Discrete-event simulation for efficient and stable resource allocation in collaborative mobile cloudlets

Steven Bohez[a,*], Tim Verbelen[a], Pieter Simoens[a,b], Bart Dhoedt[a]

[a]*Department of Information Technology*
Internet Based Communication Networks and Services (IBCN)
Ghent University - iMinds
Gaston Crommenlaan 8/201
B-9050 Ghent, Belgium
[b]*Department of Industrial Technology and Construction*
Ghent University College
Valentin Vaerwyckweg 1
B-9000 Ghent, Belgium

Abstract

The deployment of highly interactive, media-rich applications on mobile devices is hindered by the inherent limitations on compute power, memory and battery capacity of these hand-held platforms. The cloudlet concept, opportunistically offloading computation to nearby devices, has proven to be a viable solution in offering resource-intensive applications on mobile devices. In this paper, we propose to extend the cloudlet concept with collaborative scenarios, in which not only hardware resources for processing are shared between all cloudlet users, but also the data computed.

In a cloudlet, the resource demand should be spread over all available cloudlet nodes. User mobility and fluctuations in wireless bandwidth will cause the optimal resource allocation to vary over time. The cloudlet middleware must continuously balance the performance gain of reallocating components with the operational costs in terms of user experience and management complexity. In this paper, we formulate this optimization problem based on a theoretical cloudlet model capturing the infrastructure, application structure and user behaviour.

In order to solve this problem, two heuristic allocation algorithms based on Steepest Descent (SD) and Simulated Annealing (SA) are described. Besides optimality of the found solution, it is also important to limit the number of reallocations at runtime. To evaluate the performance and stability of the algorithms, we propose a discrete-event model for cloudlet simulation. For multiple application scenarios, we observe that SD performs 4 times less reallocations than SA. By introducing hysteresis, the number of reallocations by SA can be nearly halved without any significant degradation of application performance.

Keywords: mobile cloud computing, cloudlet, collaborative applications, discrete-event simulation

1. Introduction

Mobile devices such as smartphones and tablets have become more popular than ever. Gartner [1] has estimated the total number of smartphones exceeding 1.8 billion items sold in 2013. Recently, even smaller mobile devices and wearable computers have become available on the consumer market. Manufacturers such as Samsung [2] and Sony [3] have introduced smart watches acting as easily-accessible dashboards for their smartphones. Meanwhile, Google [4] has introduced a pair of nearly-autonomous smart glasses with video-capture and voice-recognition functionality.

*Corresponding author. Tel.: +32 93314940
Email addresses: steven.bohez@intec.ugent.be (Steven Bohez), tim.verbelen@intec.ugent.be (Tim Verbelen), pieter.simoens@intec.ugent.be (Pieter Simoens), bart.dhoedt@intec.ugent.be (Bart Dhoedt)

The popularity of these mobile devices has multiple reasons. Not only are they portable and always-connected, but they have access to hundreds of thousands of easy-to-install apps. Due to the high number of sensors (such as cameras and microphones, inertial sensors, location sensors and so on) and high-resolution touchscreens, mobile devices are prime candidates for highly-interactive and media-rich experiences, such as immersive games and Augmented Reality (AR). Even though continuous advances in mobile processors and battery technology have made mobile devices more powerful, they still struggle to execute this kind of applications due to their inherent limitations on resources such as processing power, memory and battery capacity. These limitations are even more pressing on the newest generation of wearable computers.

To offset resource limitations, offloading computation and/or storage to infrastructure in the network has become a necessity [5]. Due to the large and variable network delay, offloading to a distant cloud is however infeasible for applications involving highly responsive user interaction. To offload such applications, resources need to be available closer to the user, e.g. co-located with the wireless access point or at the base station [6]. This is the concept of cyber foraging [7]. As the infrastructure is placed in the vicinity of the mobile users, the network delay is limited to the delay incurred in the single-hop wireless access network.

Cyber foraging can be realized using VM-based cloudlets [8], which are personalized Virtual Machines (VMs) on a nearby trusted server that execute (parts of) the mobile application. The concept of VM-based cloudlets has recently evolved to component-based cloudlets [9, 10]. These are systems consisting of a group of computing nodes, both fixed and mobile, that are sharing resources with one another. Mobile applications consist of several loosely-coupled components that can quickly be redeployed at runtime in order to offload parts of the application to other devices. Using smaller software components instead of VMs as a unit of deployment, allows for both faster and more flexible offloading.

Another type of applications that are gaining popularity, are collaborative applications. Also called groupware, these are applications where multiple users work together in a shared context to accomplish a common goal, possibly in an interactive and real-time fashion. Niantic Labs' Ingress [11] is an example of an interactive and collaborative AR game with over half a million active users. These applications face additional challenges when executed in a mobile environment. Not only does the group of users change over time, with new users arriving and others leaving, the wireless connectivity may cause users to become temporarily disconnected.

The resource-sharing concept of cloudlets offers an interesting opportunity for collaborative scenarios: besides sharing computing resources, users may also share data such as processing results or context information. They could even share a software component that holds the same user data to reduce resource consumption. For example, in a location-based collaborative game, users in close proximity of each other could share the same virtual space and interact with the same virtual objects.

To realize interactive collaboration in a way that is transparent to the user and easy to implement by the application developer, we propose to use a middleware platform that offers support for collaboration through state sharing. In [12], we extended the architecture of the component-based cloudlet middleware from [13] with support for collaborative scenarios to create a collaborative cloudlet middleware. The basic mechanisms facilitating collaboration are shared offloading, whereby a component instance is shared between multiple users, and state synchronization, whereby state is shared through the active exchange of state updates. These mechanisms are described in more detail in Section 3 of this paper.

An important aspect of any cloudlet middleware is the autonomous deployment and configuration of all application components currently executed in the cloudlet. The cloudlet middleware should optimize the user experience while coping with the processing and network limitations of the cloudlet. Collaborative scenarios bring additional configuration complexity since some components are used by multiple users. This optimization problem is formally described in Section 4 based on a theoretical model of both the cloudlet infrastructure and the behaviour and resource usage of the application components. In Section 5, two heuristic allocation algorithms, based on the search heuristics Simulated Annealing (SA) and Steepest Descent (SD), are discussed that optimize the allocation of components over the different nodes in the cloudlet while taking into account all necessary state sharing.

