.25	
λ, X	10.25	
$\lambda^{(1)}_{cloudlet}$, $X^{(1)}_{cloudlet}$	3.997848940	$\frac{9002355498360251}{2251799813685248}$
$\lambda^{(2)}_{cloudlet}$	4.019628063	To big to be represented!
$X^{(2)}_{cloudlet}$	2.588117409	To big to be represented!
$\lambda_{cloudlet}$	8.01747700	To big to be represented!
$X_{cloudlet}$	6.585966349	To big to be represented!
$\lambda^{(1)}_{cloud}$, $X^{(1)}_{cloud}$	0.002151059	$\frac{4960006533878655}{2305843009213693952}$
$\lambda^{(2)}_{cloud}$, $X^{(2)}_{cloud}$	3.661882590_{36}	To big to be represented!
λ_{cloud}, X_{cloud}	3.664033650	To big to be represented!
$E[N^{(1)}_{cloudlet}]$	8.884108755	$\frac{116252190566493119183945816470839 20}{1308540831314824230819390435296941}$
$E[N^{(2)}_{cloudlet}]$	9.608670684	To big to be represented!

Table 8: Analytical results of Algorithm 2 based system. (Part 1)

Variable	**Approximate value**	**Symbolic variable representation**
$E[N_{cloud}^{(2)}]$	16.644920866	To big to be represented!
$E[N_{cloud}]$	16.653525106	To big to be represented!
$E[N^{(1)}$	8.892712995	$\frac{1163647805571773815839458164708392 0}{1308540831314824230819390435296941}$
$E[N^{(2)}$	27.375749424	To big to be represented!
$E[N]$	36.2684624	To big to be represented!
$E[T_{cloudlet}^{(1)}]$, $E[S_{cloudlet}^{(1)}]$	$2.\overline{2}$	$\frac{20}{9}$
$E[T_{cloudlet}^{(2)}]$, $E[S_{cloudlet}^{(2)}]$	$3.\overline{7037}$	$\frac{100}{27}$
$E[T_{cloudlet}]$, $E[S_{cloudlet}]$	2.804406796	To big to be represented!
$E[T_{cloud}^{(1)}]$, $E[S_{cloud}^{(1)}]$	4	
$E[T_{cloud}^{(2)}]$, $E[S_{cloud}^{(2)}]$	$4.\overline{54}$	$\frac{50}{11}$
$E[T_{cloud}]$, $E[S_{cloud}]$	4.545134323	To big to be represented!
$E[T^{(1)}]$, $E[S^{(1)}]$	2.223178248	$\frac{46136700210409393015}{20752587082923245568}$
$E[T^{(2)}]$, $E[S^{(2)}]$	4.380119907	To big to be represented!
$E[T]$, $E[S]$	3.538386577	To big to be represented!

7 Model verification

Our computational model is been verified by an extensive testing process, made using debugging systems provided by our programming environment,[30] during which we have focused on checking whether our computational model is working as expected.

8 Model validation

Model validation was performed by comparing simulation results with previously described analytical results

Based on our results, verified computational model is a reasonable approximation of required system; in mostly all cases, means, computed analytically, lie in our 95% confidence interval estimates.

[30] We have used IntelliJ IDEA as integrated development environment (IDE).

