1 Goal

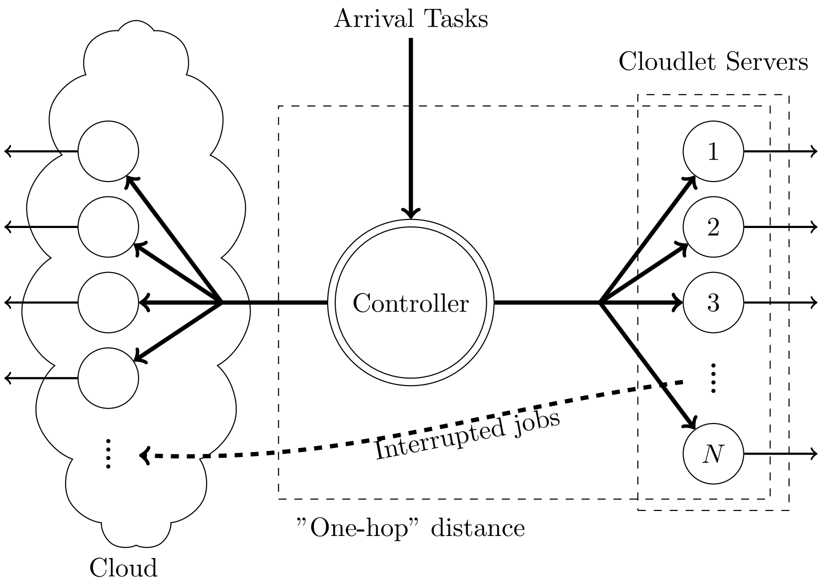
2 Conceptual Model

Let's start with the *second* step of a typical discrete-event simulation: the **conceptual model** building.

First of all our system represents a network of nodes to which some application, running in set of external mobile devices, send their tasks, called **jobs**, because of performance or energy saving reasons.

Not all job sent to system are the same, in fact we make distinction between two types of task, denoted as **class 1 job** and **class two job**; differences betweens them are described in following sections.

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 an arriving job is sent, if possible, firstly to the nearest node.



To be more precise, our system is made up of a set of nodes described below:

Cloudlet It represents an *edge cloud server* which, in addition to being located at "*one-hop*" distance from mobile devices users, and therefore nearest to them, is able to guarantee absence of interferences among tasks allocated to it as long as their number does not exceed a given threshold N. From our conceptual model point of view, cloudlet represents a **fixed-capacity multi-server service node with any queue**.

Its capacity, that is the maximum possible number of jobs in it, is fixed to N

Cloud It represents a *remote cloud server* which, although it **suffers for greater network delay**, due of its high distance from mobile devices, has virtually **unlimited resources** so that he can to process any number of tasks allocated to it. Consequently we had modelled this subsystem as an **unlimited-capacity multi-server service node with any queue**.

Being made up of an unlimited number of servers, its capacity is also unlimited.

Controller It represent the entry point of our system because all task are sent to it from outside in order to decide about whether an arrived job should be sent to the cloudlet or the cloud according to a given access control algorithm.

It is located on cloudlet and has the capability to precess each arriving job **instantaneously**. So that it is modelled simply as **single-server service node with no queue**.

Algorithm 1

```
1: function ACCESSCONTROLFUNCTION(arrivalJob)
2:     if ($n_1 + n_2 = N$) 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 *state variables* of our system. Due to access control algorithms characteristics, described below, we have decided to represent the state of our system as an ordered pair (n_1, n_2) where n_1 represents the number of class one job while n_2 the number of class two job both contained in cloudlet. Consequently we have decided to just ignore type and number of jobs contained in cloud node because not relevant in any access control algorithms and negligible for goal of our study.

A this point we can observe access control algorithms used by controller to make decisions which are fully described by their pseudo-code shown in Algorithm 1 and Algorithm 2.

Observe that to make its decision controller node can use only one access control algorithm at a time.

Note that Algorithm 2 use a further threshold variable to make its decision denoted by S which is less or equal than N. Remember that when a class 2 job is interrupted and sent on the cloud, a **setup time** has to be considered to restart the task on the cloud.

Algorithm 2