Evaluating the actual performance of such allocation algorithms is a difficult task. For a suitable allocation, it is not only important that (i) the resource consumption is optimized, (ii) the device and infrastructure constraints are satisfied, but also that (iii) the application deployment remains stable over time, i.e. unnecessary reallocations are avoided. In [12], the SA heuristic is evaluated in the static case by comparing the heuristic outcome for a number of randomly generated problems to the optimal solution obtained by exhaustive search. While this may give an indication

of the performance of the applied heuristics, the evaluation in this previous paper did not take into account runtime parameters. In practice, allocation algorithms are used in control loops, where the differences in input parameter values are typically minor between different runs. This is an important consideration when evaluating time-related performance metrics of the algorithm such as stability. Preferably, the full impact of cloudlet system dynamics on the allocation result should be evaluated, such as arriving and departing users, variations in computational and network demand, etc.

Setting up real-life experiments is however costly and time-consuming, especially for cloudlets of realistic proportions. Not only does the infrastructure need to be correctly configured, it is nearly impossible to exactly reproduce experimental conditions with different algorithms or algorithm configurations. This is however required for a fair comparison. Therefore, in Section 6, a discrete-event simulation model is proposed that captures the dynamic behaviour of the cloudlet on a method-level granularity. Using simulations, different allocation algorithms can be compared under the same circumstances and a wider range of experimental settings can be explored. In Section 7, different configurations of SA and SD are compared for different scenarios. Finally, Section 8 presents our main conclusions and ideas for future work.

2. Related work

2.1. Cyber foraging

Early cyber foraging systems, such as Spectra [14] and Chroma [15], required the application developer to preinstall routines on devices offering offloading. As this approach quickly becomes infeasible for multiple applications and devices, more recent systems use runtime code migration. These can roughly be split up into two categories: VM-based and component-based systems.

VM-based systems "copy" entire applications from mobile devices to nearby infrastructure on a virtual machine level. Either the mobile application is executed entirely in the dedicated VM, such as in the VM-based cloudlets in [8] and [16] or the decision is made on a per-routine basis, as in CloneCloud [17] by using profiling and code analysis. COMET [18] uses a Distributed Shared Memory (DSM) approach which allows for easy migration of individual application threads between VMs.

Instead of using VMs as offloadable units, component-based systems operate at the granularity level of software components, providing more flexibility for reallocation of applications. These systems can be differentiated by the type of the components they use. In some systems, these are the routines themselves, such as in MAUI [19] and Scavenger [20]. The components can be larger, such as Open Services Gateway initiative (OSGi) components in AlfredO [21] and AIOLOS [9]. Zhang et al. propose Weblets [22], which are RESTful web services, as units suitable for offloading. In [13], applications consist of several OSGi components running in an Execution Environment (EE). These components depend on and refer to one another through their required and offered interfaces. By changing component references from local to remote instances, redeployment operations by the cloudlet middleware happen transparently to both user and application developer. An in-depth comparison of cyber foraging and other mobile cloud computing systems is presented in [23].

2.2. Runtime optimization

In component-based cloudlets, components can be redeployed on other nodes at runtime according to a given optimization criterion, such as the execution time [9, 20], energy consumption [19], and/or throughput [24]. The algorithms used to allocate software components in component-based cyber foraging systems are often based on well-known graph-partitioning algorithms [25], although other techniques for runtime optimization have been successfully used, such as Naive Bayesian Learning in [22]. The management of collaborative cloudlets poses additional challenges, however, as we have to account for the collaboration requirements. Specifically, reallocating one component might affect multiple users.

In this paper, the use of search heuristics for runtime optimization of collaborative cloudlet middleware is evaluated. Simulated Annealing (SA), also known as annealing-based greedy, is a well-studied stochastic search heuristic which has been used successfully for runtime optimization of various cloud systems. In [26], the authors use SA for scheduling computing tasks on a multi-cloud system generated by Internet of Things (IoT) applications. As these tasks consist of few parallel jobs, but have high arrival rates and run-time variation, they require more specialised scheduling

algorithms. Results show that SA outperforms Shortest Queue First (SQF) techniques in terms of both performance and cost. Zhang et al. [27] use SA with similar success for bandwidth-aware scheduling of data-intensive tasks, where data transfer times and network congestion are non-negligible factors in minimizing completion time. The MAPCloud system, presented in [28], employs a variation on SA called CRAM for decomposing the workflow of mobile applications to execute on the mobile client and hybrid 2-tier cloud and incorporates Quality of Service (QoS). Their work is similar to ours in the sense that they use nearby infrastructure to offload tasks and use a SA heuristic to perform the decision. Our work, however, considers component-based instead of task-based applications and incorporates collaborative aspects.

2.3. Middleware for collaboration

In order for users to collaborate, information needs to be exchanged between them. With users joining and (possibly temporarily) leaving the group, support from the application middleware is needed to cope with these issues in transparent way, both for the user and developer. MoCa [29] is a middleware system specifically focused on mobile and collaborative applications. Mobile applications are statically divided in an application client and a server, with the server residing on infrastructure in the network alongside the MoCa-services and client proxies.

In [30], it is argued that due to the nature of mobile collaborative applications, a peer-to-peer architecture is more suited. This approach does not require a reliable centralized infrastructure, but also improves flexibility and scalability. A recent effort to create a large-scale collaborative middleware has been ContextNet [31, 32]. ContextNet aims to offer context-aware services and builds upon the Scalable Data Distribution Layer (SDDL) communication layer which provides robust, scalable and real-time data distribution between thousands of mobile clients. ContextNet however still requires a SDDL core network with management nodes. LaCOLLA [33] takes the concept of collaboration even further and introduces the idea of collectivism, where participants explicitly provide resources for the group's benefit. Moreover, these resources not only comprise data(storage) but also processing capabilities. While the focus lies more on availability of services and self-sufficiency of the user group, this is fairly similar to what happens in a collaborative cloudlet without runtime optimization.

Our collaborative cloudlet middleware adopts a peer-to-peer-like architecture, in the sense that any server-like entities required for cloudlet management and state sharing are autonomously assigned to the devices of the cloudlet. If high-capacity infrastructure is available, it will automatically be chosen to handle the more challenging processing, including any centralized processing tasks. However, if no such high-capacity infrastructure is available, these tasks are assigned to other nodes in the cloudlet.