8.1 Algorithm 1 simulation results

8.1.1 Time-Average Class 1 Job Population

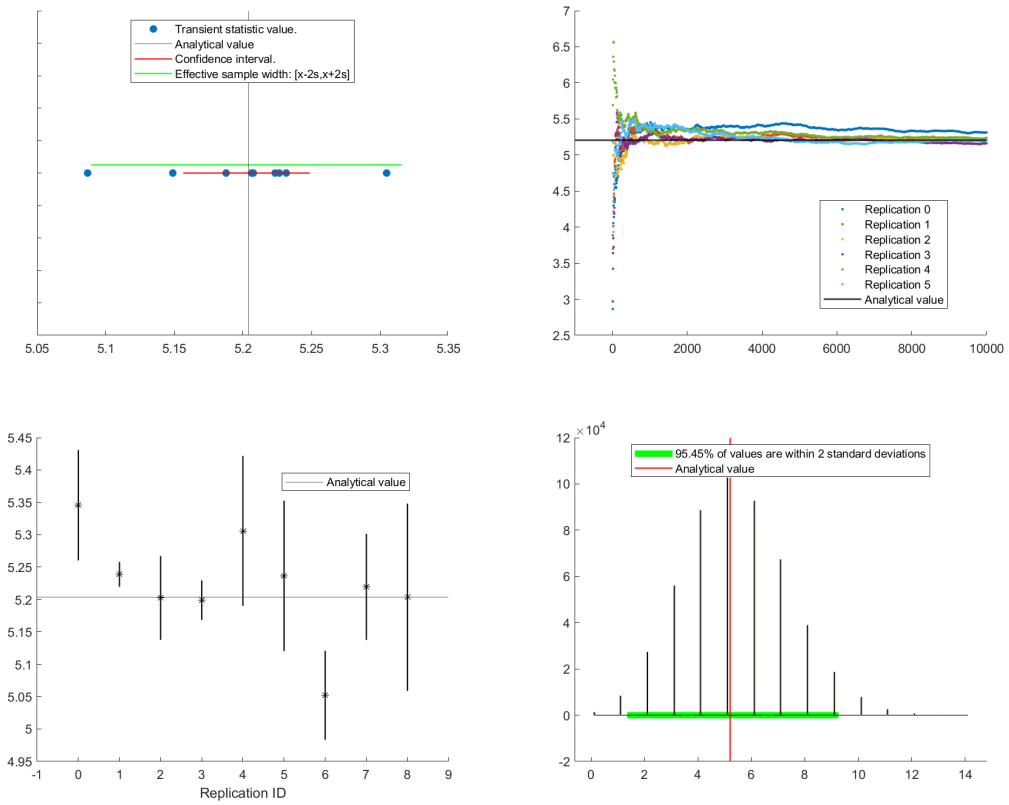


Figure 5: Cloudlet

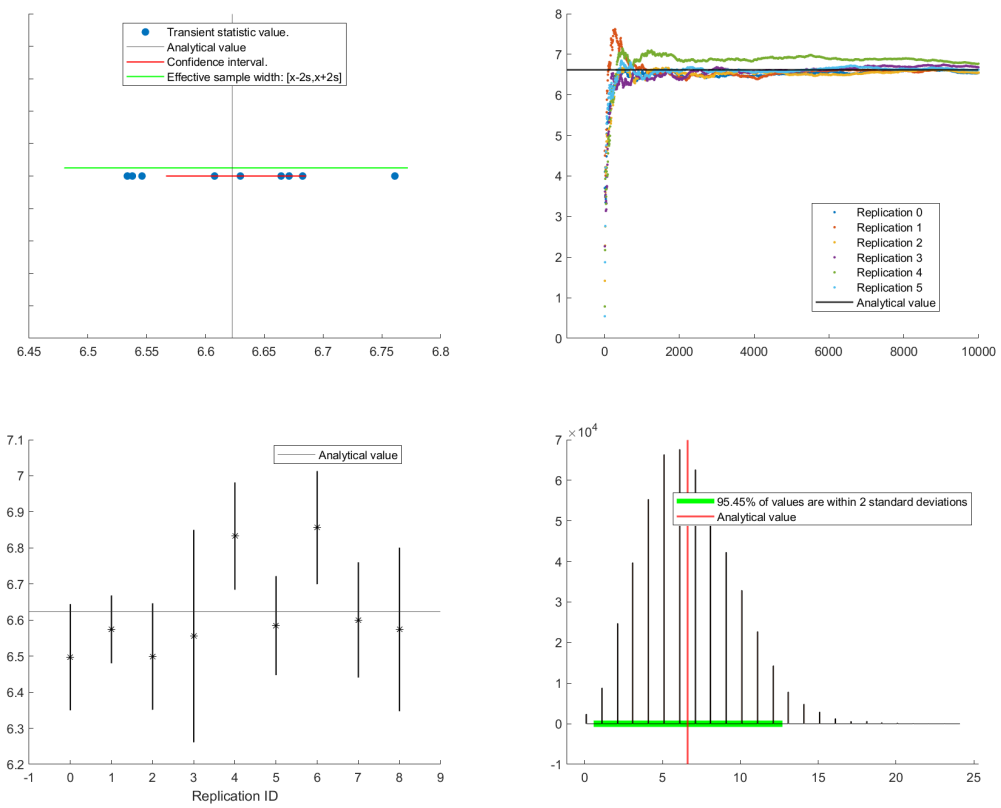


Figure 6: Cloud

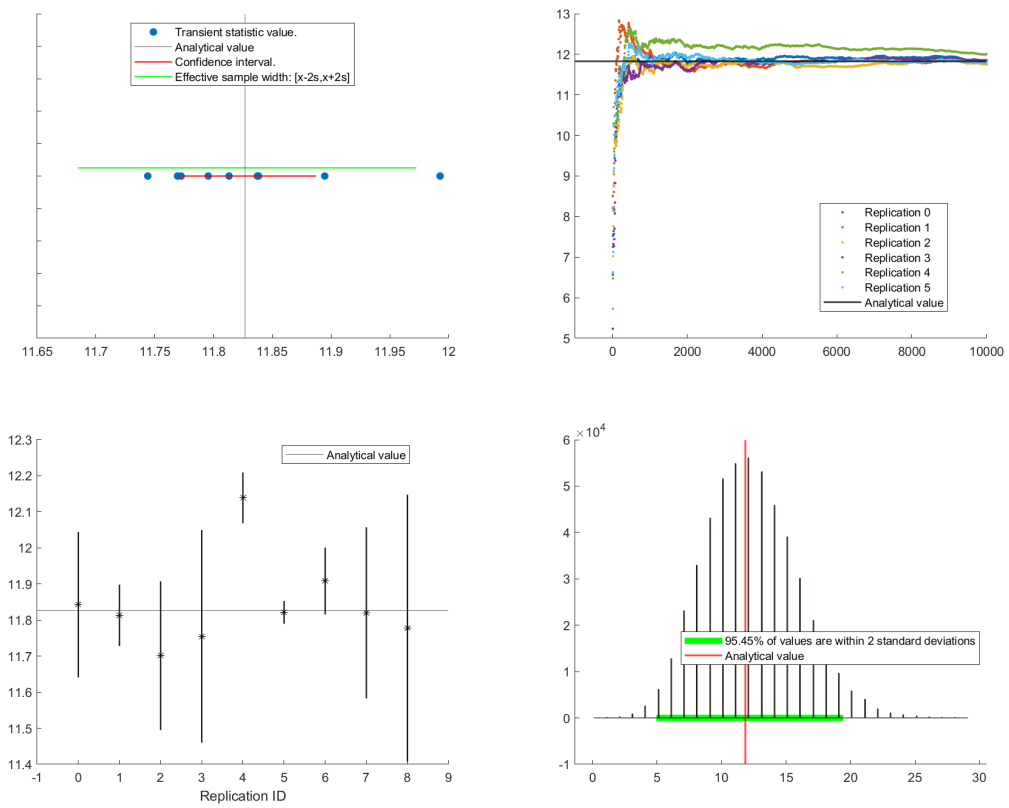


Figure 7: Global

8.1.2 Time-Average Class 2 Job Population

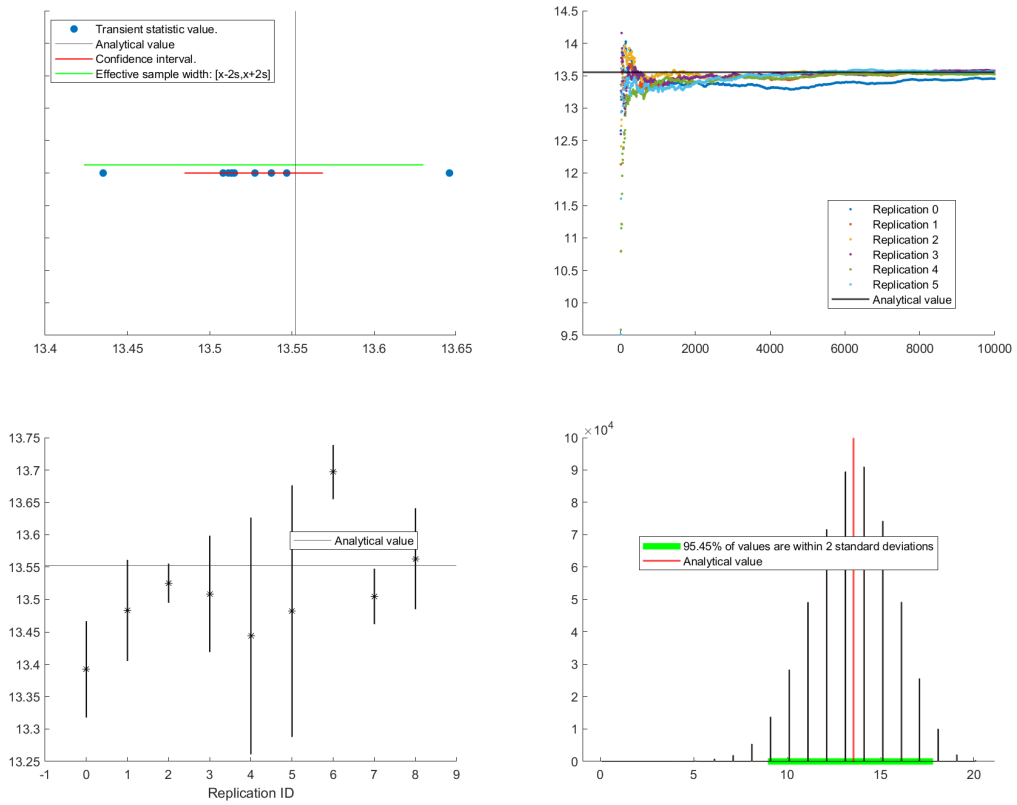


Figure 8: Cloudlet

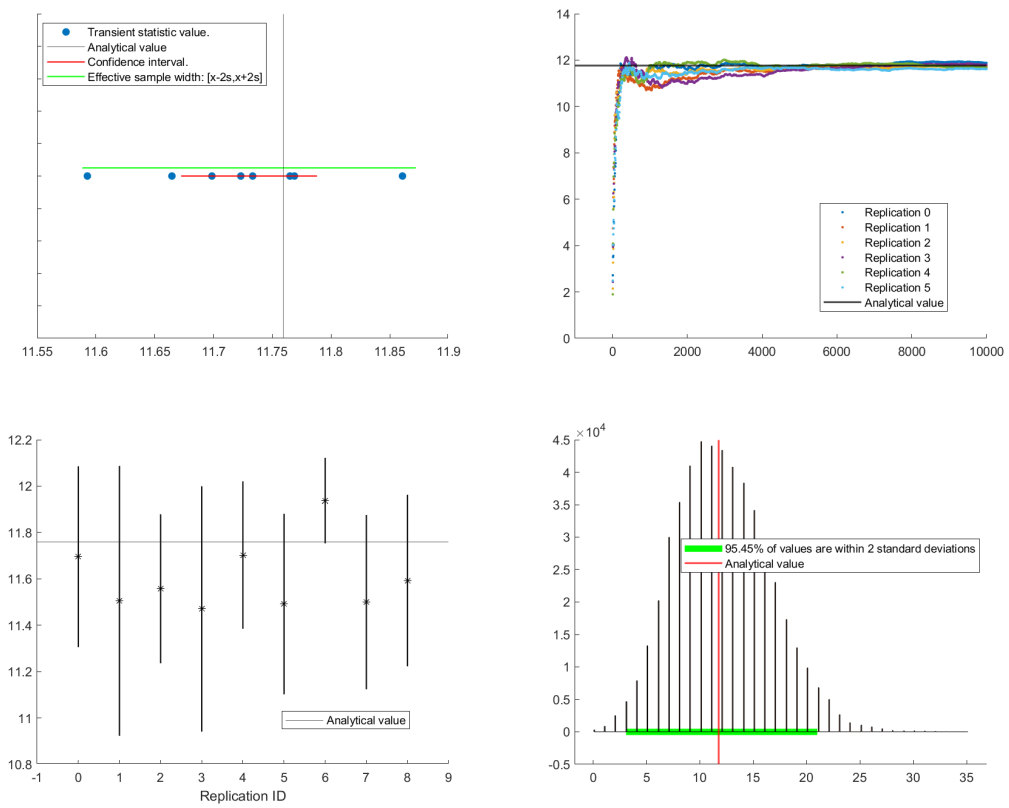


Figure 9: Cloud

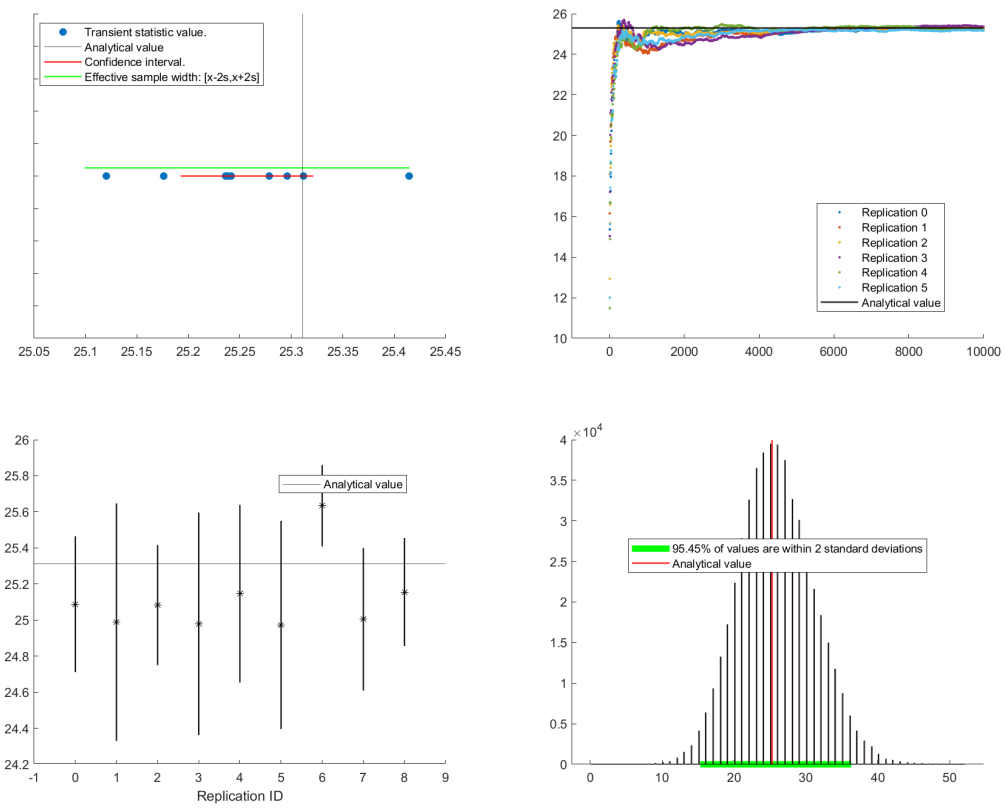


Figure 10: Global

8.1.3 Time-Average Job Population

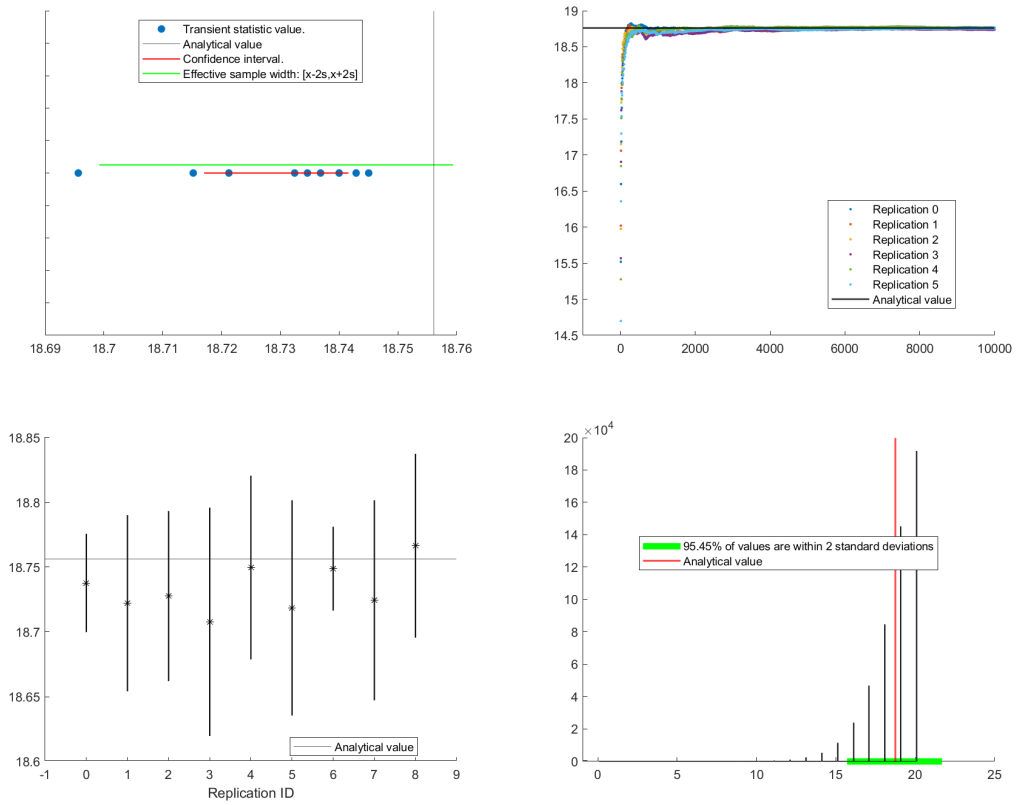


Figure 11: Cloudlet

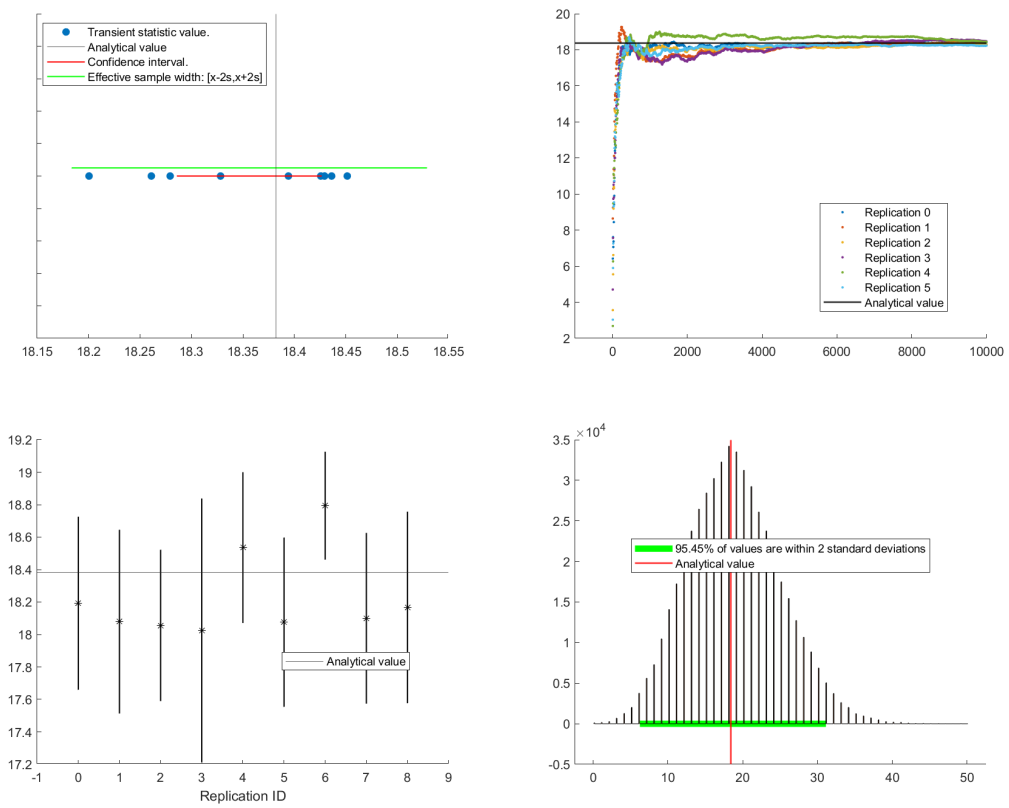


Figure 12: Cloud

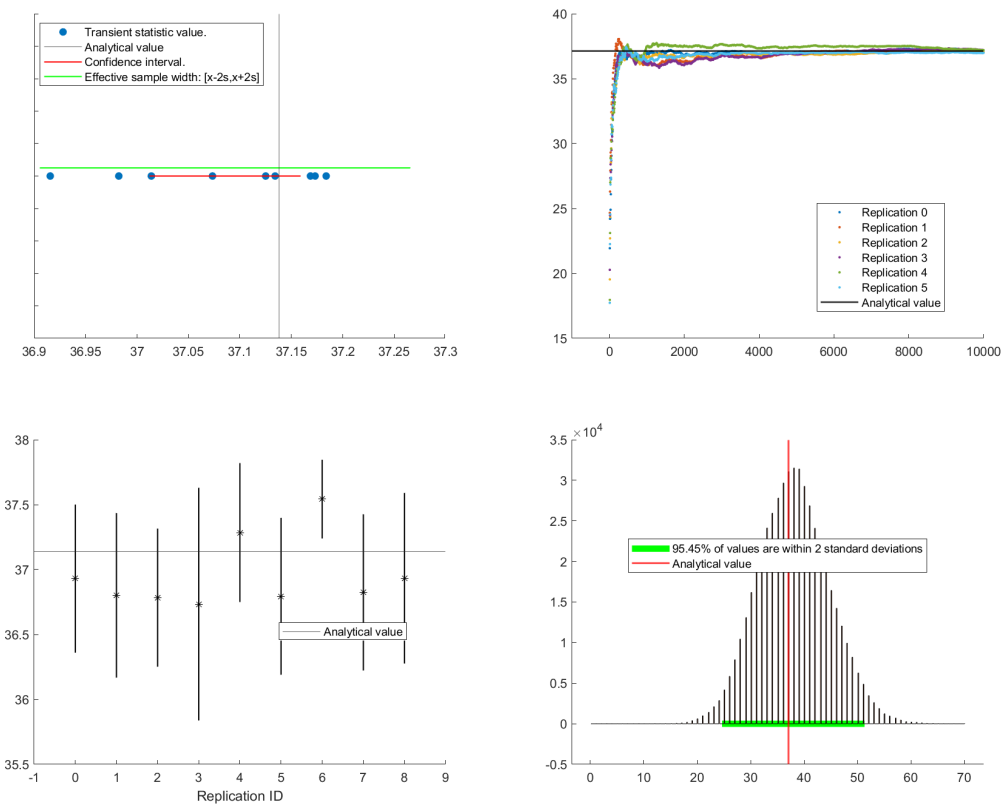


Figure 13: Global

8.1.4 Time-Average Class 1 Job Service/Response Time

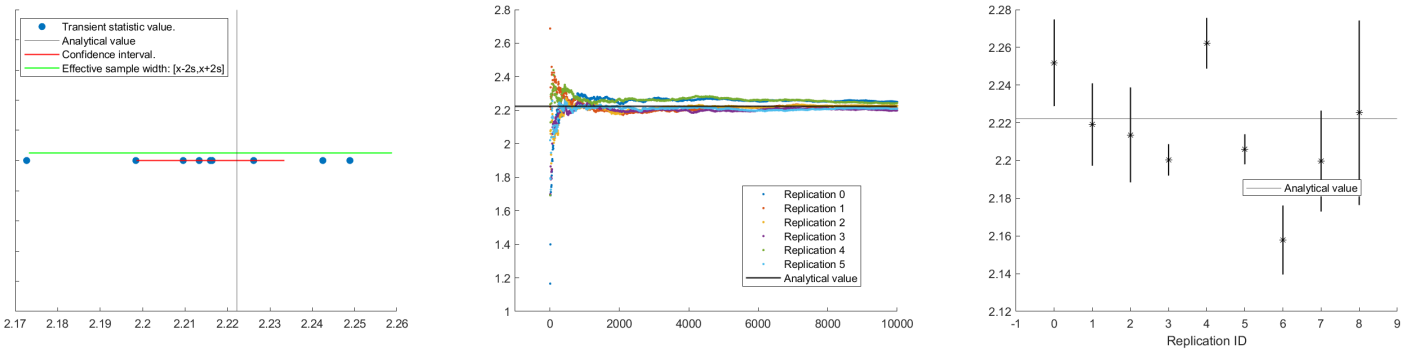


Figure 14: Cloudlet

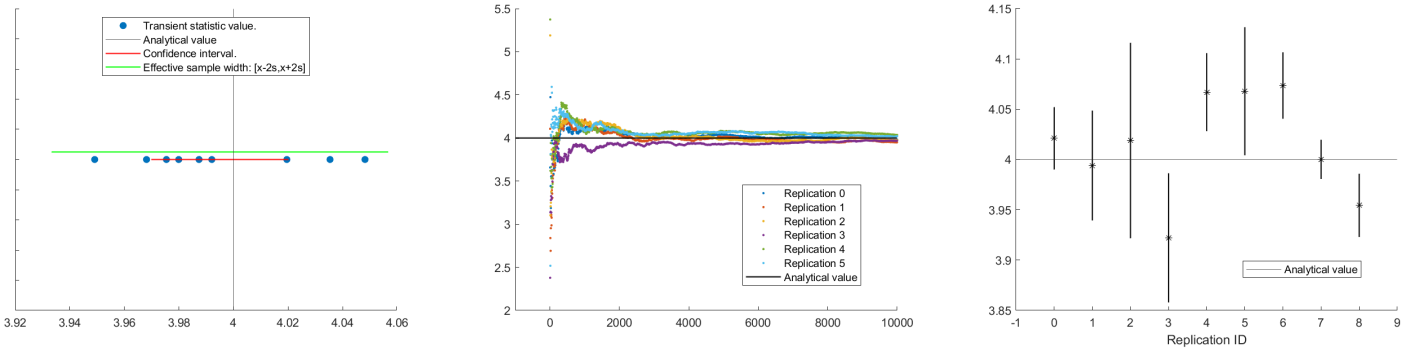


Figure 15: Cloud

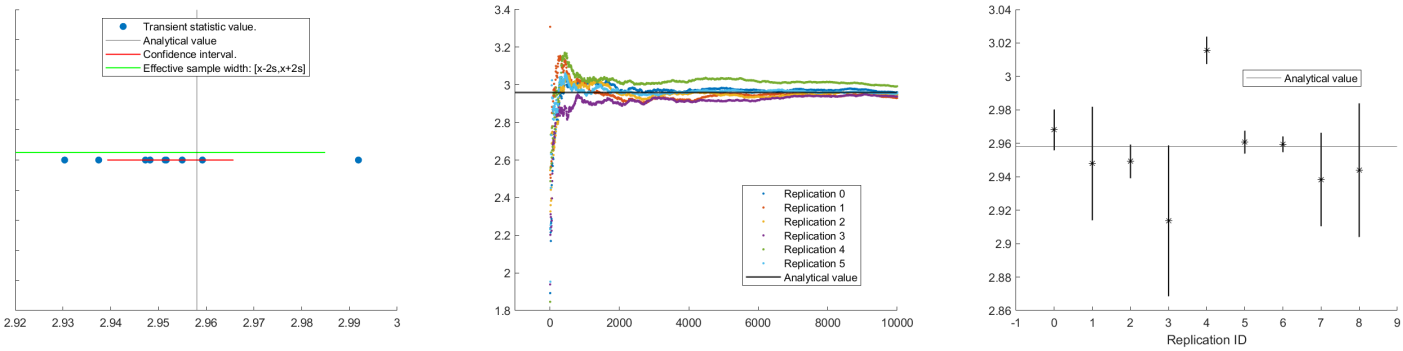


Figure 16: Global

8.1.5 Time-Average Class 2 Job Service/Response Time

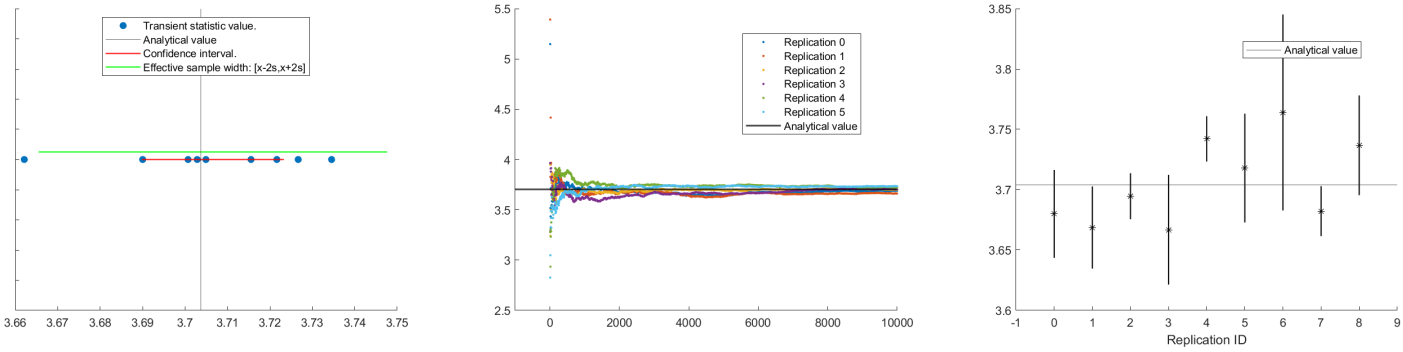


Figure 17: Cloudlet

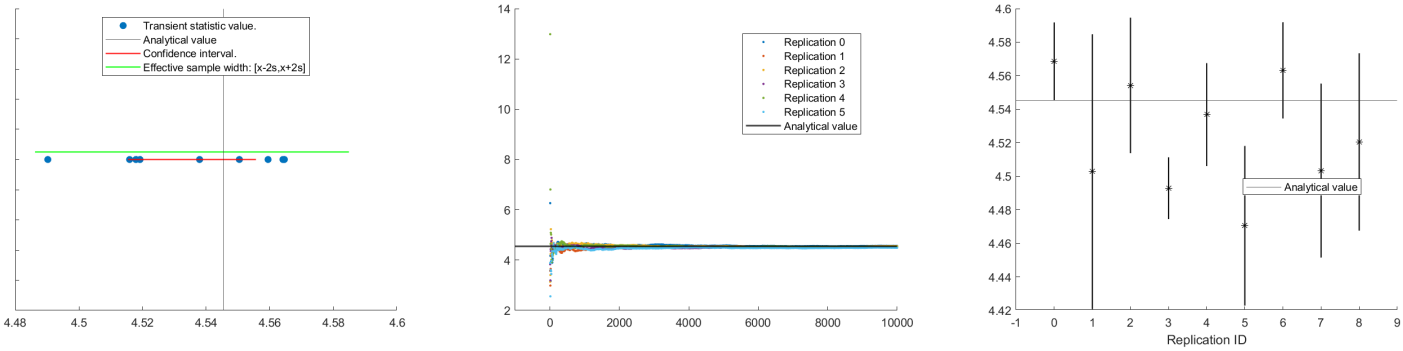