```
function ACCESSCONTROLFUNCTION($arrivalJob$)
    if ($arrivalJob.isClassOne$) then
        if $n_1 = N$ then
            Send $arrivalJob$ on the cloud.
        else if $n_1 + n_2 \leq S$ then
            Send $arrivalJob$ on the cloudlet.
        else if $n_2 \geq 0$ 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 ($n_1 + n_2 \geq S$) then
            Send $arrivalJob$ on the cloud.
        else
            Send $arrivalJob$ on the cloudlet.
```

3 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.[1]

3.1 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 $x \in \{cloudlet, cloud\} = X$, used, instead, to refer to a specific system's node.

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

$n^{(c)}(\tau)$	$=$	Total number of class c jobs currently running on whole system at time τ
$d^{(c)}(\tau)$	$=$	Total number of class c departed jobs at time τ
$n_x^{(c)}(\tau)$	$=$	Number of class c jobs currently running on node x at time τ
$d_x^{(c)}(\tau)$	$=$	Number of class c departed jobs from node x at time τ
$s_{x,i}^{(c)}$	$=$	Service time of class c job i served on x node
$i_{cloudlet}^{(2)}$	$=$	Number of class 2 interrupted job which were running on cloudlet node

Is $n_x^{(c)}$ the number of class c jobs running on node x. We define the **state** of our system as follow:

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

In this context, $\omega_{cloudlet}$ and ω_{cloud} are the states of cloudlet and cloud node respectively. Specifically:

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

Thus:

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

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

Obliviously is true that:

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

$$d^{(c)}(\tau) = \sum_{x \in X} d_x^{(c)}(\tau) \quad \forall \tau \in (t_0, t) \tag{5}$$

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

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

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

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

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

3.2 Statistics computing

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

Is $E[N_x](c)$ the *time-averaged number* of class c jobs in x system's node; we can compute it as follows:[2]

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

Using above equation is very simple to compute both $E[N_x]$ and $E[N]$, denoting respectively the time-averaged number of jobs running in a x system's node and time-averaged number of all jobs running in whole system, proceeding as shown below:

$$\begin{aligned} E[N_x] \quad &= \quad \sum_{c \in C} E[N_x](c) \quad \forall x \in X \\ E[N] \quad &= \quad \sum_{x \in X} E[N_x] \end{aligned} \tag{9}$$

Are $E[S_x](c)$ and $E[T_x](c)$ respectively the *time-averaged service time* and the *time-averaged response time* experienced by class c jobs in x system 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 $E[S_x](c)$:[3]

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

While, $E[S_x]$ and $E[T_x]$, respectively time-averaged service time and the time-averaged response time experienced by all jobs in a x system node, are computable using:

$$E[T_x] = E[S_x] \quad = \quad \frac{d_x^{(1)}(t)}{d_x^{(1)}(t) + d_x^{(2)}(t)} E[S_x](1) + \frac{d_x^{(2)}(t)}{d_x^{(1)}(t) + d_x^{(2)}(t)} E[S_x](2) \quad \forall x \in X \tag{11}$$

[2] *Ibid.*, 19

[3] *Ibid.*, 17

3.3 Events

Having to built next-event simulation model, let's look to all possible **events** which can occurs during our simulation. We recall that, by definition, **an event is an occurrence that may change the state of the system**, therefore our system's state variables can only when an event occurs[4]. All event capable to alter system's state are reported below:

Table 1: System's events

Event name	**Event's place**	**State variables incremented by one**	**State variables decremented by one**
Class 1 Job arrival	Cloudlet	$n^{(1)}_{\text{cloudlet}}(t)$	
	Cloud	$n^{(1)}_{\text{cloud}}(t)$	
	Controller	$n^{(1)}_{\text{system}}(t)$	
Class 2 Job arrival	Cloudlet	$n^{(2)}_{\text{cloudlet}}(t)$	
	Cloud	$n^{(2)}_{\text{cloud}}(t)$	
	Controller	$n^{(2)}_{\text{system}}(t)$	
Class 1 Job departure	Cloudlet	$d^{(1)}_{\text{cloudlet}}(t)$, $d^{(1)}_{\text{system}}(t)$	$n^{(1)}_{\text{cloudlet}}(t)$, $n^{(1)}_{\text{system}}(t)$
	Cloud	$d^{(1)}_{\text{cloud}}(t)$, $d^{(1)}_{\text{system}}(t)$	$n^{(1)}_{\text{cloud}}(t)$, $n^{(1)}_{\text{system}}(t)$.

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

- Although initial state variables can have any non-negative integer value[5], we have choosen, as common, to set $n^{(j)}_x(0) = 0$ and $d^{(j)}_x(0) = 0$ setting, in this way, initial system status as **idle**.
- As a consequence of the previous point, first event must be either a **Class 1 Job arrival** or a **Class 2 Job arrival** both 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.

[4]See. 187
[5]See. pag 190

4 Computational Model

Let's start now description about 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.

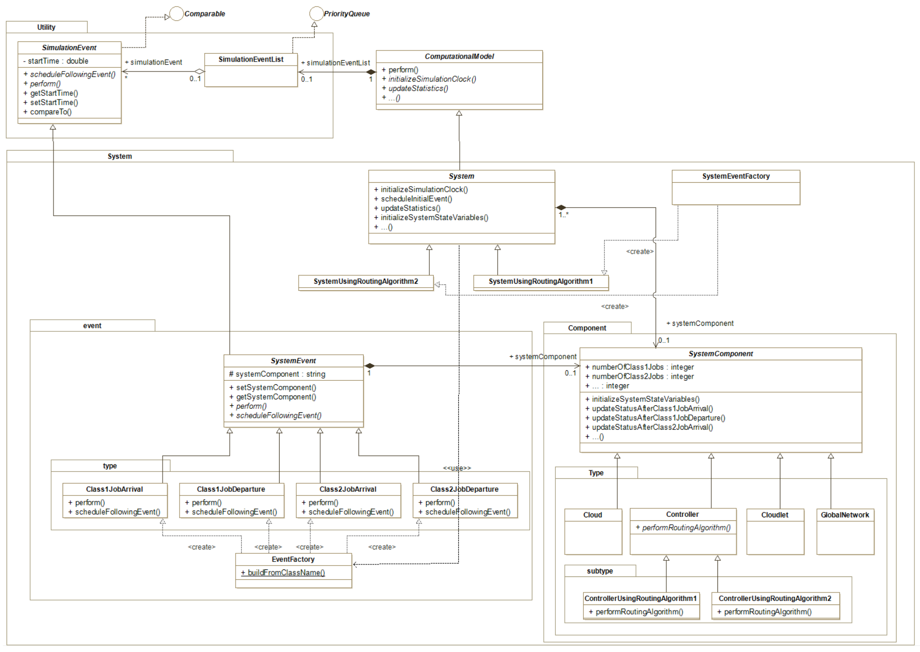


Figure 1: Class diagram showing main classes and methods only.

4.1 Events implementation

In order to properly build discrete-event simulation models using *next-event* approach we need to understand how all system's events, seen in previous section, are implemented and managed.

As shown in figure 1, a *generic* system's event has been implemented and represented using an *abstract* Java class called `SimulationEvent` which has two main features:

- Has only one `double` type field, called `StartTime`, used to hold instant of time according to which an event occurs.

- Defines interfaces of two very important methods, used by the next-event simulation algorithm implemented in `ComputationalModel` class, which we will fully describe in next section, called `perform()`, used to update system's state variables when an event occurs, and `scheduleFollowing-Event()`, which is used, instead, to inserts into an *event list* events that follows the current one.

Obviously exact methods behaviour depends strictly by which class implements it, however is 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, `ComputationalModel` doesn't need, and doesn't have to, know any specific system event information.

System-specific event are instead represented by instances of `SystemEvent` class, partially shown in listing 1. Any `SystemEvent` instance, knowing knowledge about which system's component is involved during an event occurrence, can, implementing `perform()` method, update state variable of that component taking into account its current state, without altering others parts of system. To be more precise, as you can see from the class diagram shown in figure 1, `perform()` method implementation is provided by `SystemEvent` subclasses which provide own very specific methods implementation in order to properly simulate effects of each specific system described before.

In this way, taking advantage of polymorphism pattern, every `SystemEvent` subtype instance can update properly each system's component, including whole system state, while `ComputationalModel` instance, which actively execute next-event simulation algorithm calling method implemented by them, ignores all specific information about our system.

To reduce further coupling between classes during `SystemEvent` object instantiation, a sort of factory method pattern, based on Java reflection ability, is used.

Listing 1: `SimulationEvent` class implementation.