2.4. Cloud simulators

In order to efficiently compare resource provisioning policies, simulation is often used in the domain of cloud computing as the cost of setting up testbeds that reflect operational cloud systems is prohibiting. One of the most established cloud simulators is CloudSim [34], which itself is built on GridSim [35], a discrete-event simulator for grid computing environments that models cluster behaviour, including traffic profiles and resource usage. CloudSim extends GridSim with models for cloud-specific aspects, such as virtualization, resource provisioning, application services and so on. The main benefit of CloudSim is that it is able to simulate clouds of very large scale, containing tens of thousands of VMs. CloudSim has been modified and extended by various authors to focus on specific aspects of cloud computing.

Another simulator, GreenCloud [36], uses a more sophisticated and accurate network model by building on top of a network simulator. The main focus of GreenCloud is the measurement of energy consumption. MDCSim [37] on the other hand, focusses on detailed simulation of multi-tier data centres. Finally, iCanCloud [38] aims to predict performance and cost of using specific hardware for a particular set of applications. In [39], a more detailed overview is given of available cloud simulators. However, these simulators are aimed at large-scale, centralized cloud systems and are not able to model the mobile aspects of a component-based cloudlet. Moreover, the simulated clouds are used for back-end processing of discrete tasks, not for the long-lived user sessions with many time-critical calls that occur in a cloudlet. Finally, they provide no means for modelling collaboration as described in the following section. We present in this paper a new discrete-event model on top of which a custom cloudlet simulator is implemented.

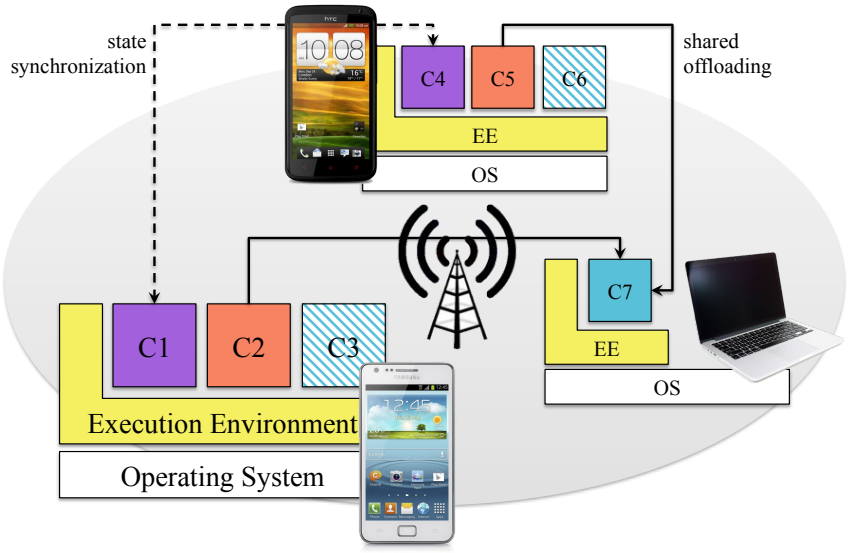


Figure 1: Collaboration in a cloudlet. Solid lines represent collaboration through a shared component instance, dashed lines represent collaboration through explicit synchronization messages. The shaded components are disabled.

3. Support for collaboration in cloudlet middleware

Implementing collaborative applications can be a tedious task for the application developer, as this involves distribution and synchronization of state between all application instances, and correctly handling arriving and leaving of users. To alleviate this task, we have built a framework on top of the component-based cloudlet middleware proposed in [13], by adding mechanisms to transparently share components and their state between multiple users. State sharing and user arrivals are handled by the framework, which minimizes the burden put on the application developer. In this section, we give an overview of the framework and the different collaboration mechanisms.

Fig. 1 shows the architecture of the cloudlet middleware. Applications consist of several components that are managed by an Execution Environment (EE). A single device in the cloudlet may host multiple EEs. The EE is responsible for starting and stopping components and resolving dependencies by matching the required and offered interfaces of each component.

In the cloudlet middleware, migrating an individual component from a user's mobile device to another node, further called solitary offloading, is achieved by instantiating the component on the target EE. All method calls from depending components are then redirected to the offloaded instance. In a collaborative scenario, this can be easily extended to shared offloading, where multiple mobile devices redirect their method calls to the same remote instance, and thus implicitly sharing the state of this single remote component. The general idea is shown in Fig. 1. Component C7 is shared between users: the depending components C2 and C5 refer to the same instance. Shared offloading is a passive mechanism, as it requires no further action after the initial set-up.

To access this shared instance implies that on all nodes, except the node hosting the shared instance, remote method calls will be needed. This may introduce additional network delays. Also, for some components, offloading is simply not feasible due to hardware dependencies. To address these issues, state synchronization is proposed as a second collaboration mechanism. State synchronization uses an active exchange of messages between multiple component instances of the same type. Components can thus remain allocated on user devices, while users are still able to collaborate by state sharing, as shown in Fig. 1. The EE will identify other instances of the same component deployed on other connected EEs, and will actively distribute state updates of this component to the other EEs. This is done using a client-server model, where one instance becomes responsible for correctly distributing the state updates and resolving any conflicts.

To support collaboration, the application developer only has to define the state of a component and how state updates of possibly multiple sources have to be merged in a consistent way. This is done by implementing a set of

method calls for each component that needs to share state. Specifically, the `mergeState`-method will be invoked on the synchronization server component to process a state update, after which the `setState`-method is called on all client components to distribute the processed update. The collaboration mechanisms and programming model are described in more detail in [12].

4. Theoretical model for runtime optimization

The collaborative cloudlet middleware previously introduced offers various possibilities for configuring collaboration and component deployment. In the case of shared offloading, a decision needs to be made on the allocation of the shared instance and in the case of state synchronization, on the server instance that will merge and distribute state updates. The configuration of the cloudlet should be adaptable at runtime in order to cope with the dynamic behaviour of the cloudet, such as users joining and leaving, and variations in processing and network load. This makes manual configuration infeasible and requires an autonomous control loop.

A well-known approach is to use a Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) control loop as proposed by [40]. In this approach, an element to be autonomously managed (in our case, the cloudlet) is controlled by a 4-step loop. In the *monitor* step, monitoring information (device load, application behaviour, etc.) is collected for a certain time interval. This information is aggregated and stored in the *knowledge* component. Based on the knowledge and a model, the *analyze* step will perform possibly complex analysis of and reasoning on the state of the system (e.g. check if all service requirements are met). The *plan* function will then determine which actions need to be taken in order to meet certain goals. In our case, this plan step implies executing the allocation algorithm in order to optimize the cloudlet. Finally, the *execute* step will actually perform the actions suggested by the previous step.