Figure 18: Cloud

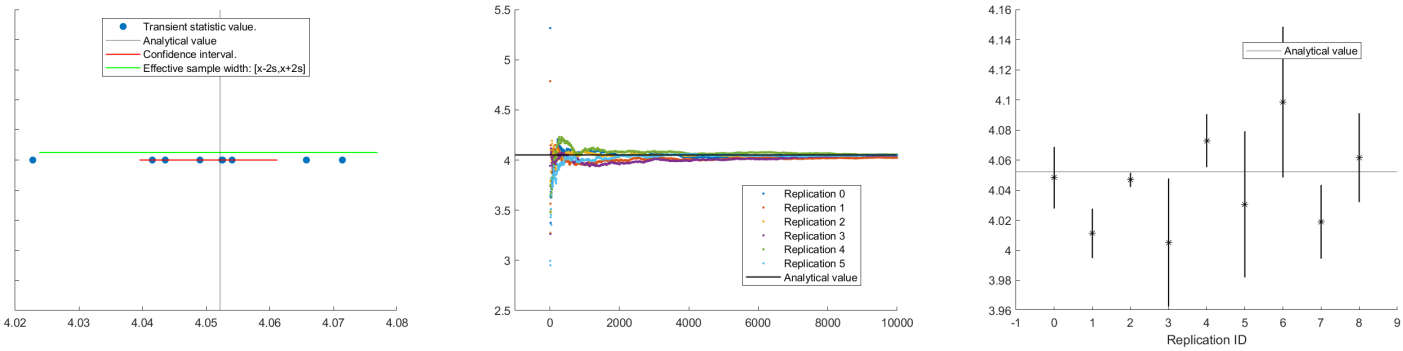


Figure 19: Global

8.1.6 Time-Average Service/Response Time

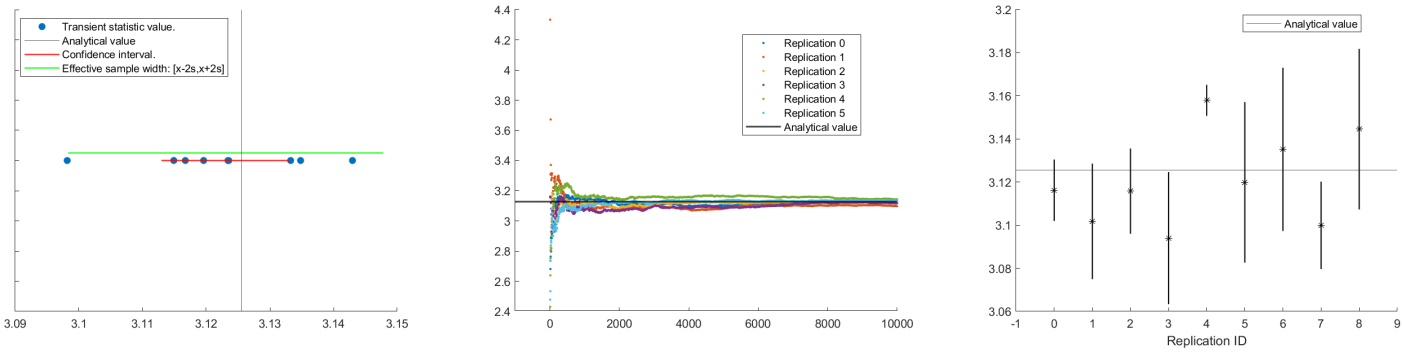


Figure 20: Cloudlet

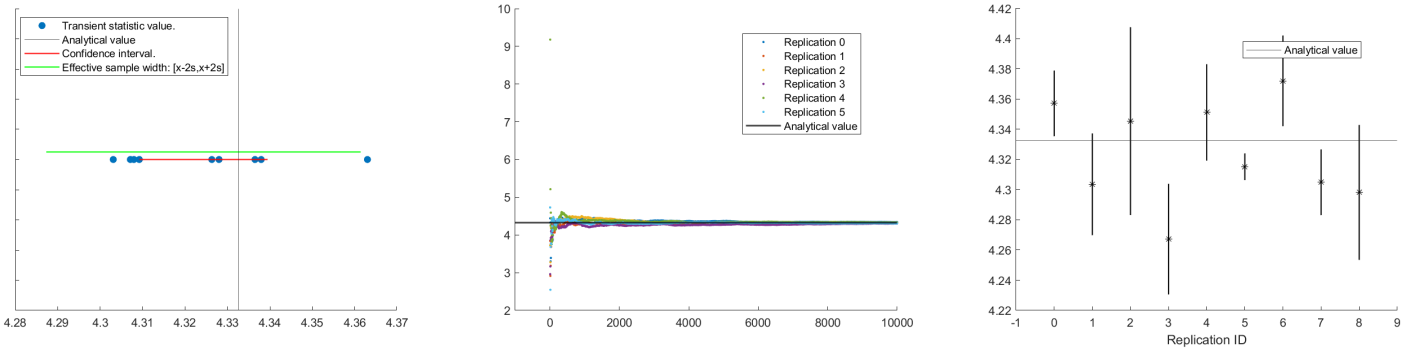


Figure 21: Cloud

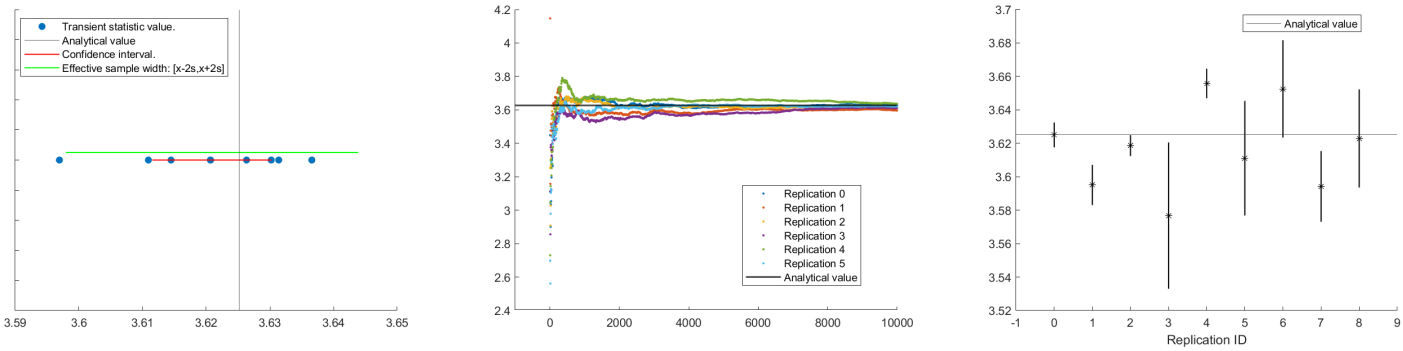


Figure 22: Global

8.1.7 Class 1 Job Throughput

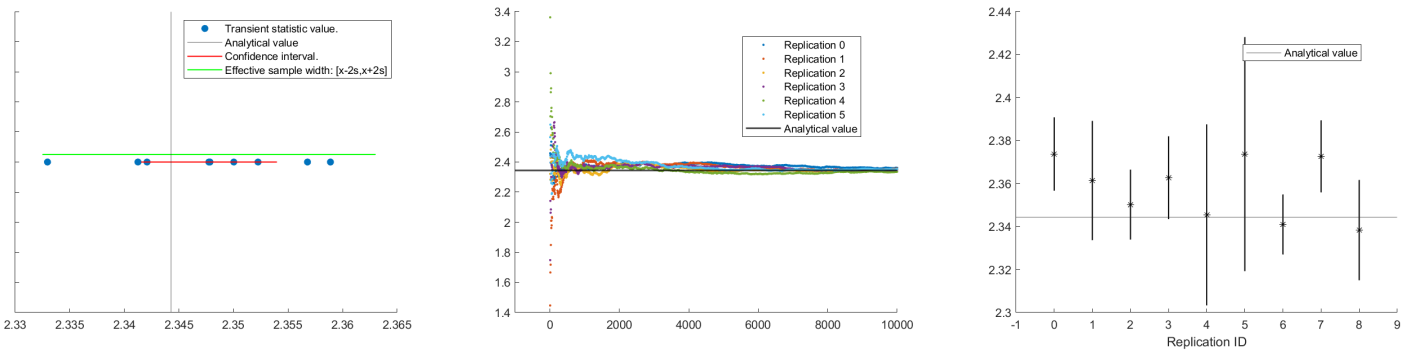


Figure 23: Cloudlet

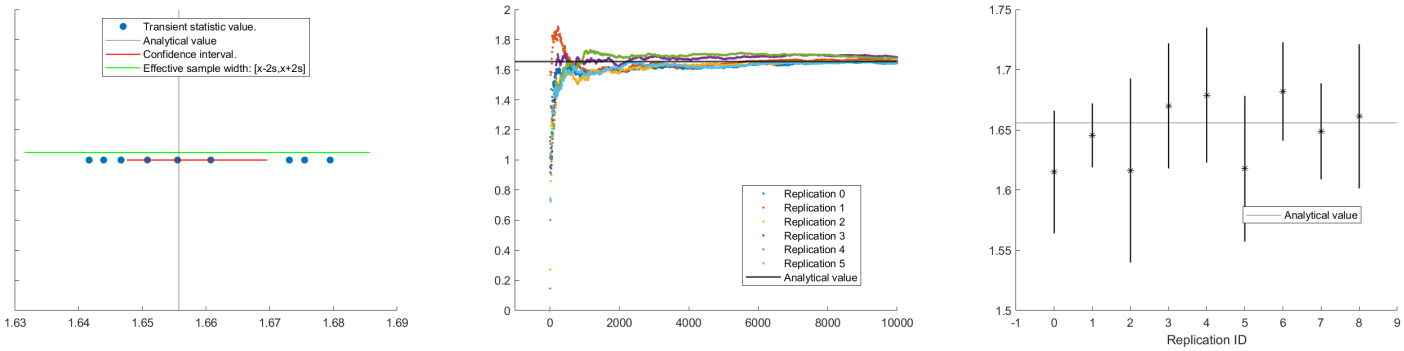


Figure 24: Cloud

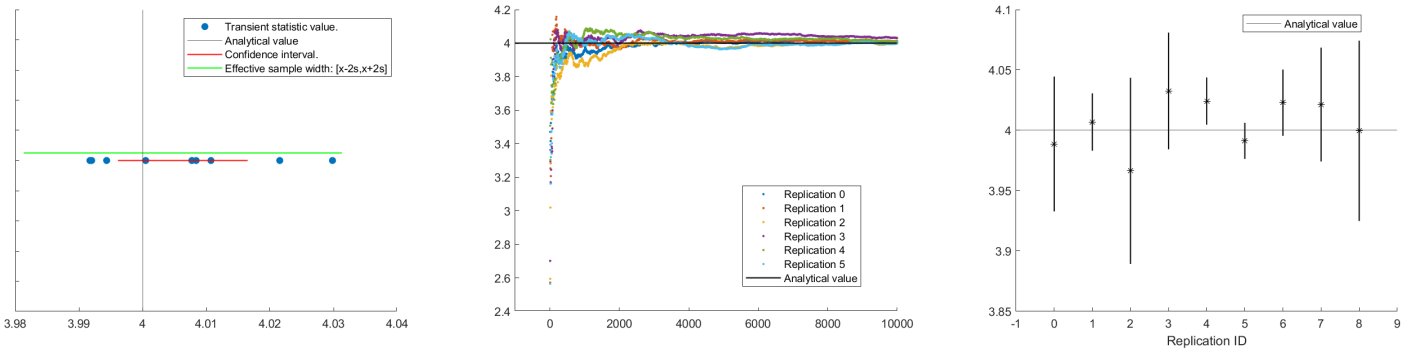


Figure 25: Global

8.1.8 Class 2 Job Throughput

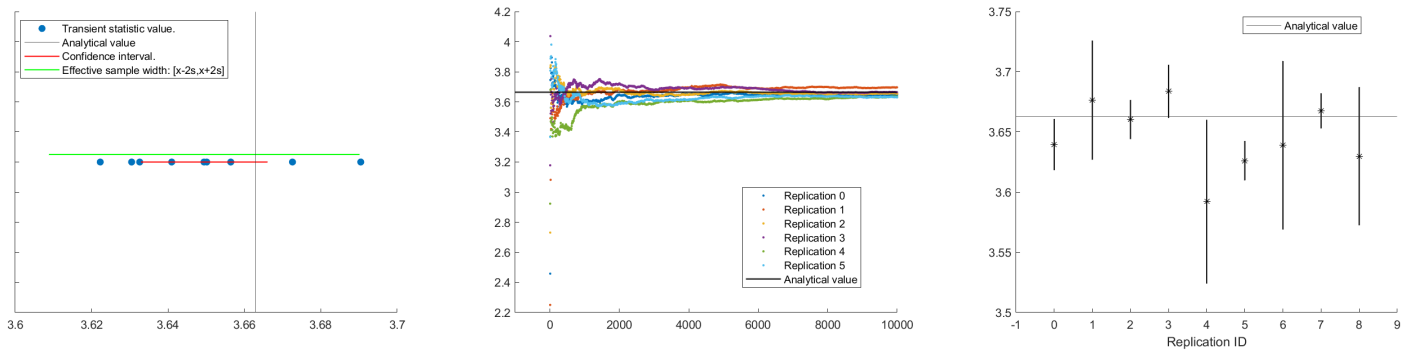


Figure 26: Cloudlet

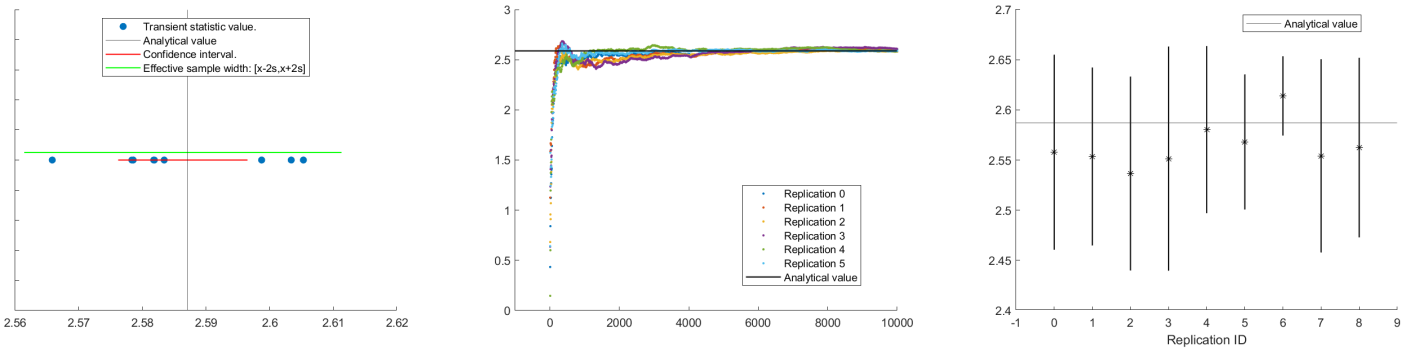


Figure 27: Cloud

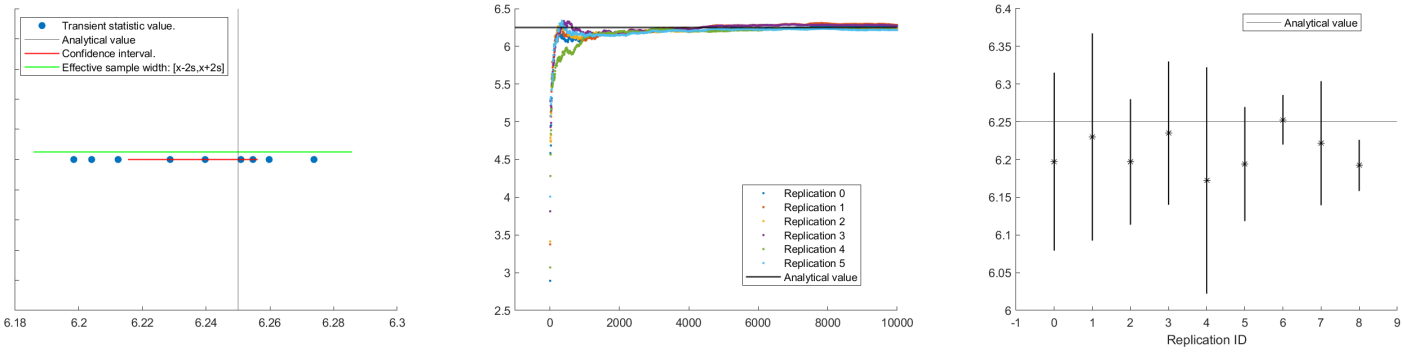


Figure 28: Global

8.1.9 Throughput

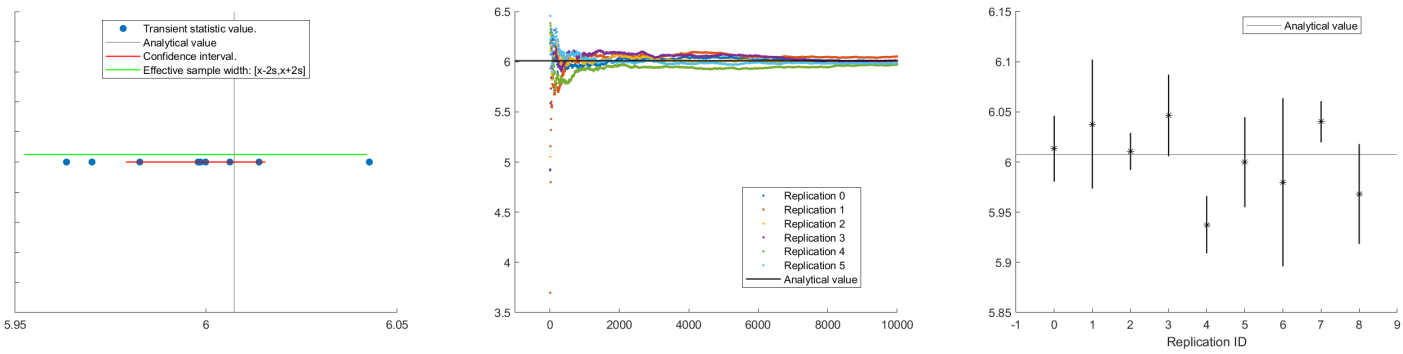


Figure 29: Cloudlet

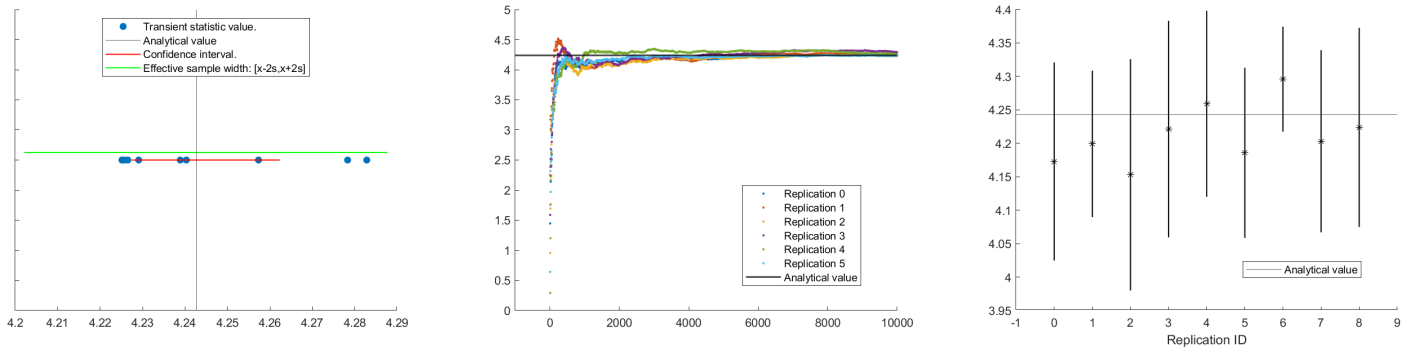


Figure 30: Cloud

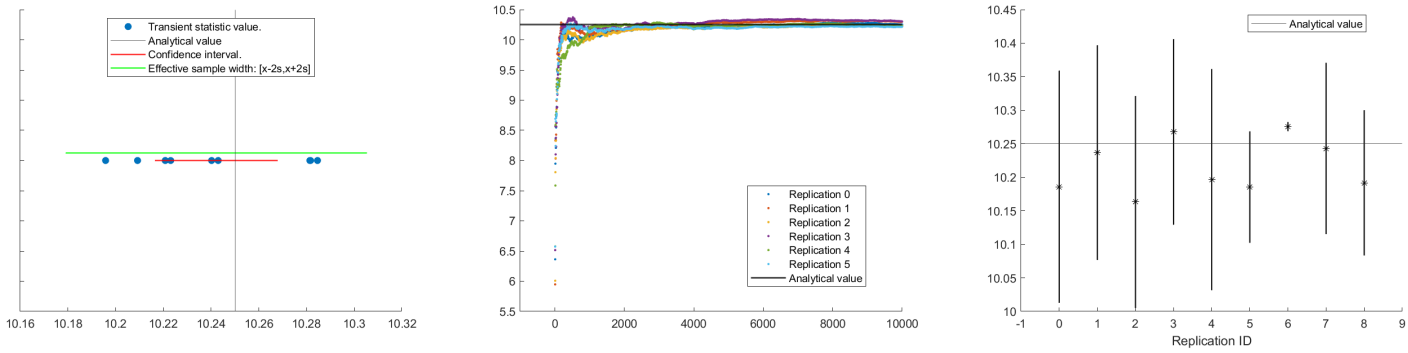


Figure 31: Global

8.2 Algorithm 2 simulation results

8.2.1 Time-Average Class 1 Job Population

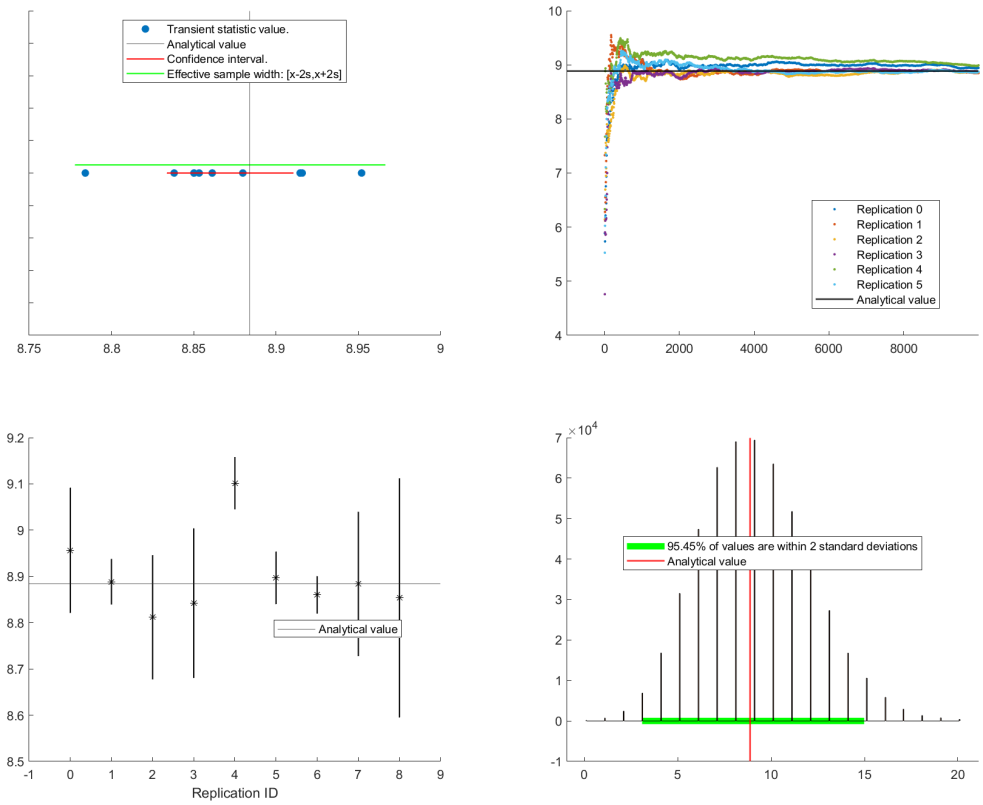


Figure 32: Cloudlet

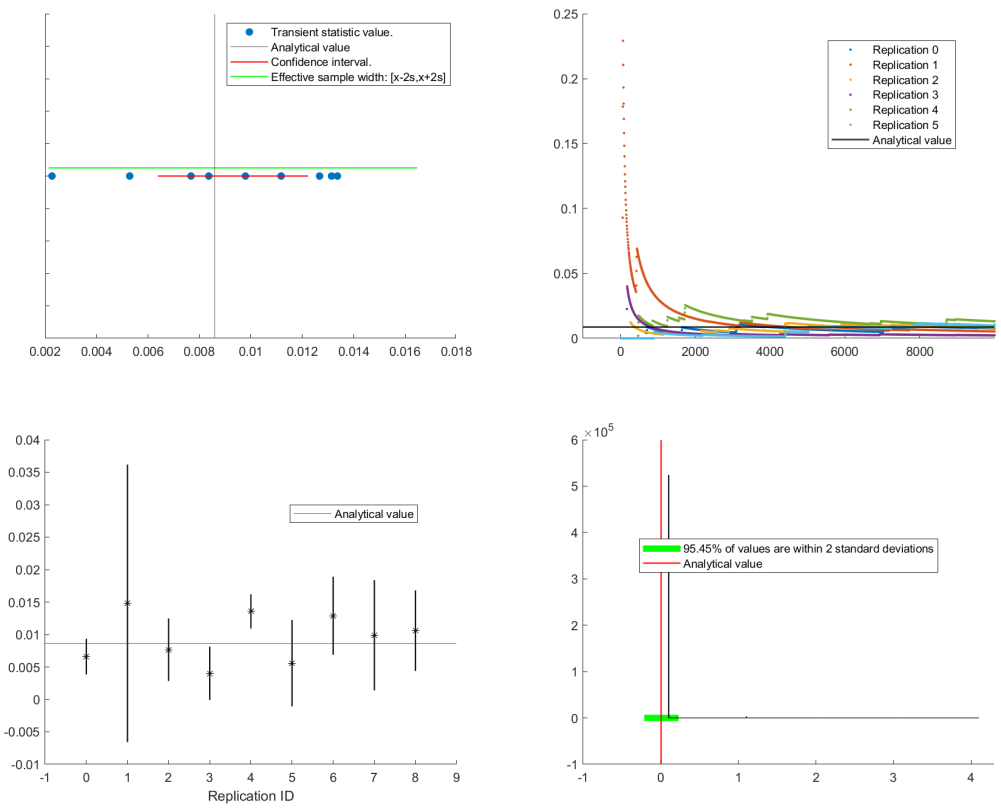