```java
public abstract class SystemEvent extends SimulationEvent {

    protected SystemComponent systemComponent;

    public void setSystemComponent(SystemComponent
        systemComponent) {
        this.systemComponent = systemComponent;
    }

    public SystemComponent getSystemComponent() {
        return systemComponent;
    }
    ...
```

4.2 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, providing $O(log_2(n))$ time for the en-queuing and dequeuing operation and constant time for root element retrieval.[6]

Its implementation is already available in JDK through Java built-in class called `PriorityQueue<E>` which provides all method that we need. However, in order to work properly, is required that a total ordering among all possible

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

event exist. To provide it, we need all events instances implement an interface called `Comparable` providing an implementation, shown in listing 2, of a method called `compareTo`.

Remember that `compareTo` method returns a negative integer, zero, or a positive integer if an this object is less than, equal to, or greater than the specified object.[7]

Listing 2: Snippet of `SimulationEvent` class implementation

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

4.3 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, which partial implementation is reported in listing 3. Simulation clock state is represented by two `double` type fields of which one, called `currentEventTime`, is used to holding current event simulated time value, while the second, called `nextEventTime`, is used to keep track of next event simulated time value.

From software architecture point of view, as you can see by code snippet shown below, believing that multiple `SimulationClock` instance are not necessary, that class is been implemented using *singleton* pattern.

Listing 3: `SimulationClock` class implementation.

```java
public class SimulationClock {

    private static SimulationClock instance = null;

    private double currentEventTime;
    private double nextEventTime;

    private SimulationClock(){
    }

    public static SimulationClock getInstance() {
        if (instance == null)
            instance = new SimulationClock();
        return instance;
    }

    /* ... */
```

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

4.4 Next-event simulation logic

To execute a next-event simulation, we simply need to call a method, called `perform()`, using an instance of `ComputationalModel` class. As shown in listing 4, in that method has been implemented the algorithm [8] used to perform our simulation, which consists of the following steps:

Simulation initialization As the phase name suggested, this step is performed to initialize system's state variables, including simulation clock and event list, inserting into them first possible events.

As you can see from code snippet, this task is performed calling `initialize-Simulation()` method, which make a set of call to some abstract methods that have to be implemented by a concrete class which will describe next.

Process current event 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 instance field, used to holding time of occurrence of next possible event, is updated in order to compute some system metrics.

Statistic update While simulation advance from one event time to the next, calling an abstract method called `updateStatistics()`, all system's statistic metrics are updated.

Simulation result generation and print During this final step all data previously computed are elaborated using batch means approach and output data saved on file system.

Listing 4: Snippet of `perform` method