In our middleware, this control loop is performed by a cloudlet manager node, which is autonomously chosen based on available node resources. The allocation algorithm can be seen as solving an optimization problem, based on a mathematical model of the cloudlet infrastructure and application behaviour.

4.1. Cloudlet model

We extend the cloudlet model presented in [41] by incorporating aspects of collaboration. There are three main aspects of the cloudlet model: the infrastructure, the components and their allocation and the observed behaviour.

4.1.1. Infrastructure

The infrastructure of the cloudlet consists of the devices (or computing nodes) and the network. The computing nodes are grouped in a set D, with every node $d \in D$ having two properties: the processing speed $speed_d$ of a single core (e.g. instructions per second) and the number of cores $\#cores_d$. The network is modelled as a single shared (wireless) medium that can be represented by its *bandwidth*.

4.1.2. Application structure

The application structure is modelled as a group of components c. All the components in the cloudlet form the component set C. To track the current allocation of individual components, we define the allocation matrix X_{cd}.

$$X_{cd} = \begin{cases} 1 & \text{if } c \text{ is allocated on } d \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Using X_{cd}, we define the variable $H_{c_i c_j}$ as being 1 if and only if components c_i and c_j are allocated on different nodes, so any communication between these components needs to go over the network.

$$H_{c_i c_j} = 1 - \sum_{d \in D} X_{c_i d} \cdot X_{c_j d} \tag{2}$$

Each component c also has a certain type, $\tau(c)$, so an application consists of a group of components with distinct types. To run the application, a component instance of each type needs to be deployed. Different instances of the

same type may be running in the cloudlet, originating from application instances belonging to different users. For collaboration, this implies that components requiring state sharing only do so with components of the same type.

In order to incorporate shared offloading into the model, we use an additional variable $Y_{c_i c_j}$, which is 1 if c_i is currently being substituted by c_j. After performing shared offloading on a set components C' to a shared instance c_{shared}, it should hold that $Y_{c_i c_j} = 1 \Leftrightarrow c_j = c_{shared}, \forall c_i \in C'$.

$$Y_{c_i c_j} = \begin{cases} 1 & \text{if } c_i \text{ is substituted by } c_j \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

4.1.3. Application behaviour

The runtime behaviour of an application is modelled using sequences of method calls between component instances. Each sequence follows a specific path in the control flow graph of the application, and is assumed to be executed in a single thread. Note that an application is therefore defined by its component types and the sequences of method calls that occur between these components.

A sequence $s \in S$ occurs with a certain frequency $freq_s$ and consists of a number of successive method calls $m^s_{c_i c_j}$ originating from c_i and executed by c_j. Every method call in s generates a load $load_{m^s_{c_i c_j}}$ (e.g. a number of instructions) that needs to be processed, an argument and result of size, $arg_{m^s_{c_i c_j}}$ and $res_{m^s_{c_i c_j}}$ respectively, and occurs $\#calls_{m^s_{c_i c_j}}$ times in the sequence (e.g. in a loop). Each method also has a name $\nu\left(m^s_{c_i c_j}\right)$. Note that $\#calls_{m^s_{c_i c_j}}$ is an inherent property of the sequence, a different number of calls of $m^s_{c_i c_j}$ yields a different sequence.

While state synchronization is technically not part of any application behaviour, it can be modelled in a similar way. When using the client-server approach described in Section 3, each state update of component c_i will first generate a method call $m^{merge}_{c_i c_{server}}$, with $\tau(c_i) = \tau(c_{server})$ and $\nu\left(m^{merge}_{c_i c_{server}}\right) =$ `mergeState`. Afterwards, a set of $m^{set}_{c_{server} c_j}$ calls will be generated, for $\forall c_j \in C : \tau\left(c_j\right) = \tau(c_{server})$ with $\nu\left(m^{set}_{c_{server} c_j}\right) =$ `setState`.

4.2. Objective

Using the model described above, a performance metric of the collaborative cloudlet to be optimized can be defined. The objective to be minimized is the average relative usage $usage_{avg}$ of all the nodes in the cloudlet. By minimizing this objective function, the load on the mobile devices can be reduced while not overloading the high-capacity nodes. For the same usage, a high-capacity, fixed node is generally able to process more load than a mobile node as it has a higher speed and/or has more cores. The average usage is calculated by averaging the $usage_d$ of all nodes in the cloudlet.

$$usage_{avg} = \frac{\sum_{d \in D} usage_d}{\#D} \tag{4}$$

The relative usage of a single node $usage_d$ can be found by dividing the imposed load (per unit of time) by the maximum load the node can process. The maximum load a node can process is the speed of a single core multiplied by the number of cores.

$$usage_d = \frac{load_d}{speed_d \cdot \#cores_d} \tag{5}$$

The load on a single node is simply the sum of the load imposed by each observed sequence on the node.

$$load_d = \sum_{s \in S} load_{sd} \tag{6}$$

The load per unit of time on a specific node d incurred by a specific sequence s can be calculated as the sum of the loads per unit of time of the method calls $m^s_{c_i c_j}$ in the sequence. A single call of $m^s_{c_i c_j}$ generates a load $load_{m^s_{c_i c_j}}$.

By multiplying this by the number of calls $\#calls_{m^s_{c_ic_j}}$ in the sequence and the frequency $freq_s$ at which the sequence occurs, we get the total load per unit of time of $m^s_{c_ic_j}$. However, we are only interested in the methods $m^s_{c_ic_j}$ of s that are actually executed on node d. A method call is executed on d if the target component c_j is allocated on d, or if $X_{c_jd} = 1$. However, we also need to account for any substitutions of c_j by c_k using $Y_{c_jc_k}$. Only if the actual target component c_k is allocated on d ($X_{c_kd} = 1$), may the load generated by $m^s_{c_ic_j}$ be added to $load_{sd}$.

$$load_{sd} = \sum_{c_i \in C} \sum_{c_j \in C} \sum_{c_k \in C} X_{c_kd} \cdot Y_{c_jc_k} \cdot load_{m^s_{c_ic_j}} \cdot \#calls_{m^s_{c_ic_j}} \cdot freq_s \tag{7}$$

4.3. Constraints

Besides the objective to be minimized, there are certain constraints that need to be satisfied when solving the optimization problem. A first constraint is that all software components need to be deployed. This constraint is however implicitly satisfied by the allocation algorithm if only valid reallocations are allowed (see Section 5). Other constraints reflect the limited capacity of the infrastructure in the cloudlet. A first constraint is that the total load imposed on any node cannot exceed its total capacity.