Figure 33: Cloud

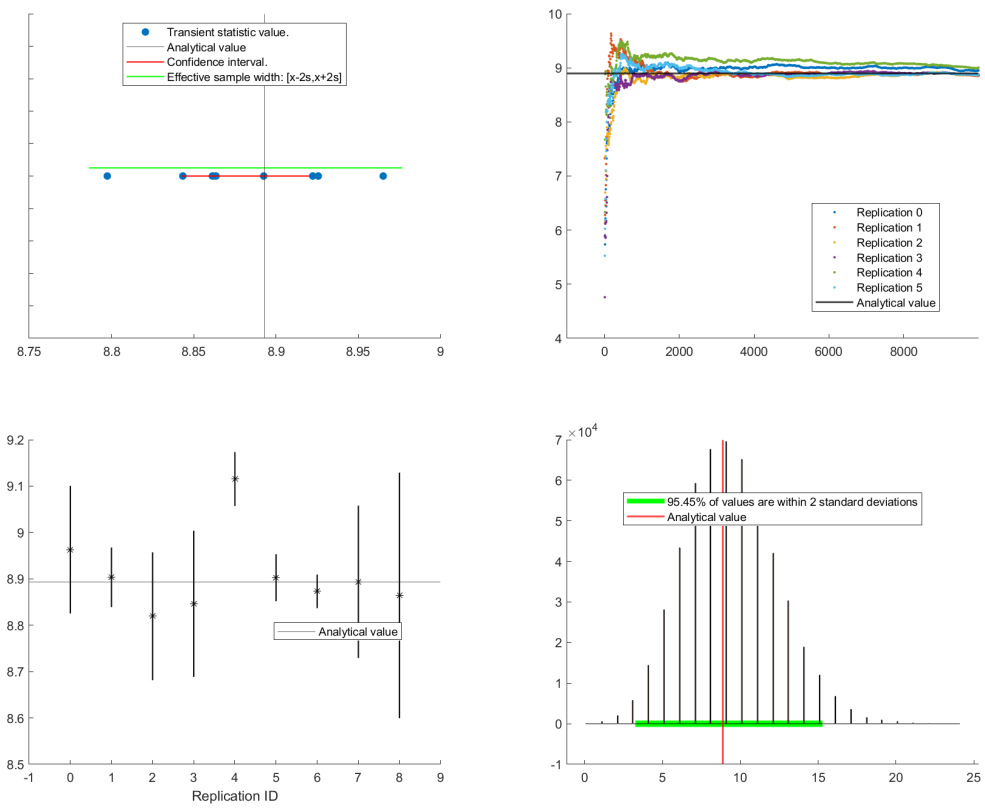


Figure 34: Global

8.2.2 Time-Average Class 2 Job Population

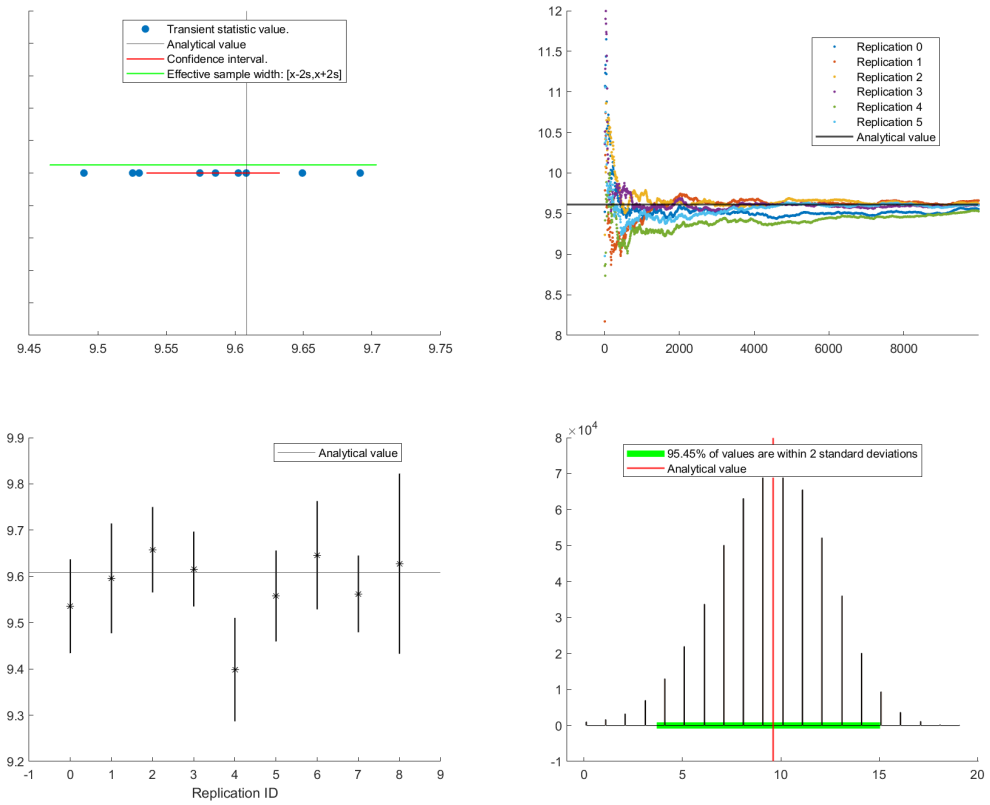


Figure 35: Cloudlet

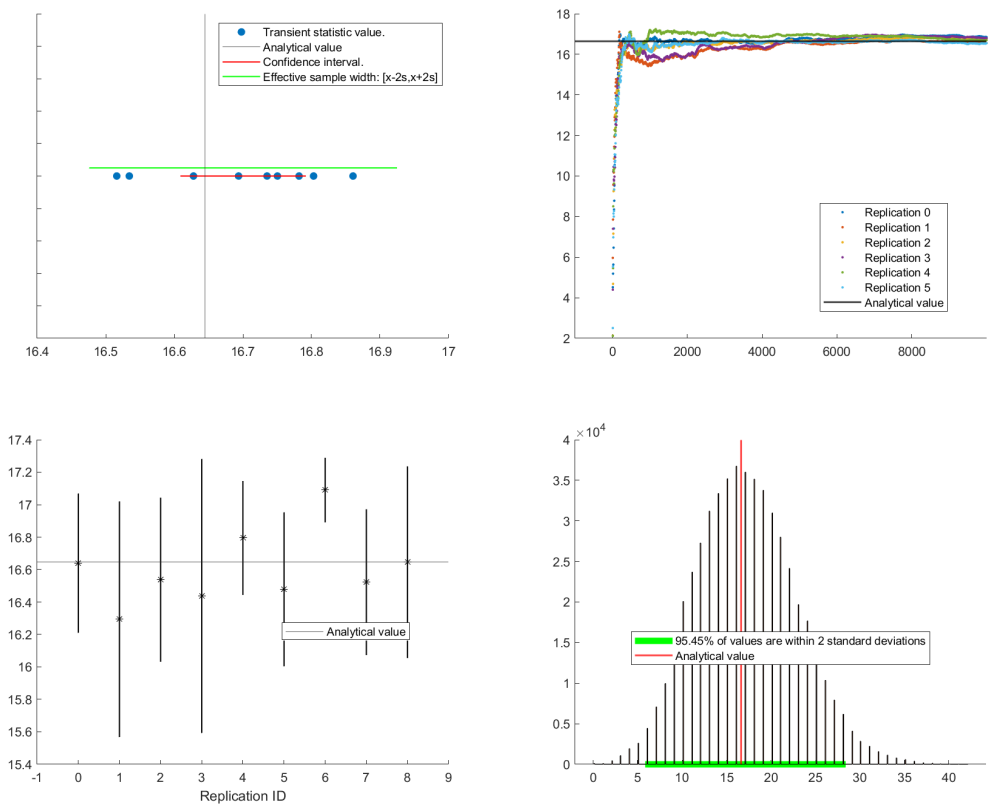


Figure 36: Cloud

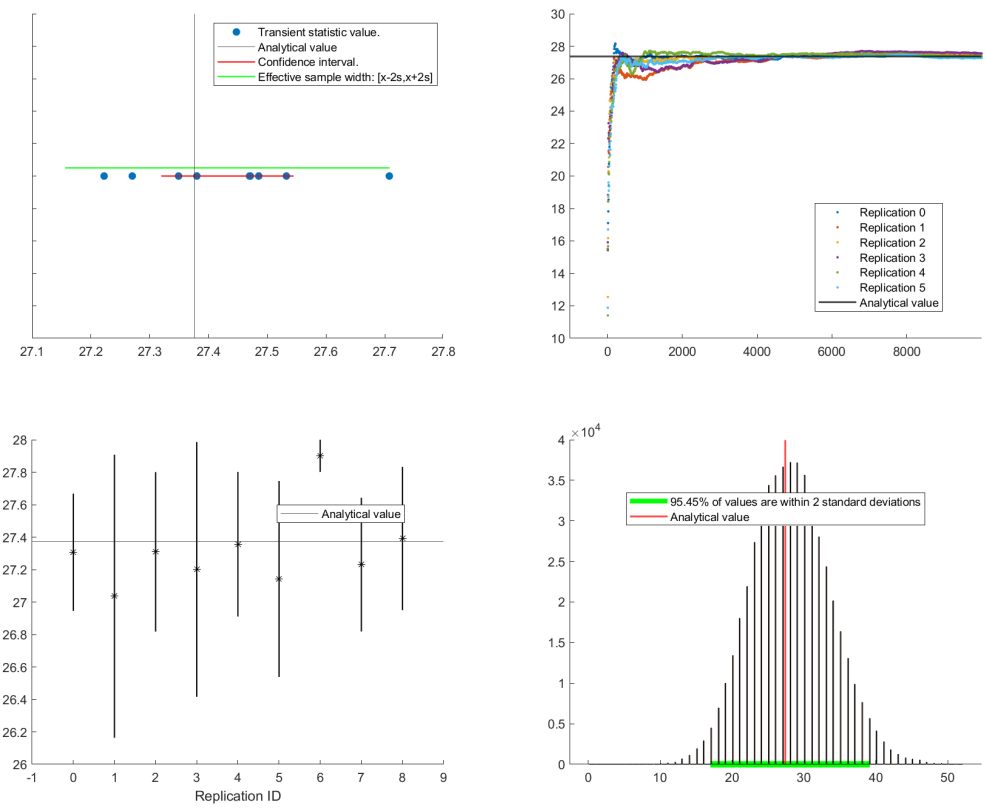


Figure 37: Global

8.2.3 Time-Average Job Population

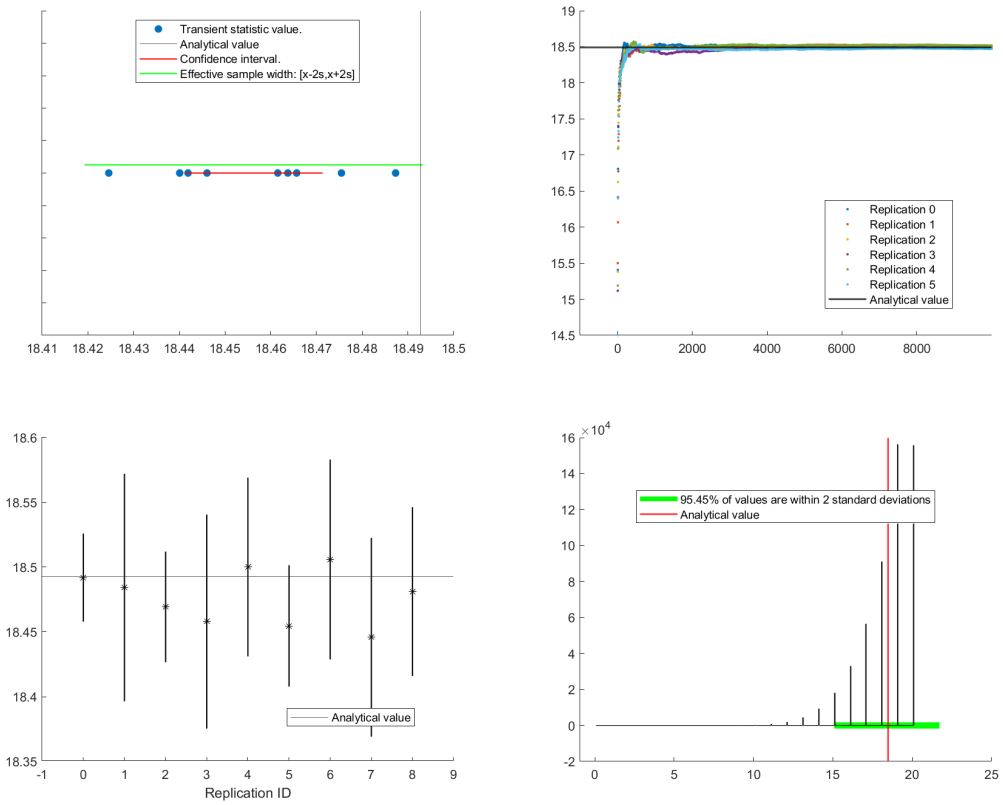


Figure 38: Cloudlet

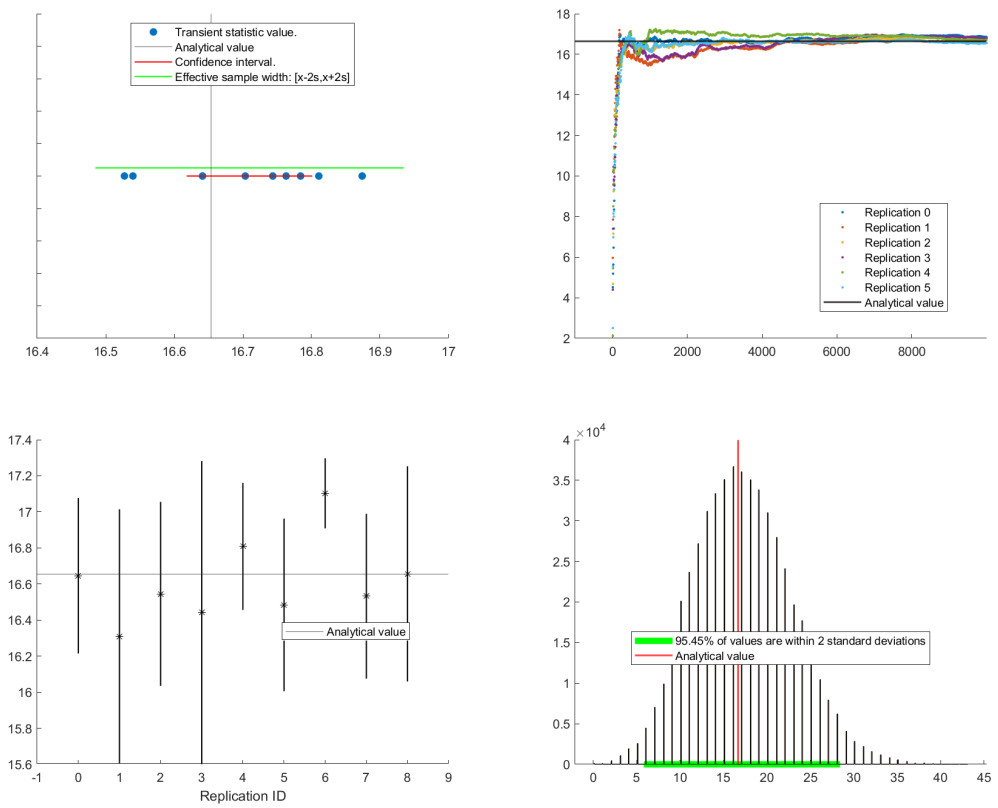


Figure 39: Cloud

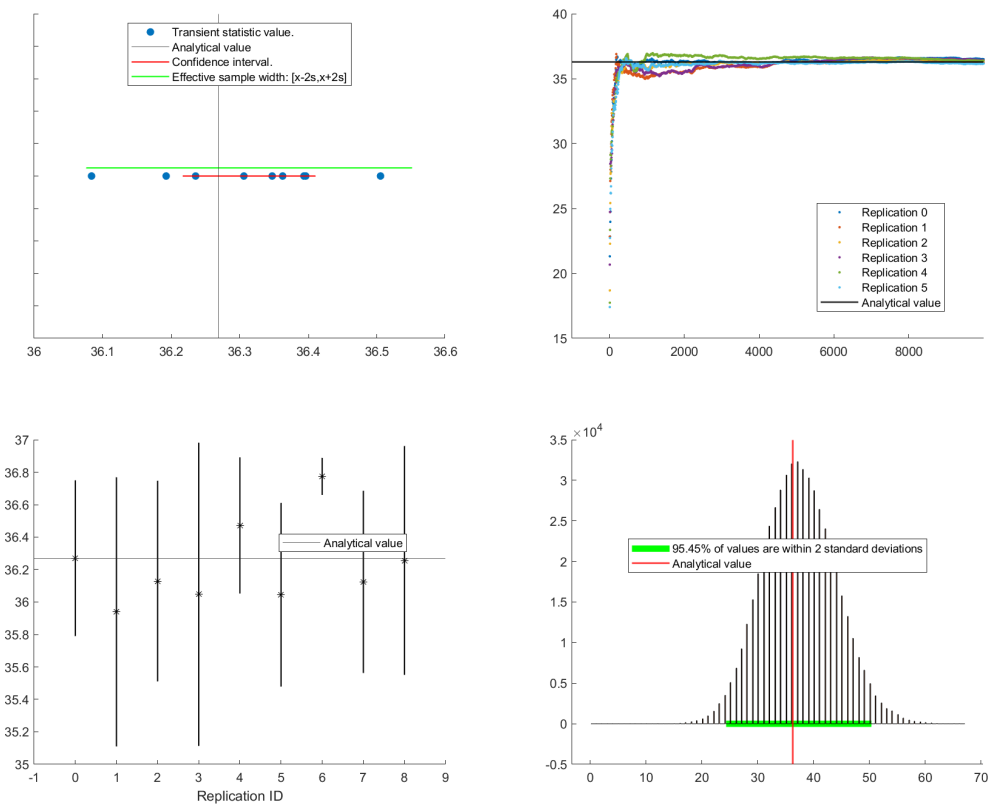


Figure 40: Global

8.2.4 Time-Average Class 1 Job Service/Response Time

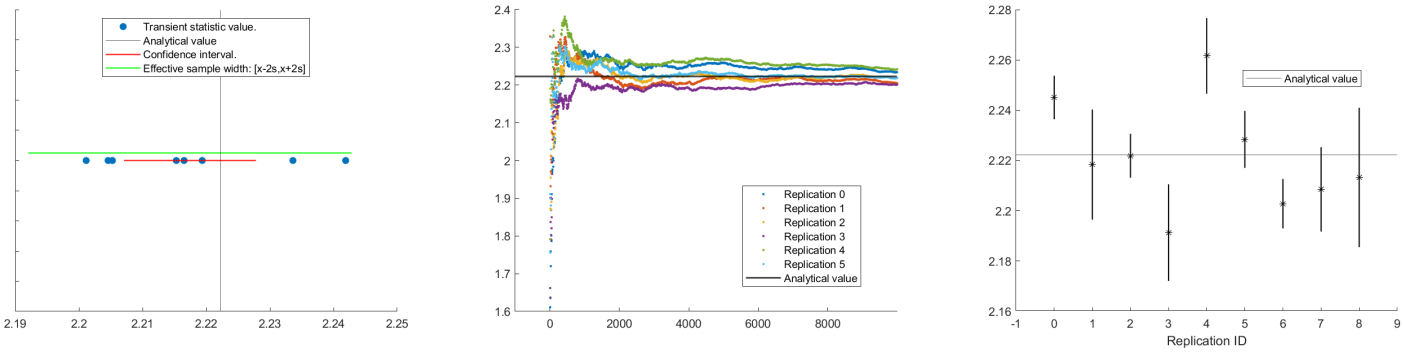


Figure 41: Cloudlet

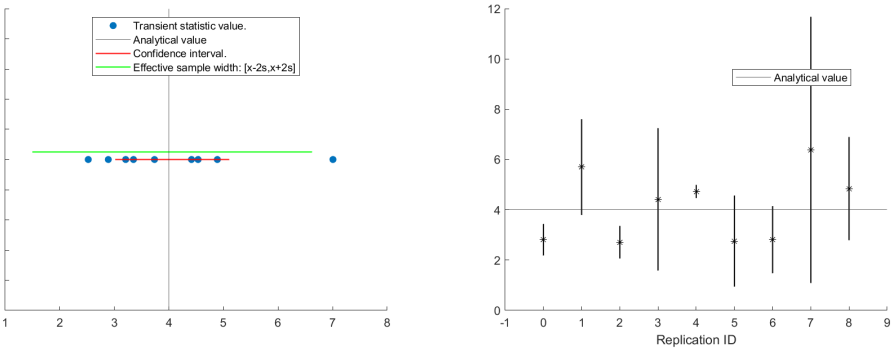


Figure 42: Cloud

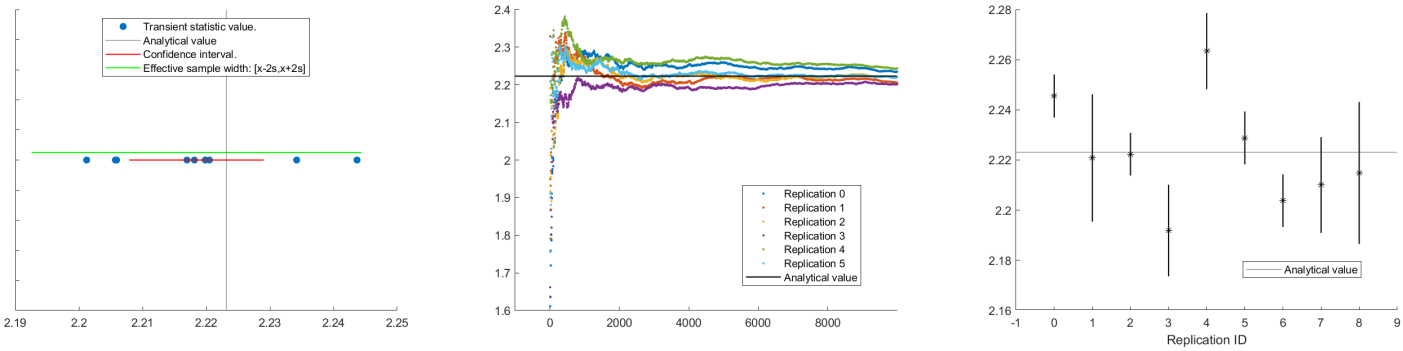


Figure 43: Global

8.2.5 Time-Average Class 2 Job Service/Response Time

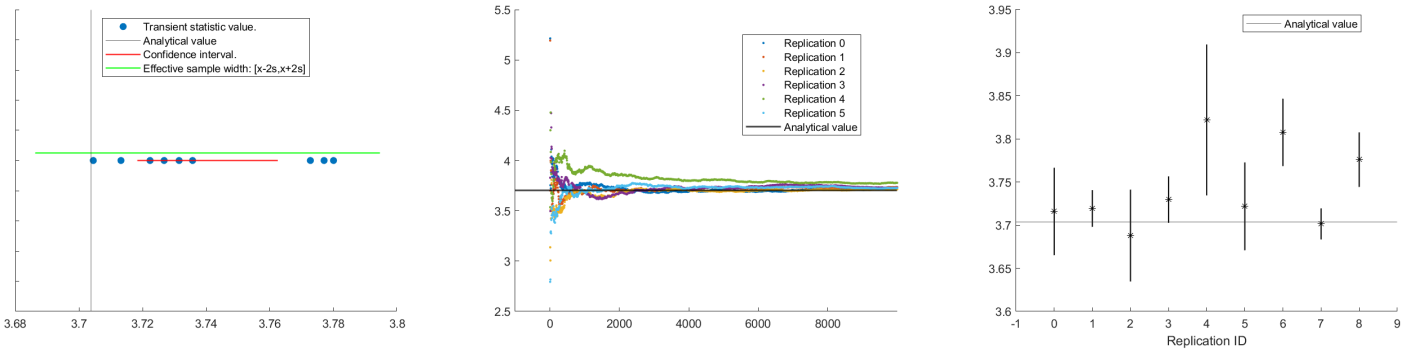


Figure 44: Cloudlet

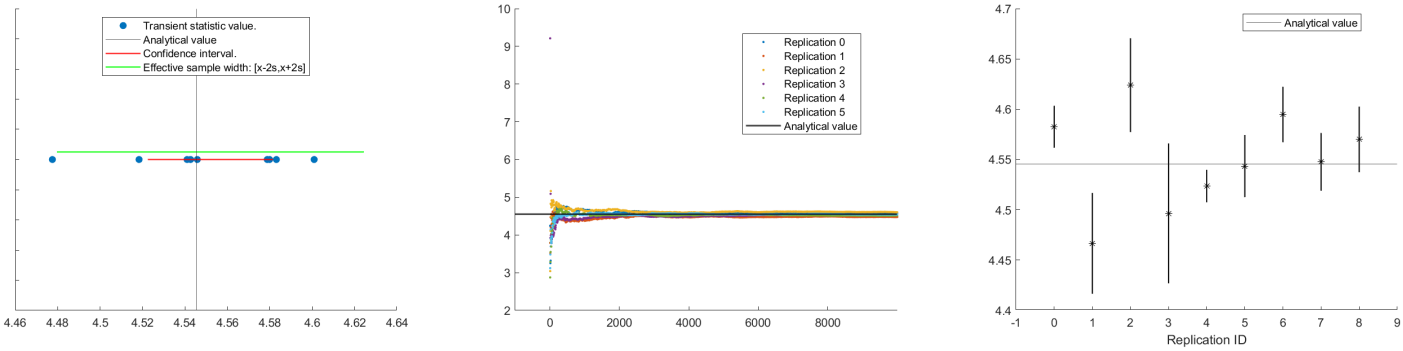


Figure 45: Cloud

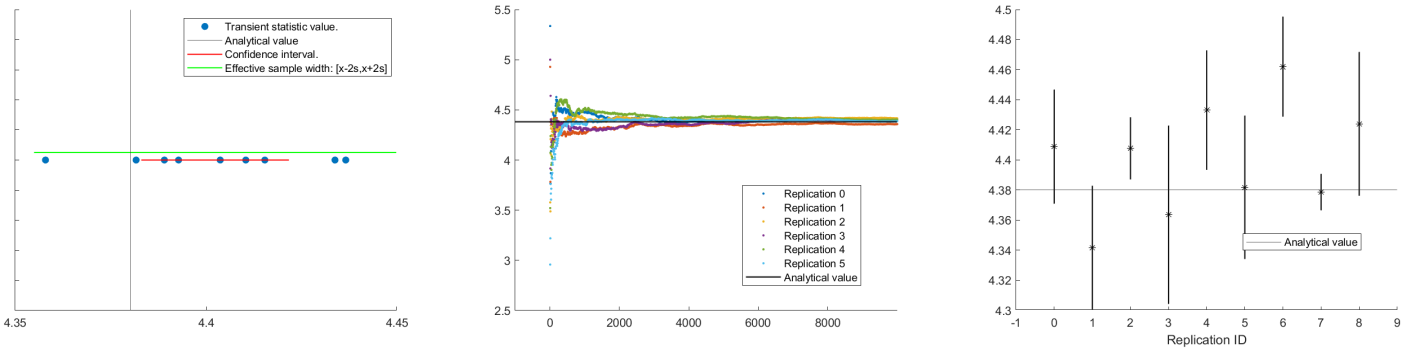


Figure 46: Global

8.2.6 Time-Average Service/Response Time

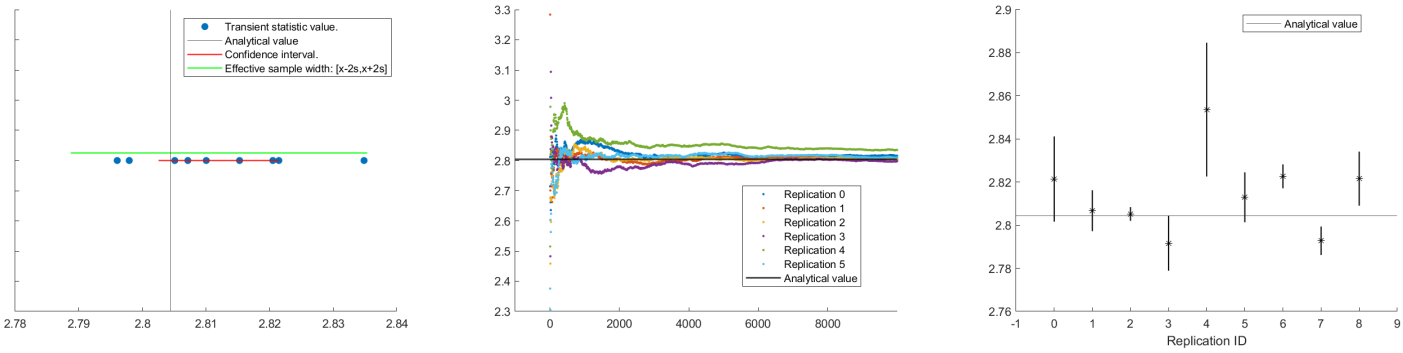


Figure 47: Cloudlet