```
1   ...
2   public void perform() {
3
4           initializeSimulation();
5
6           while (!this.simulationEventList.isEmpty()) {
7
8               SimulationEvent actualEvent = this.
                    simulationEventList.poll();
9
10              if (actualEvent != null) {
11
12                  SimulationClock.getInstance().
                        setCurrentEventTime(actualEvent.
                        getStartTime());
13
14                  actualEvent.perform();
```

[8] See Algorithm 5.1.1, pag. 189

```
                actualEvent.scheduleFollowingEvent();

                SimulationEvent nextEvent = this.
                    simulationEventList.peek();

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

        produceStatisticResultsThroughBatchMeansMethod();

        printSimulationResults();
    }
...
```

4.5 System network implementation

While a very generic queues network is represented by an instance of `ComputationalModel` class, our specific system is been modelled using `System` type instances which have very important responsibilities including:

A System type instance has many role including:

- Allocates `SystemComponent` type instances used to represent each system's nodes.
- Initializes all instance fields representing a system variables of each system's node including simulated clock.
- Updates event list for future events scheduling on each system's component.

As we now

`SystemComponent` class has key role during our simulation performing because:

- Has a set of fields used to keep track of component's variable states, like number of departed jobs or current jobs population, including those used to compute time-average statistics.
- Provides a very important set of functions for status updating after event occurrences; 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.
- Contains a set of method used to computed inter-arrival and service times, necessary for future event scheduling. Obviously each concrete `SystemComponent` instance has his own mean rates to compute those values.

- Includes another set used instead to schedule future events after event occurrence, like a departure after an arrival.

  Is very important to precise that these methods are abstract and have to be implemented by each concrete implementations in order to reflect each component behaviour; In fact controller node doesn't need to schedule departure after an arrival because of his routing role.

All system's component, that is cloud, cloudlet and controller, are represented, obviously, by `Cloud`, `Cloudlet` and `Controller` classes each of which is a `SystemComponent` subclass; however we have used another subclass, called `GlobalNetwork`, in order to represent whole system keeping track of golabal state variables.