$$load_d \leq speed_d \cdot \#cores_d, \forall d \in D \tag{8}$$

Because each sequence is assumed to be processed in a single thread (as explained above), the load of any single sequence may not exceed the speed of any node in the cloudlet.

$$load_{sd} \leq speed_d, \forall s \in S, \forall d \in D \tag{9}$$

Finally, the network must be able to cope with the bandwidth needed for remote method calls, i.e. it must fit within the available bandwidth.

$$traffic \leq bandwidth \tag{10}$$

The total bandwidth needed can be found by summing the generated traffic of every observed sequence.

$$traffic = \sum_{s \in S} traffic_s \tag{11}$$

The amount of traffic generated by a given sequence $traffic_s$ can be calculated based on the argument and result sizes of the remote method calls in the sequence. Only when the source and target components are on different nodes, is the call $m^s_{c_ic_j}$ a remote call and hence generates traffic. These are calls $m^s_{c_ic_j}$ for which $H_{c_ic_j} = 1$. However, we must again take any substitutions $Y_{c_ic_k}$ and/or $Y_{c_jc_l}$ into account. If a call $m^s_{c_ic_j}$ is remote, the amount of traffic it generates equals the sum of the argument and result sizes, multiplied by the number of calls it occurs in the sequence and the frequency of the sequence itself.

$$traffic_s = \sum_{c_i \in C} \sum_{c_j \in C} \sum_{c_k \in C} \sum_{c_l \in C} H_{c_kc_l} \cdot Y_{c_ic_k} \cdot Y_{c_jc_l} \cdot \left(arg_{m^s_{c_ic_j}} + res_{m^s_{c_ic_j}} \right) \cdot \#calls_{m^s_{c_ic_j}} \cdot freq_s \tag{12}$$

To incorporate these constraints into the optimization problem, they are added as additional terms to the objective function. This gives us the following, complete objective function.

$$
\begin{aligned}
f_{obj} = & \; usage_{avg} \\
& + \alpha \cdot V\left(traffic, bandwidth\right) \\
& + \beta \cdot \sum_{d \in D} V\left(load_d, speed_d \cdot \#cores_d\right) \\
& + \gamma \cdot \sum_{d \in D} \sum_{s \in S} V\left(load_{sd}, speed_d\right)
\end{aligned} \tag{13}
$$

The function V is the cost function associated with the constraints and is defined as follows.

$$
V\left(a, b\right) = W\left(\frac{a-b}{b}\right) \tag{14}
$$

$$
W\left(c\right) = \begin{cases} 0 & c \leq 0 \\ c & 0 \leq c \end{cases} \tag{15}
$$

By defining W as the hinge cost function, V expresses the relative violation of value a with respect to its limit b. The coefficients α, β and γ express the penalty of violating the corresponding constraint (here $\alpha = \beta = \gamma = 10^2$).

4.4. Complexity

At first glance, the complexity of Equation 7 seems to be of order $O\left(\#C^3\right)$, where $\#C$ is the total number of components. However, this calculation can be simplified by using an indexing operation (complexity $O\left(1\right)$) to find the substitute c_k of the target component instead of a summation, and by only iterating over the known methods $m^s_{c_i c_j}$ occurring in sequence. If $\#s_{max}$ denotes the maximum number of different methods occurring in all observed sequences s, then the complexity of Equation 7 is $O\left(\#s_{max}\right)$. This implies that the complexity of Equations 6 and 5 is $O\left(\#S \cdot \#s_{max}\right)$, where $\#S$ is the number of observed sequences. Furthermore, the complexity of Equation 4 and the first term of the objective function becomes $O\left(\#D \cdot \#S \cdot \#s_{max}\right)$, where $\#D$ is the number of nodes.

In a similar way, the complexity of Equation 12 can be reduced to $O\left(\#s_{max}\right)$. Equation 11 and the second term of the objective function are therefore of complexity $O\left(\#S \cdot \#s_{max}\right)$. By storing and reusing the values of $load_{sd}$ and $load_d$ when calculating $usage_{avg}$, the third and fourth term of the objective function are of complexity $O\left(\#D\right)$ and $O\left(\#D \cdot \#S\right)$ respectively. Evaluating the full objective function is then of complexity $O\left(\#D \cdot \#S \cdot \#s_{max}\right)$.

5. Allocation algorithms for runtime optimization

In order to be used in an operational cloudlet, allocation algorithms must find a solution in limited time. This implies that the execution time of the algorithm is no longer than the desired control loop period discussed in Section 4.1. Consider the scenario of a cloudlet where every component is offloadable to every node, then the total number of valid reallocations scales as $O\left(\#D^{\#C}\right)$, where $\#D$ is the number of nodes in the cloudlet and $\#C$ is the total number of components. Looking at how the number of reallocations scales exponentially with the number of components, a brute-force, exhaustive approach is infeasible. Even more intelligent solvers such as Quadratic Programming (QP) do not scale well enough with the cloudlet size to be applied at runtime.

Heuristic allocation algorithms are hence needed for runtime optimization of the cloudlet. While heuristics are not guaranteed to find the optimal solution, the goal is to provide a sufficient approximation in limited execution time and to scale better with problem size. The output of an allocation algorithm is a set of actions that improve the current allocation of the cloudlet.

5.1. Actions

In a collaborative cloudlet which supports the mechanisms discussed in Section 3, valid actions belong to either one of the following types.

Solitary offloading This type of action will migrate a single component instance c from node d to d' in the cloudlet. This boils down to updating $X_{cd} \leftarrow 0$ and $X_{cd'} \leftarrow 1$. These actions are only applicable to components that are offloadable but do not require state sharing.

Shared offloading When multiple instances c_i of a component type exist, shared offloading will replace them with a single, shared instance c_{shared}. This results in updating $Y_{c_i c_i} \leftarrow 0$ and $Y_{c_i c_{shared}} \leftarrow 1$, for $\forall c_i \in C : \tau(c_i) = \tau(c_{shared})$. Shared offloading can also change the allocation of the shared instance to node d'. This requires updating $X_{c_{shared} d} \leftarrow 0$ and $X_{c_{shared} d'} \leftarrow 1$. Shared offloading is only possible for component types that are offloadable and need to share state.