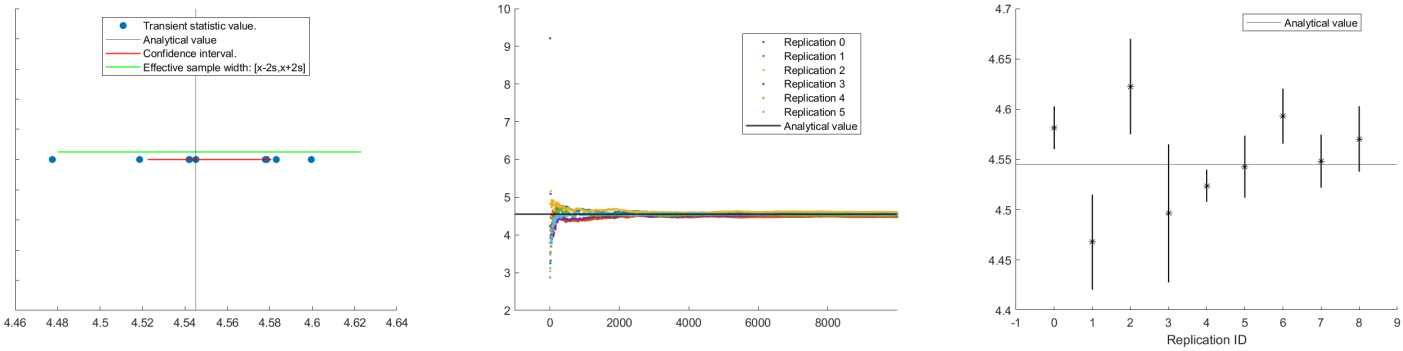


Figure 48: Cloud

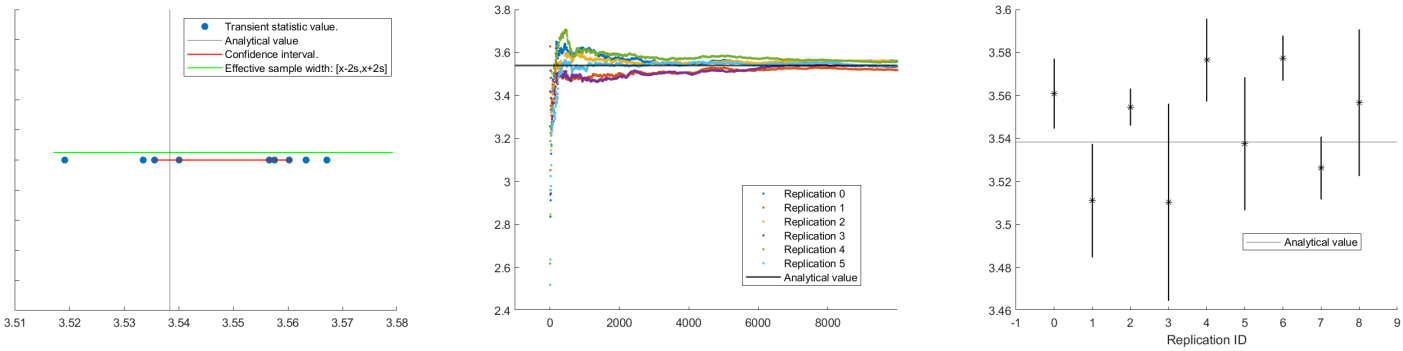


Figure 49: Global

8.2.7 Class 1 Job Throughput

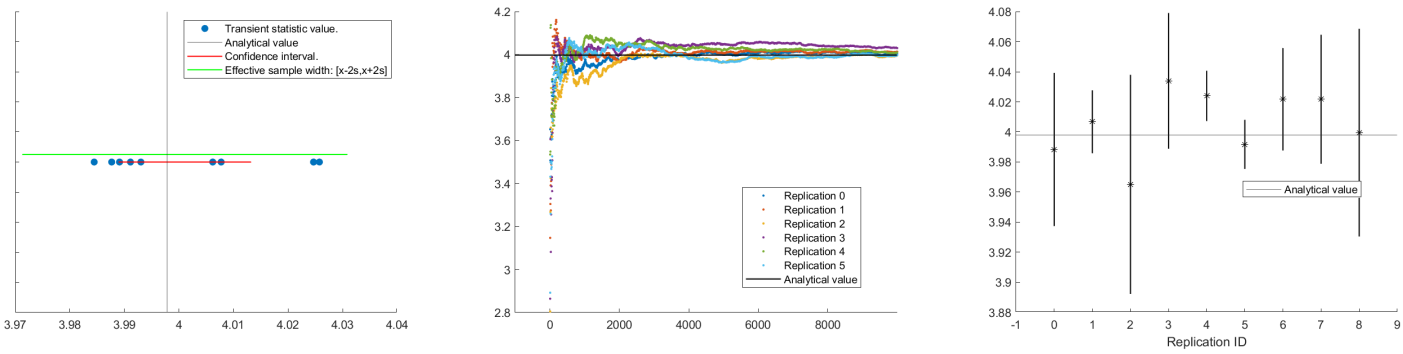


Figure 50: Cloudlet

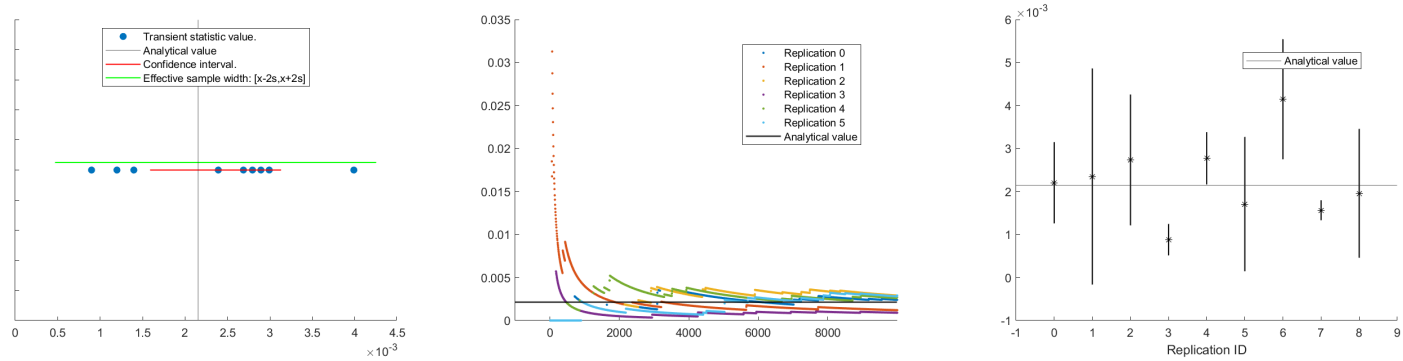


Figure 51: Cloud

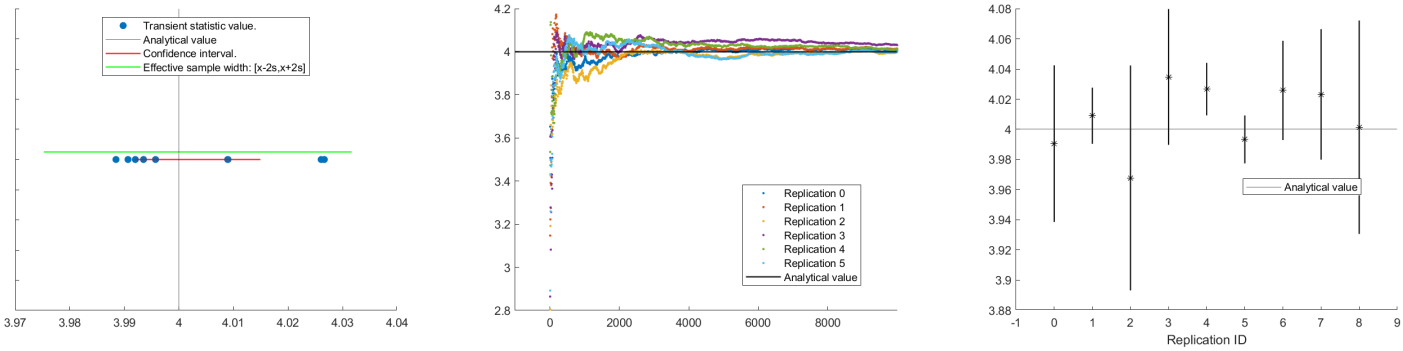


Figure 52: Global

8.2.8 Class 2 Job Throughput

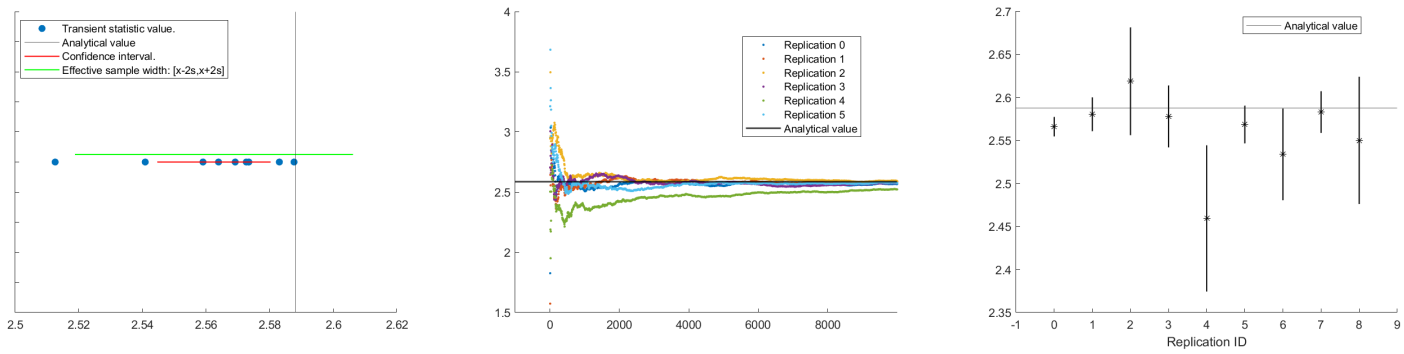


Figure 53: Cloudlet

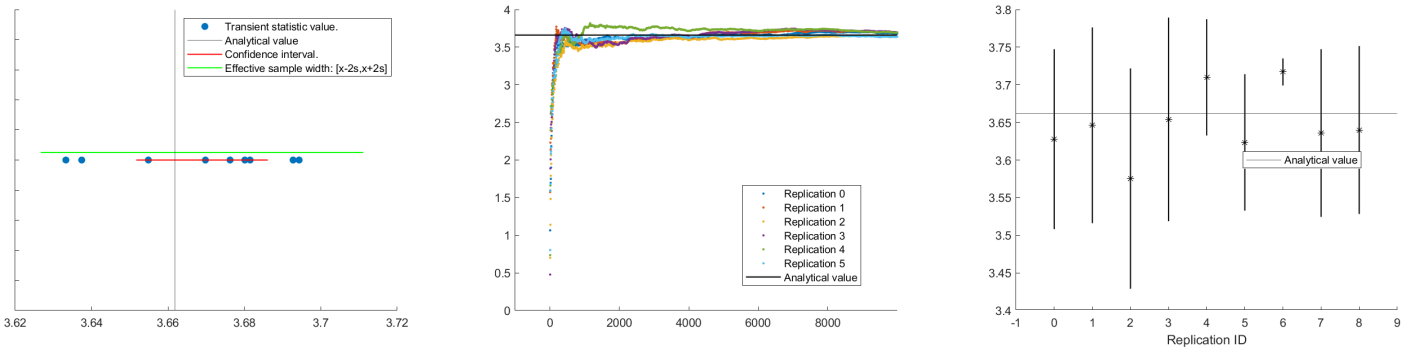


Figure 54: Cloud

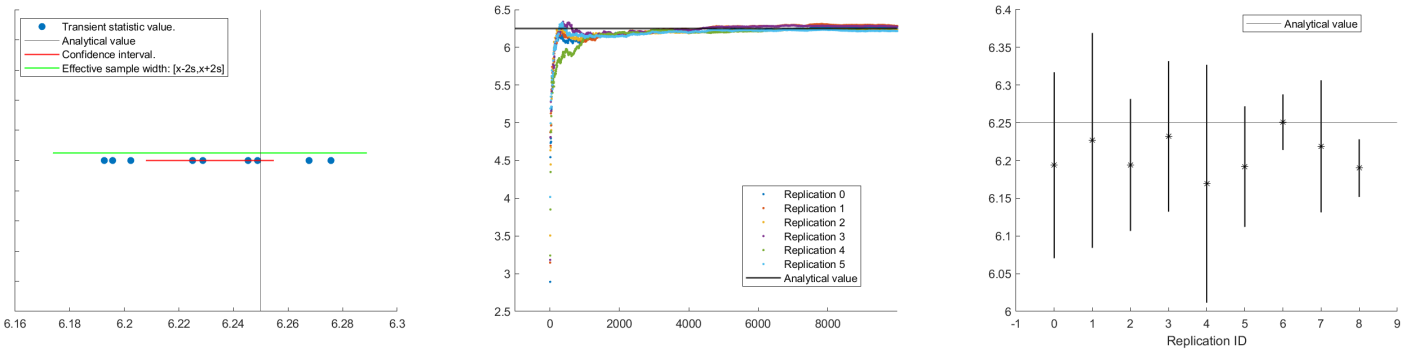


Figure 55: Global

8.2.9 Throughput

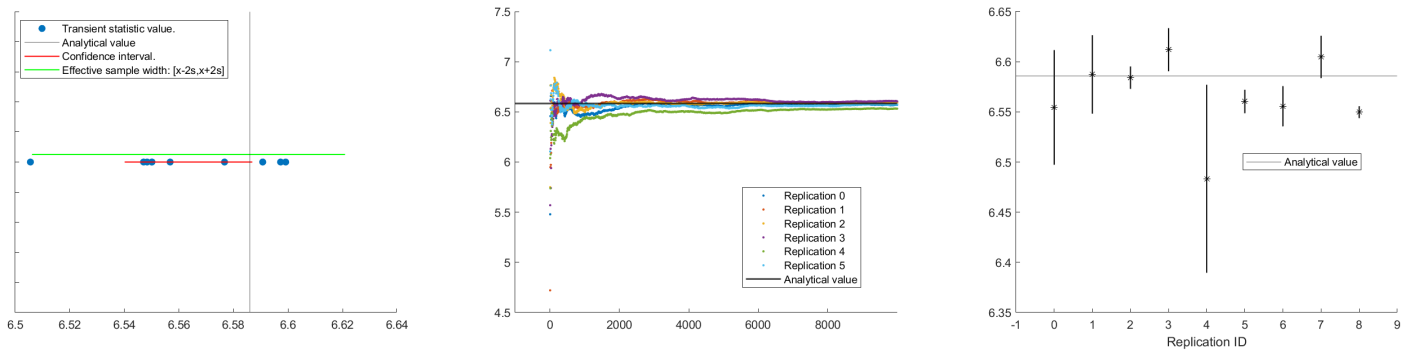


Figure 56: Cloudlet

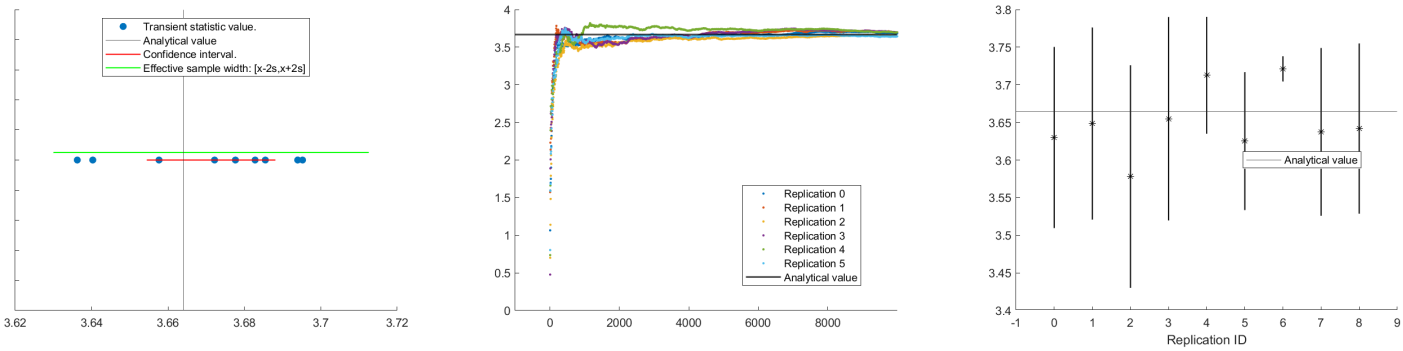


Figure 57: Cloud

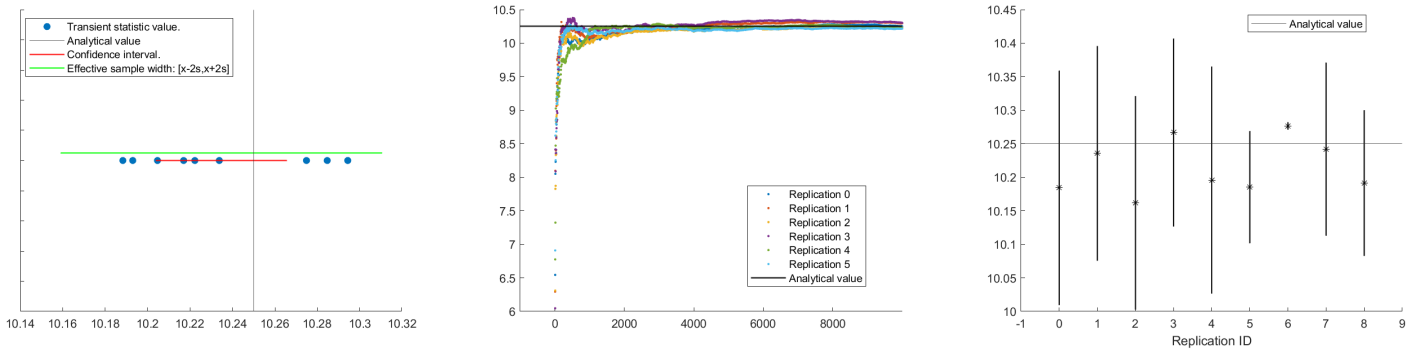


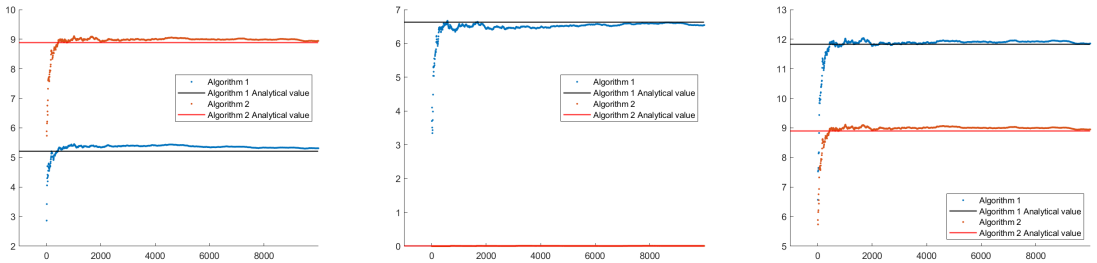
Figure 58: Global

9 Conclusions: which routing policy is better?

In this last section we will compare performance achieved by both routing policies used to build our system.

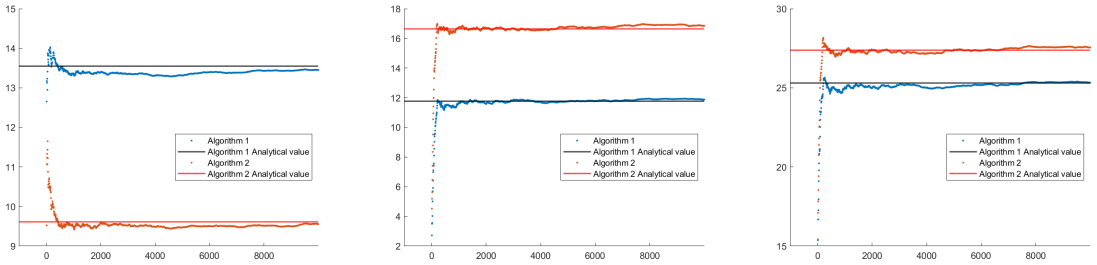
9.1 Time Average population comparison