Observe that, `Controller` class is also an abstract class because required implementation of a method call `performRoutingAlgorithm()` used to perform the two required routing algorithms whenever an arrival occurs on controller node; those algorithms are provide by `ControllerUsingRoutingAlgorithm1` and `ControllerUsingRoutingAlgorithm2` classes.

4.6 Output analysis

each statistic is associate to an TransientStatistics type instance.

5 Analytical solution

In this last section we will develop an analytical solution to validate the results obtained previously through our simulations.

5.1 System based on access control Algorithm 1

Let's start with presentation of analytical solution of system based on access control algorithm 1.

In order to compute all parameters and metrics of interest associated with above-mentioned system, due to cloudlet's limited resources according to which it can accept jobs until their number does not exceed a given threshold N, is crucial compute first the **fraction of jobs that are forwarded to cloudlet and to cloud**; to do it, we must compute at first probability according to which the sum of job of each class in cloudlet system is equal to that threshold.

To determine this probability, we had modelled cloudlet with a **continuous-time Markov chain** (**CTMC**), of which you can see a graphical representationin Figure 2, where each chain's state, denoted with (n_1, n_2), is represented by the number of class 1 job, n_1, and class 2 job, n_2, present in system at a certain moment.

5.1.1 Balance equation computing

Obviously 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 [9], **remembering that limiting probabilities sum to 1** (i.e., $\sum_{j=0}^{\infty} \pi_j = 1$). These balance equations, shown in table 2, are been resolved using a very simple MATLAB script called `MATLAB_ALG1_CTMC_ResolverScript.m` in which each equation, previously generated through a Java script[10], is been resolved using a MATLAB function called `solve(eqn,var)`.[11].

[9] *Cfr.* Mor Harchol-Balter - *Performance Modeling and Design of Computer Systems* - Carnegie Mellon University, Pennsylvania, pag. 237

[10] Cfr. `CTMCResolverScriptGenerator.java` and `ResolverUsingRoutingAlgorithm1.java` files.

[11] *Cfr.* `https://www.mathworks.com/help/symbolic/solve.html`

Figure 2: Access control algorithm 1 based cloudlet's CTMC.

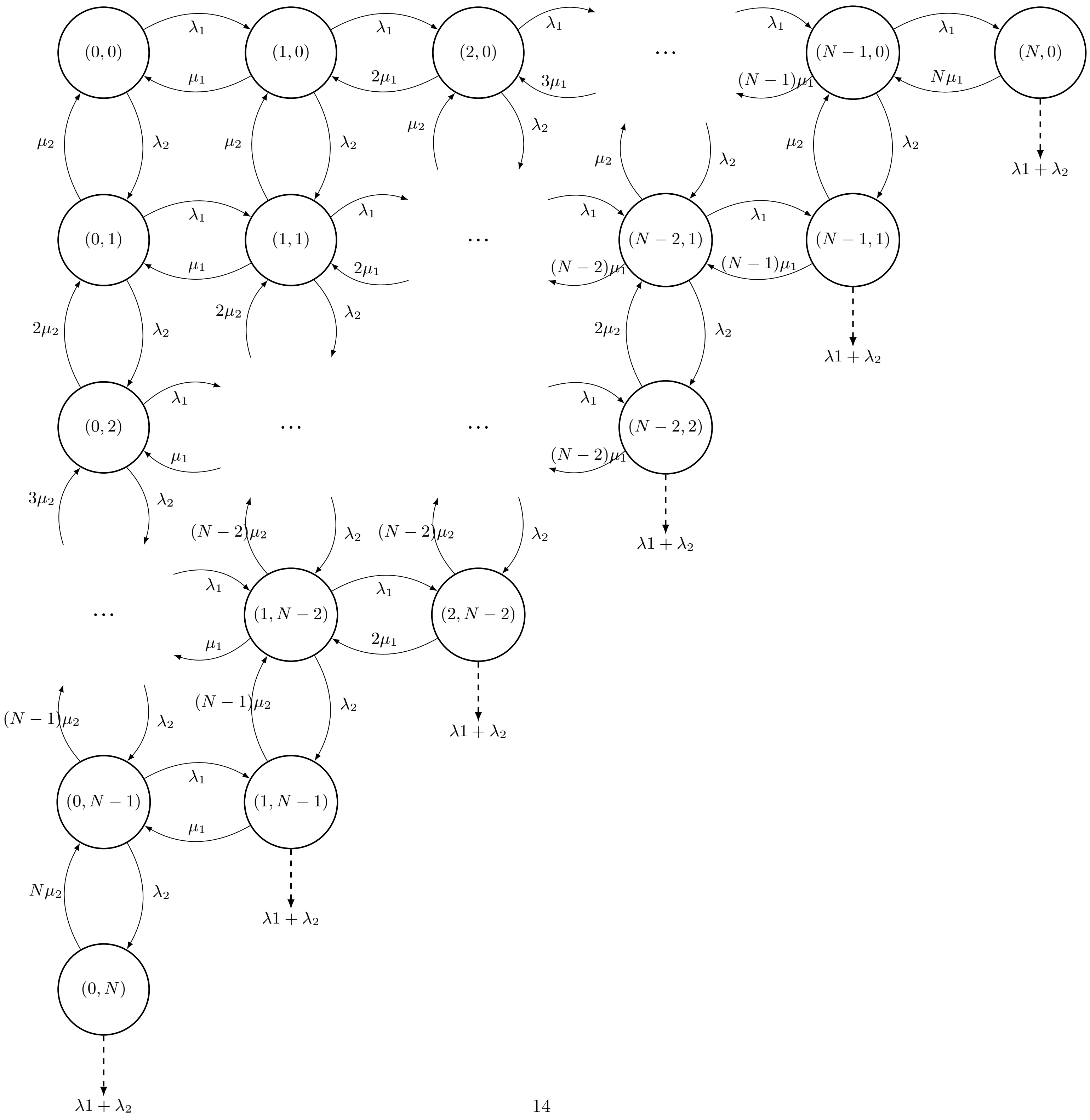


Table 2: Balance equations.

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

5.1.2 Probabilities computing

Having found the stationary probabilities, we can now find $\Pi_{\textbf{SendToCloud}}$, 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:

$$\begin{aligned}\Pi_{\text{SendToCloud}} &= P\{\text{An arrival job on controller sees } N \text{ jobs in cloudlet}\}\\ &= \text{Limiting probability that there are } N \text{ jobs in system}\\ &= \sum_{\substack{n_1,n_2\in\mathbb{N}\cap[0,N]\\ n1+n2=N}} \pi_{(n_1,n_2)}\end{aligned} \tag{12}$$

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.