Switch server This will change the synchronization server for a component type that is currently using state synchronization as collaboration mechanism. State synchronization is also used as the default collaboration mechanism, as it is applicable to any component type that needs to share state, in contrast to shared offloading. This switching requires updating the sequences representing the synchronization messages. For every `mergeState` call $m^{merge}_{c_i c_{server_{old}}}, \forall c_i \in C : \tau(c_i) = \tau(c_{server_{new}})$, the target node has to be changed from $c_{server_{old}}$ to $c_{server_{new}}$. Likewise, the source node for every `setState` has to be changed to the new server.

Sync2Shared This action switches from using state synchronization for collaboration, to sharing a component instance. The synchronization server is chosen as the shared instance. These actions are only possible for offloadable component types and require the sequences representing synchronization messages to be excluded when calculating the objective function.

Shared2Sync Actions of this type will switch from sharing an instance to state synchronization as the collaboration mechanism. The shared instance will become the new synchronization server. Synchronization messages are again included in calculating the objective function.

If these actions are performed on a valid configuration of the cloudlet, i.e. all components are deployed and all required state sharing is possible, the result will also be a valid configuration. If the input of the allocation algorithm is a valid configuration, then the result will also be valid, as long as the suggested actions belong to any of these types.

5.2. Search heuristics

In the following sections two well-known heuristics, Steepest Descent (SD) and Simulated Annealing (SA), are described and their application in autonomous management of collaborative cloudlets is evaluated. Both search heuristics explore the solution space by performing selected actions from the types discussed above.

5.2.1. Steepest Descent

A first heuristic algorithm to be described is SD, for which the procedure is shown in Algorithm 1. SD is a deterministic algorithm that will immediately converge to a minimum following the steepest decline available. After initialisation with the current deployment, the gain of all valid actions from the current deployment is determined and the action with the highest positive gain is applied. The gain of an action is defined as the difference in objective function that would occur by performing that action. This process repeats itself until no further actions with a positive gain are found or the number of iterations reaches a given threshold. SD is easily parallelizable, as the gain of every action can be calculated independently.

5.2.2. Simulated Annealing

The major disadvantage of SD is that only a limited part of the solution space is explored and it may hence converge to a local minimum instead of the global minimum. To cope with this issue, an algorithm that performs a random instead of deterministic search can be used.

SA is a more advanced random search heuristic that uses a control factor called the temperature. The procedure is shown in Algorithm 2. SA is initialised using the current deployment of the cloudlet, after which a starting temperature and an epoch length are determined. The algorithm then advances through a number of epochs between which the temperature is gradually decreased. During each epoch, a fixed number of valid actions are randomly selected. A selected action is accepted with probability $\exp(G/T)$, where G is the gain in the objective function by performing the

Algorithm 1 Steepest Descent

Input: *Initial*
Output: *Best*
  $Best \leftarrow Initial$
  repeat
    $K_{max} \leftarrow \emptyset$
    $G_{max} \leftarrow 0$
    for $\forall K \in allActions\,(Best)$ **do**
      $G \leftarrow gain\,(K, Best)$
      if $G > G_{max}$ **then**
        $G_{max} \leftarrow G$
        $K_{max} \leftarrow K$
      end if
    end for
    if $K_{max} \neq \emptyset$ **then**
      $Best \leftarrow performAction\,(K_{max}, Best)$
    end if
  until the stop criterion is met

action and T is the current temperature. Actions with a positive gain are always accepted, but actions with negative gain also have a chance of being accepted depending on the temperature. By initialising with a high temperature, a lot of actions with negative gain are likely to be accepted and a large part of the solution space can be explored. Decreasing the temperature is required for the algorithm to converge. In our implementation, the stop criterion depends on the fraction of actions that are accepted during an epoch.

A number of variations of the SA algorithm exist, among others on how the initial solution is chosen and on the choice of the cooling scheme. Johnson et al. [42] show that most of these variations have little to no impact on performance in the general case and the SA configuration boils down to a trade-off between running time and solution quality. We have opted for the well-performing baseline approach described in [42], except for the random initialisation being replaced with the current cloudlet deployment to avoid unnecessary actions. This approach was also described and evaluated in [25]. The different parameters of SA are described below.

Initial fraction accepted actions with loss The initial temperature is selected so that a given fraction of the actions with a negative gain will be accepted in the first epoch.

Temperature coefficient The temperature decreases geometrically between epochs, i.e. the temperature is multiplied with a temperature coefficient < 1 after each epoch.

Epoch coefficient The number of actions tried during each epoch is proportional to the total number of valid actions in the current cloudlet configuration, scaled with an epoch coefficient.

Fraction accepted actions threshold When the fraction of actions that got accepted during an epoch falls below this threshold, a stopcounter is increased. This stopcounter is reset when a globally better solution is found. Higher values of this threshold will make the algorithm return more quickly. For the sake of clarity, this logic was not included in Algorithm 2.

Stop threshold When the stopcounter itself exceeds the stop threshold, the algorithm terminates.

5.3. Stability

While any suggested action or set of actions that reduces the objective function can be applied to the cloudlet, the gain achieved from performing these actions may only be marginal. While not incorporated into the optimization problem, performing each action in an operational cloudlet comes with a cost. Two issues may arise. Software

Algorithm 2 Simulated Annealing

Input: *Initial*
Output: *Best*
  $Current \leftarrow Initial$
  $Best \leftarrow Initial$
  $T \leftarrow startTemperature(Initial)$
  $L \leftarrow epochLength(Initial)$
  repeat
    for L times **do**
      $K \leftarrow randomAction\,(Current)$
      $G \leftarrow gain\,(K, Current)$
      if accepted with probability $\exp\,(G/T)$ **then**
        $Current \leftarrow performAction\,(K, Current)$
        if $f_{obj}\,(Current) < f_{obj}\,(Best)$ **then**
          $Best \leftarrow Current$
        end if
      end if
    end for
    Decrease T
  until the stop criterion is met

components may be unavailable while offloading and any calls will need to be halted until the migration is complete. The resulting cost may be significantly higher than the realized gains.