Let's start our comparison from a time average job population point of view observing Figure 59 and Figure 60, in which are represented scatter plots about both cases according to which our system uses routing policy based on Algorithm 1 or Algorithm 2.



(a) Cloudlet (b) Cloud (c) Global

Figure 59: Time-Average Class 1 Job Population



(a) Cloudlet (b) Cloud (c) Global

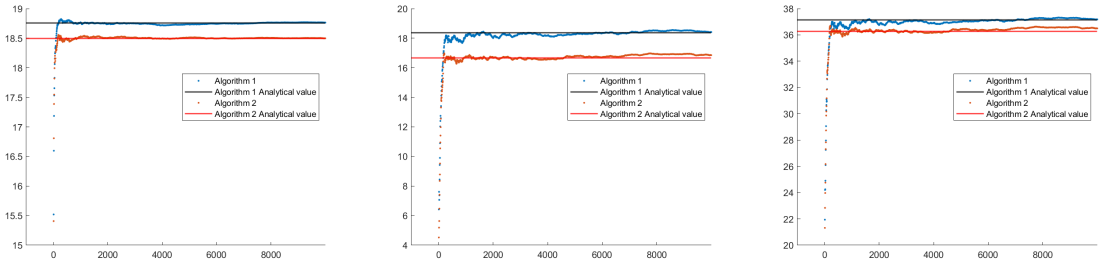
Figure 60: Time-Average Class 2 Job Population

As we expect, when we are using a routing policy based on Algorithm 2, the number of class 1 jobs in cloudlet node (Figure 59 (a)) increases a lot while decreases dramatically in reference to cloud node, which is almost empty (Figure 59 (b)).

Regarding class 2 jobs population, the situation is practically reversed respect to previous case.

In general, in a system whose routing policy is based on Algorithm 2, time-average number of jobs is less than the other case because that routing policy is able to make better use of the performance achieved by cloudlet node when in it class 1 jobs are running. To convince yourself about that, observe Figure 61.

Obviously all considerations made so far are confirmed by the analytical results.