$$\begin{aligned}\Pi_{\text{SendToCloudlet}} &= P\{\text{An arrival job on controller sees less than } N \text{ jobs in cloudlet}\}\\ &= 1-P\{\text{An arrival job on controller sees } N \text{ jobs in cloudlet}\}\\ &= 1-\Pi_{\text{SendToCloud}}\end{aligned} \tag{13}$$

5.1.3 Average arrival rates

Is $\lambda_i(c)$ the **total arrival rate into a system's component i of class c job**. Applying previous results, using an appropriate equation[12], we can now compute per-class average arrival rates as follow:

$$\begin{aligned}
\lambda_{\text{cloud}}(1) &= \lambda_1 \cdot \Pi_{\text{SendToCloud}} \\
\lambda_{\text{cloud}}(2) &= \lambda_2 \cdot \Pi_{\text{SendToCloud}} \\
\lambda_{\text{cloudlet}}(1) &= \lambda_1 \cdot \Pi_{\text{SendToCloudlet}} \\
\lambda_{\text{cloudlet}}(2) &= \lambda_2 \cdot \Pi_{\text{SendToCloudlet}}
\end{aligned} \tag{14}$$

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

$$\begin{aligned}
\lambda_{\text{cloud}} &= \lambda_{\text{cloud}}(1) + \lambda_{\text{cloud}}(2) \\
\lambda_{\text{cloudlet}} &= \lambda_{\text{cloudlet}}(1) + \lambda_{\text{cloudlet}}(2)
\end{aligned} \tag{15}$$

5.1.4 Average population

Is $E[N_i](c)$ the **average number of class c jobs into a component i.**

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:

$$\begin{aligned}
E[N_{\text{cloudlet}}](1) &= \sum_{(n_1,n_2)\in M} n_1 \cdot \pi(n_1,n_2) \\
E[N_{\text{cloudlet}}](2) &= \sum_{(n_1,n_2)\in M} n_2 \cdot \pi(n_1,n_2) \\
E[N_{\text{cloudlet}}] &= E[N_{\text{cloudlet}}](1) + E[N_{\text{cloudlet}}](2) \\
&= \sum_{(n_1,n_2)\in M} (n_1+n_2) \cdot \pi(n_1,n_2)
\end{aligned} \tag{16}$$

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**[13] in order to get cloud's average population as shown in Equation 17.

$$\begin{aligned}
E[N_{\text{cloud}}](1) &= \lambda_{\text{cloud}}(1) \cdot E[T_{\text{cloud}}](1) \\
&= \lambda_{\text{cloud}}(1) \cdot (E[T_{Q_{\text{cloud}}}](1) + E[S_{\text{cloud}}](1)) \\
&= \lambda_{\text{cloud}}(1) \cdot (\tfrac{1}{\mu_{\text{cloud}}(1)}) \\
&= \frac{\lambda_{\text{cloud}}(1)}{\mu_{\text{cloud}}(1)}
\end{aligned} \tag{17}$$

[12] *Cfr. Ivi*, pag. 315, equation (18.1)
[13] *Cfr. Ivi*, pag. 95, theorem (6.1)

Similarly:

$$E[N_{\text{cloud}}](2) = \frac{\lambda_{\text{cloud}}(2)}{\mu_{\text{cloud}}(2)} \tag{18}$$

$$E[N_{\text{cloud}}] = E[N_{\text{cloud}}](1) + E[N_{\text{cloud}}](2) \tag{19}$$

Finally we can get global average job populations as follows:

$$\begin{aligned} E[N](1) &= E[N_{\text{cloudlet}}](1) + E[N_{\text{cloud}}](1) \\ E[N](2) &= E[N_{\text{cloudlet}}](2) + E[N_{\text{cloud}}](2) \\ E[N] &= E[N](1) + E[N](2) \end{aligned} \tag{20}$$

5.1.5 Average response time

Is $E[T_i](c)$ the **mean response time experienced by a class c jobs into a component i.**

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 $E[T_i](c)$ using Little's Law.