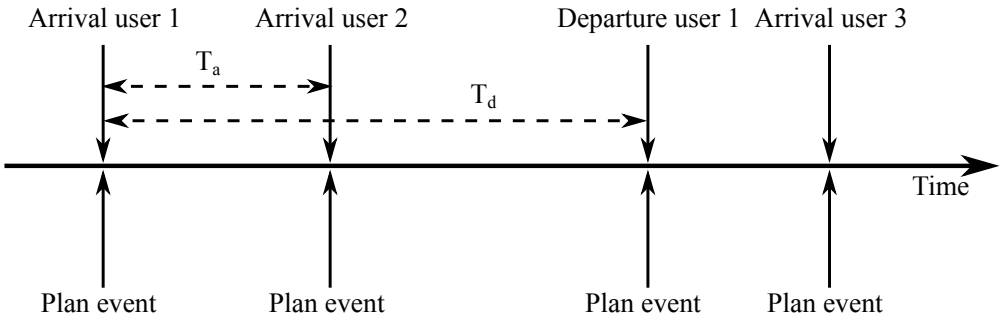
Another problem arises from the dynamic behaviour of the cloudlet. The effect of actions may be undone by new observations or varying user count. This may cause actions to be reverted and can lead to unwanted oscillations and therefore unstable behaviour. In order to improve the stability of the cloudlet, a hysteresis coefficient $\eta \leq 1$ is incorporated into the allocation algorithm and a proposed solution is only accepted if $f_{obj}\,(Best) \leq f_{max}$ with $f_{max} = \eta \cdot f_{obj}\,(Initial)$. This hysteresis coefficient ensures that a minimum relative gain needs to be achieved before the suggested actions are performed.

6. Discrete-event cloudlet simulator

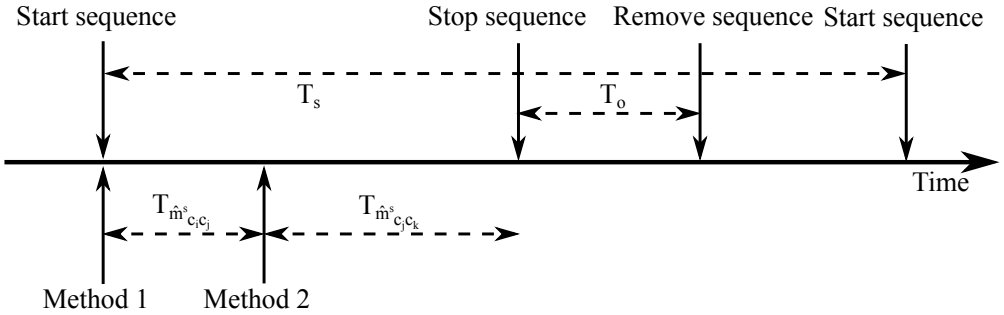
In order to thoroughly compare the proposed allocation algorithms, the dynamic behaviour of a collaborative cloudlet and the impact of the allocation algorithm thereon needs to be assessed. However, for cloudlets consisting of dozens of users, an experimental set-up quickly becomes too costly and time-consuming. Not only do we need to correctly configure dozens of devices, we also need to be able to meticulously repeat every user action to be able to compare algorithms or parameter settings. Hence the need for simulating a collaborative cloudlet arises. This allows to evaluate algorithm configurations in a large array of application scenarios and cloudlet sizes. We propose a discrete-event simulation that is able to capture the dynamic behaviour of the cloudlet. These events reflect the changes that may occur in a real cloudlet system on a method-call granularity.

6.1. Discrete-event model

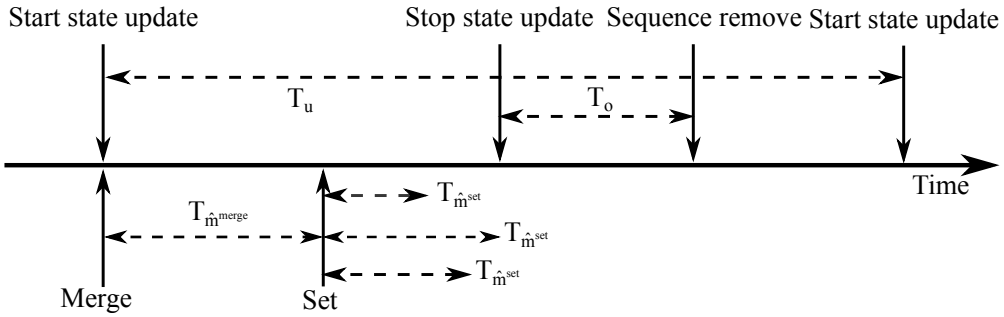
The discrete-event model consists of several types of events. The user-related events reflect the arrival and departure of users. The application-related events model the behaviour of the applications in the system, while special synchronization-related events are required for the exchange of synchronization messages. Finally, the management-related events model the autonomous control loop of the collaborative cloudlet middleware and the events related to the simulation itself. An overview of the event model is shown in Fig. 2. Each event, regardless of its type, has a time stamp at which it occurs in the simulated time frame and instructions to be processed at that time stamp. All events are processed chronologically.



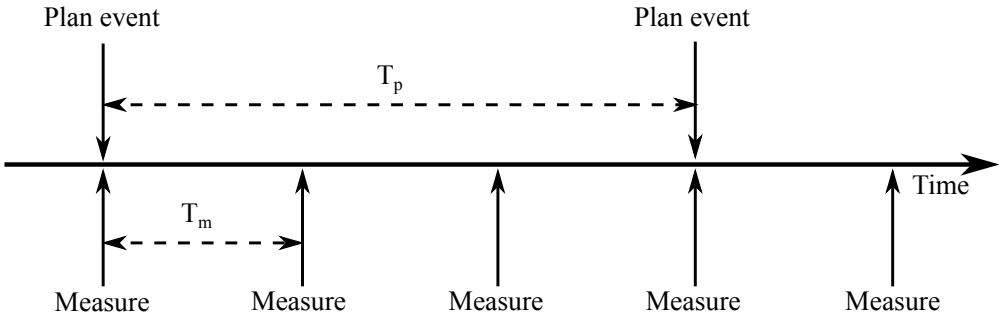
(a) User-related events. Both the inter-arrival time T_a and participation time T_d are stochastic variables. The plan events coinciding with the user events provide the reactive planning strategy.



(b) Application-related events. $T_{\hat{m}^s_{c_i c_j}}$ is the executing time of a method in the observed sequence, T_o is a constant observation window.



(c) Synchronization-related events. The stop state update event is only processed after all the sets have been fully executed.



(d) Management-related events. Both measure and plan events follow a periodic process, with T_p and T_m constant.

Figure 2: Overview of the discrete-event model used for cloudlet simulation.

6.1.1. User-related events

The user-related events model the arrival and departure of users in the simulated cloudlet system and are shown in Fig. 2a. At the start of the simulation no users are present in the cloudlet, so in turn no application components are present. A number of fixed nodes $\#D_0$ may be present for offloading components to. User arrivals are modelled as a stochastic process, with the inter-arrival times T_a being stochastic variables following a given distribution. When a user arrives, its node d_i and application components C_i are added to the cloudlet, deployed on the user node $\forall c \in C_i : X_{cd} = 1 \Leftrightarrow d = d_i$. For now we assume that each user is executing the same collaborative application, so application components are the same for every user. The component types and properties (i.e. sharing state and being offloadable) of this application are pre-determined parameters of the simulation.