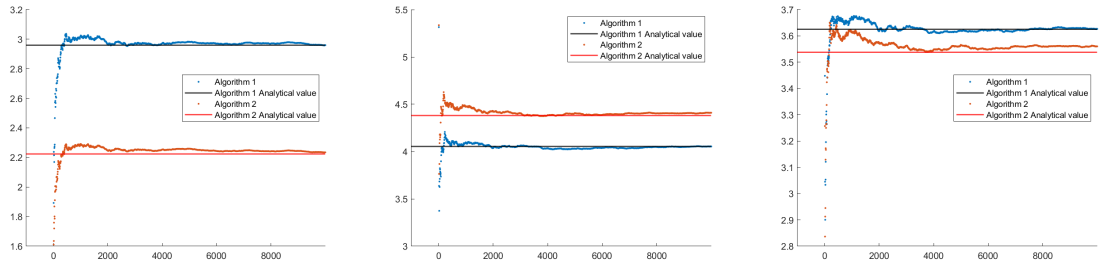
(a) Cloudlet (b) Cloud (c) Global

Figure 61: Time-Average Job Population

9.2 Time Average service/response time comparison

Since class 1 jobs execution on the cloudlet is just more convenient respect to class 2 jobs, mobile devices users experience a lower delay when routing policy is based on Algorithm 2 (Figure 62 (a)).

This statement is true because that algorithm gives more priority to class 1 jobs in order to be performed in cloudlet node respect to the other class, whose jobs experience, instead, a longer time average response time due to interruptions and set-up time required for restating (Figure 62 (b)).



(a) Class 1 Jobs (b) Class 2 Jobs (c) Both Classes

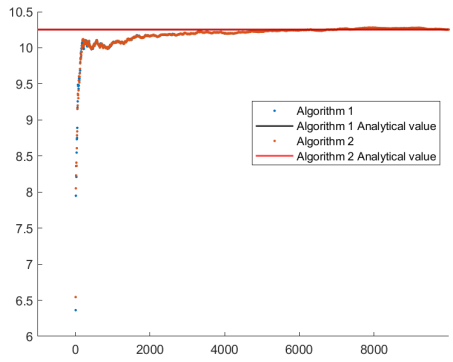
Figure 62: Global Time-Average service/response time

In general, when we are using a routing policy based on Algorithm 2, according to analytical results and data displayed in Figure 62 (c), users can experience a lower average response time.

9.3 Throughput comparison