- Since our system haven't queues, because of there is no waiting time experienced by jobs, is true that $E[T_i](c)$ is also equal to $E[S_i](c)$, that is **mean service time experienced by a class c jobs into a component i**, which is equal to $1/\mu_i(c)$, where $\mu_i(c)$ **means average service rate at which a class c jobs into a component i is served**.

Therefore we can get these metric as follows.

$$\begin{aligned} E[T_{\text{cloudlet}}](1) &= \frac{E[N_{\text{cloudlet}}](1)}{\lambda_{\text{cloudlet}}(1)} \\ &= E[S_{\text{cloudlet}}](1) \\ &= \frac{1}{\mu_{\text{cloudlet}}(1)} \end{aligned} \tag{21}$$

$$\begin{aligned} E[T_{\text{cloudlet}}](2) &= \frac{E[N_{\text{cloudlet}}](2)}{\lambda_{\text{cloudlet}}(1)} \\ &= E[S_{\text{cloudlet}}](2) \\ &= \frac{1}{\mu_{\text{cloudlet}}(2)} \end{aligned} \tag{22}$$

$$\begin{aligned} E[T_{\text{cloud}}](1) &= \frac{E[N_{\text{cloud}}](1)}{\lambda_{\text{cloud}}(1)} \\ &= E[S_{\text{cloud}}](1) \\ &= \frac{1}{\mu_{\text{cloud}}(1)} \end{aligned} \tag{23}$$

$$\begin{aligned} E[T_{\text{cloud}}](2) &= \frac{E[N_{\text{cloud}}](2)}{\lambda_{\text{cloud}}(2)} \\ &= E[S_{\text{cloud}}](2) \\ &= \frac{1}{\mu_{\text{cloud}}(2)} \end{aligned} \tag{24}$$

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

$$\begin{aligned} E[T](1) &= E[T_{\text{cloudlet}}](1) \cdot P\{\text{An arrival class 1 job is sent to cloudlet}\} \\ &\quad + E[T_{\text{cloud}}](1) \cdot P\{\text{An arrival class 1 job is sent to cloud}\} \\ &= E[T_{\text{cloudlet}}](1) \cdot \Pi_{\text{SendToCloudlet}} + E[T_{\text{cloud}}](1) \cdot \Pi_{\text{SendToCloud}} \end{aligned} \tag{25}$$

Similarly:

$$\begin{aligned} E[T](2) &= E[T_{\text{cloudlet}}](2) \cdot P\{\text{An arrival class 2 job is sent to cloudlet}\} \\ &\quad + E[T_{\text{cloud}}](2) \cdot P\{\text{An arrival class 2 job is sent to cloud}\} \\ &= E[T_{\text{cloudlet}}](2) \cdot \Pi_{\text{SendToCloudlet}} + E[T_{\text{cloud}}](2) \cdot \Pi_{\text{SendToCloud}} \end{aligned} \tag{26}$$

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

$$E[T] = E[T](1) \cdot \frac{\lambda_1}{\lambda_1 + \lambda_2} + E[T](2) \cdot \frac{\lambda_2}{\lambda_1 + \lambda_2} \tag{27}$$

5.1.6 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:

$$X = \lambda = \lambda_1 + \lambda_2 \tag{28}$$

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

$$X_{\text{cloud}} = \lambda_{\text{cloud}} \tag{29}$$

$$X_{\text{cloudlet}} = X - X_{\text{cloud}} \tag{30}$$

5.1.7 Summary of analytical results

	Class 1 Jobs			Class 2 Jobs			Cloudlet	Cloud	Glo
	Cloudlet	Cloud	System	Cloudlet	Cloud	System			
oudlet									0.5860746
loud									0.4139253
s/s)	2.3444	1.655701571128044	4	3.663125	2.586875	6.25	6.007525	4.242475	10.
bs)	5.2040	6.622806284512176	11.8264	13.5521	11.75852273	25.31062273	18.7561	18.38092273	37.137
] (s)	20/9	4	2.835407407	100/27	50/11	4.225763636	3460/1107	1954/451	3.6831
	0.2604888889			0.6783564815			0.9388453704		

5.2 System based on access control Algorithm 2

$$\begin{cases}
(\lambda_1+\lambda_2)\pi_{(0,0)} = \mu_1\pi_{(1,0)} + \mu_2\pi_{(0,1)} & \\
(\lambda_1+\lambda_2+i\mu_1)\pi_{(i,0)} = \lambda_1\pi_{(i-1,0)} + \mu_1(i+1)\pi_{(i+1,0)} + \mu_2\pi_{(i,1)} & \forall i\in\mathbb{N} \mid 1\le i\le S-1 \\
(\lambda_1+\lambda_2+i\mu_2)\pi_{(0,i)} = \lambda_2\pi_{(0,i-1)} + \mu_1\pi_{(1,i)} + \mu_2(i+1)\pi_{(0,i+1)} & \forall i\in\mathbb{N} \mid 1\le i\le S-1 \\
(S\mu_1+\lambda_1)\pi_{(S,0)} = \lambda_1\pi_{(S-1,0)} + \lambda_1\pi_{(S-1,1)} + (S+1)\mu_1\pi_{(S+1,0)} & \\
(\lambda_1+S\mu_2)\pi_{(0,S)} = \lambda_2\pi_{(0,S-1)} & \\
(i\mu_1+j\mu_2+\lambda_1)\pi_{(i,j)} = \lambda_1\pi_{(i-1,j)} + \lambda_2\pi_{(i,j-1)} & \forall i,j\in\mathbb{N} \mid \begin{matrix}1\le i\le S-1\\1\le j\le S-1\end{matrix} \mid i+j=S \\
(\lambda_1+\lambda_2+i\mu_1+j\mu_2)\pi_{(i,j)} = \lambda_1\pi_{(i-1,j)} + \lambda_2\pi_{(i,j-1)} + \mu_1(i+1)\pi_{(i+1,j)} + \mu_2(j+1)\pi_{(i,j+1)} & \forall i,j\in\mathbb{N} \mid \begin{matrix}1\le i\le S-1\\1\le j\le S-1\end{matrix} \mid i+j<S \\
(i\mu_1+\lambda_1)\pi_{(i,0)} = \mu_1(i+1)\pi_{(i+1,0)} + \lambda_1\pi_{(i-1,0)} & \forall i\in\mathbb{N} \mid S+1\le i\le N-1 \\
N\mu_1\pi_{(N,0)} = \lambda_1\pi_{(N-1,0)} & \\
\sum\pi_{(i,j)} = 1 & \forall i,j\in\mathbb{N}_0
\end{cases}$$

5.2.1 Probabilities computing

although... To properly analyse this system, we need do compute some useful probabilities.

Similarly to the previous case we need to now following probabilities:

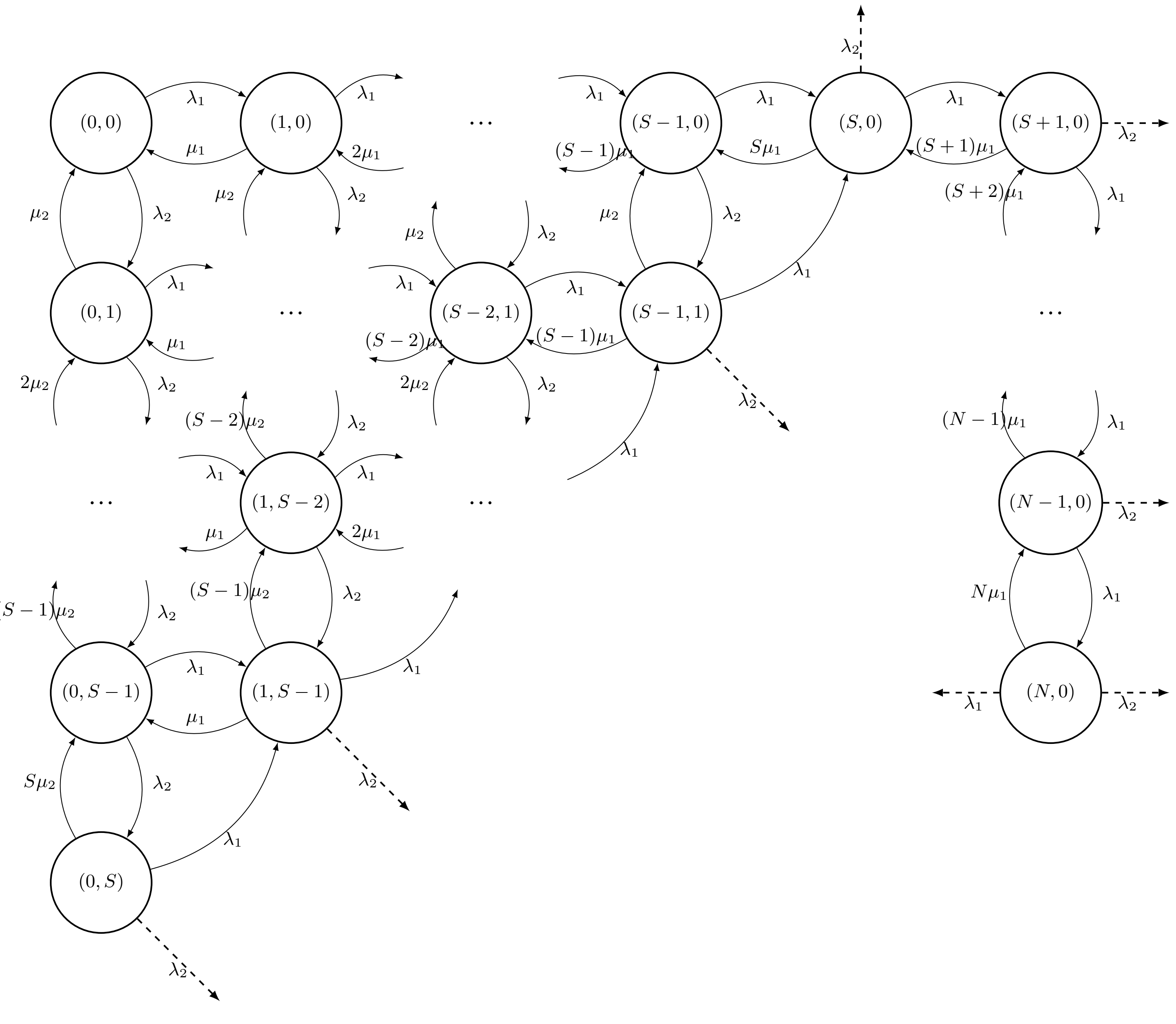
Table 3: Lista dei file del malware FASTCash

$\Pi_{\text{SendToCloudlet}}(k)$	Probability that an arriving job of class k on controller has to be forwarded to cloudlet
$\Pi_{\text{SendToCloud}}(k)$	Probability that an arriving job of class k on controller has to be forwarded to cloud
$\Pi_{\text{Class2JobInterruption}}(k)$	Probability that a job of class 2 running on cloudlet has to be interrupted and forwarded to cloud due of arriving class 1 job on cloudlet.

Differently from the previous case, owing to the use of a different access control algorithm, jobs of different classes have different probability according to which an arriving job on controller has to be send to cloud. From CTMC analysis, we can easily understand that:

$$\begin{aligned}
\Pi_{\text{SendToCloud}}(1) &= P\{\text{An arrival class 1 job on controller sees } N \text{ class 1 jobs in cloudlet}\} \\
&= P\{n_1 = N\} \\
&= \pi(N,0)
\end{aligned} \tag{31}$$

Figure 3: Cloudlet system component modeled using a CTMC



$$\begin{aligned}
\Pi_{\text{SendToCloud}}(2) &= P\{\text{An arrival class 2 job on controller sees that number of jobs in cloudlet exceed or} \\
&= P\{n_1 + n_2 \geq N\} \\
&= \sum_{\substack{n_1, n_2 = 0 \\ n_1 + n_2 = S}}^{S} \pi(n_1, n_2) + \sum_{n_1 = S+1}^{N} \pi(n_1, 0) \\
&= \sum_{\substack{0 \leq n_1 \leq N \\ 0 \leq n_2 \leq S \\ n_1 + n_2 \geq S}} \pi(n_1, n_2)
\end{aligned} \tag{32}$$