In previous subsection we have demonstrated that both average population and response time are lower when we are using a routing policy based on Algorithm 2. What can we say about throughput?

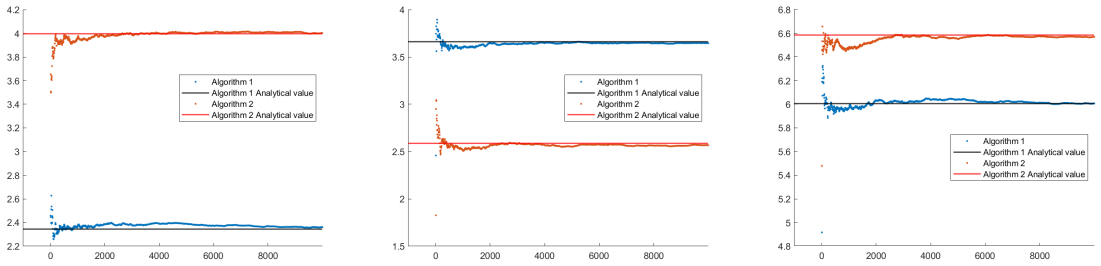
Is normal to expect same global throughout when we are use both routing policies; obviously there are no differences because, due of system stability, throughput depends on arrival rates only; changing in routing policy cannot alter global system throughput; To realize it, just look Figure 63.



(a)

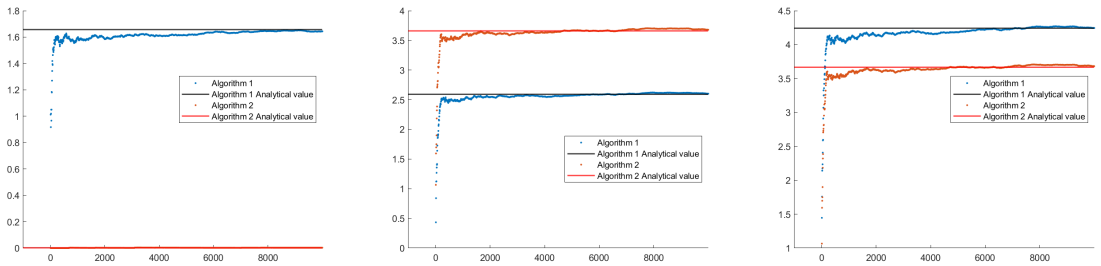
Figure 63: Global Throughput

However, per-component and per-class throughout values aren't the same, because different routing policies give rise to different per-component and per-class arrival rates. Give a look to Figure 64 and fig:CloudThroughput in order to see differences.



(a) Class 1 Jobs (b) Class 2 Jobs (c) Both Classes

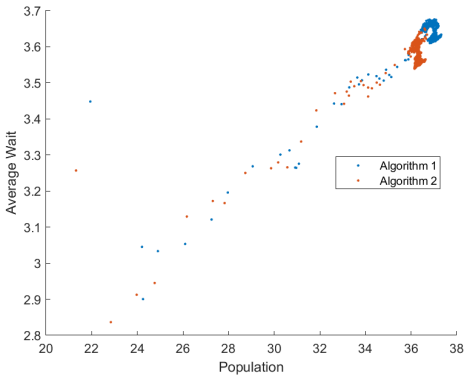
Figure 64: Cloudlet Throughput



(a) Class 1 Jobs (b) Class 2 Jobs (c) Both Classes

Figure 65: Cloud Throughput

9.4 Other comparison



(a)

Figure 66: Global Time-Average service/response time trend over Global Time-Average Jobs Population