The user's participation time T_d is also a stochastic variable sampled from a distribution. A user departure event can be scheduled based on his arrival and participation time. When a user leaves the cloudlet, components deployed on its device are removed. When a shared component or synchronization server is present on the user node, care needs to be taken to restore the collaboration as fast as possible. For this reason, and to ensure fast collaboration when a new user arrives, a reactive planning strategy is employed. At each user arrival or departure event, the allocation algorithm will be executed pre-emptively to ensure a valid cloudlet configuration. These plan events are discussed in Section 6.1.4.

6.1.2. Application-related events

Events of the application-type reflect the behaviour of the applications that are running in the cloudlet. These are shown in Fig. 2b. As discussed in Section 4.1, we use method call sequences to represent this behaviour. Before simulation, a number of method call sequences between components is generated to model possible application behaviour. A stochastic process will model the generation of observations of each particular sequence. Each sequence (possibly) has a different distribution of its inter-arrival times T_s. Inter-arrival times of observations of a particular sequence are sampled independently of the user generating the observation. Essentially, each additional user in the system initiates a stochastic process associated with each sequence.

For each method call in a sequence observation, a method event will represent the start of the execution of the call. At this time, the observed method load, argument and result size are sampled from their respective distributions. These distributions are different for each method. While the first call in the sequence starts executing simultaneously with the start sequence event, successive method calls will need to wait until the previous call has finished executing. The execution time $T_{\hat{m}^s_{c_ic_j}}$ of an observed method $\hat{m}^s_{c_ic_j}$ can be calculated as follows. Note that $\#calls_{m^s_{c_ic_j}}$ is inherent to the sequence and not to the specific observation.

$$T_{\hat{m}^s_{c_ic_j}} = \sum_{d \in D} \sum_{c_k \in C} X_{c_kd} \cdot Y_{c_jc_k} \cdot load_{\hat{m}^s_{c_ic_j}} \cdot \#calls_{m^s_{c_ic_j}} / speed_d \tag{16}$$

$T_{\hat{m}^s_{c_ic_j}}$ is only calculated when the respective method event is processed. The execution time depends on the allocation of components, which may change during the sequence and even during the execution of a method call. When all methods are executed, a sequence stop event signals the end of the sequence observation and the observed sequence is added to a set of observations. This set of observations is part of the input of the allocation algorithm in future plan events, along with the component deployment. Only a limited observation window T_o is considered and a sequence remove event is scheduled T_o after the stop event (T_o being constant), which will remove the sequence observation from the set. The frequency $freq_s$ of a sequence s is then estimated by counting the number of observations in this window.

6.1.3. Synchronization-related events

Synchronization messages can not simply be modelled using the same event-model as application-related events and thus require special attention. Sequence observations only occur between components of a single user (except when shared offloading is used, but $Y_{c_ic_j}$ makes this transparent for the event-model), while state synchronization involves multiple users. Where every sequence has a separate stochastic process modelling its observations, each component type that shares state has a separate stochastic process modelling its state updates. These state updates

have stochastic inter-arrival times T_u. Analogous to regular sequences, these stochastic processes are duplicated for each user as each client component generates synchronization events.

If state synchronization is actually used when processing a state update event, a merge event is generated that represents the start of the `mergeState` call $\hat{m}^{merge}_{c_i c_{server}}$, as shown in Fig. 2c. This call is executed by the current synchronization server and its execution time $T_{\hat{m}^{merge}_{c_i c_{server}}}$ can be estimated in similar way as for a regular method. After the merge is processed, a `setState` $\hat{m}^{set}_{c_{server} c_j}$ is performed on all the client components c_j. The assumption is made, however, that all these `setState`-invocations happen in parallel. As these invocations are performed on multiple nodes with possibly different hardware characteristics, their execution times may also be different. We generate a stop event only after all the `setState`-calls have finished. In this stop event, each `mergeState` and `setState` is added to the observations as a separate sequence because the sequence model requires sequential and not parallel method calls per sequence. After T_o, these sequences are again removed from the observation set. In the case that the synchronization server is changed during execution or the collaboration mechanism is switched to shared offloading, the synchronization sequences are not added to the observations, as would be if the state synchronization session are interrupted.

6.1.4. Management-related events

The management-related events describe the types of events that take care of the performance monitoring and autonomous control of the simulated collaborative cloudlet. An overview is shown in Fig. 2d. The performance monitoring is done in measure events, which will calculate the desired performance metrics based on the current set of observations. These include the generated traffic, average usage, the constraint violations, the number of users, and the cumulative number of actions performed. Due to the high computing cost of the metrics, calculating them on a per-event basis would slow down the simulation significantly. A periodic measuring process with period T_m was opted for instead.

Similarly, a periodic process with period T_p generates proactive plan events and models the MAPE-K autonomous control loop executing the allocation algorithm. The knowledge consists of the set of sequence observations and the allocation and substitution matrices. The monitoring step for the proactive plan events spans the intervals $\left[k \cdot T_p - T_o, k \cdot T_p\right], k \in \mathbb{N}$, while the analyze, plan and execute step take place the moment the plan event is processed. In the simulated cloudlet, these last steps are assumed to be instantaneous, implying that the execution time of the allocation algorithm is not incorporated in the simulator. Likewise, the actions suggested by the allocation algorithm are also performed at the same time stamp. The measure events coinciding with the plan events (see Fig. 2d) are processed after the suggested actions have been performed, and thus capture the expected effects of these actions.

While the main goal of the reactive plan events described in Section 6.1.1 is to maintain a valid cloudlet configuration, the goal of the proactive plan events is to optimize its configuration. Both types of events are necessary. Imagine, for example, a user node leaving on which a shared component instance or synchronization server is running. This would effectively interrupt the collaboration. By triggering a reactive plan event, however, one can immediately restore collaboration. In our case, when a synchronization server of shared component instance is removed from the cloudlet, we fall back to state synchronization with a randomly chosen synchronization server. On the other hand, if no proactive plan events were performed, the autonomous control of the cloudlet would depend entirely on the user arrival rate.

6.2. Assumptions

The cloudlet is essentially modelled as a $M/M/\infty$ queue, meaning that the user arrival process is a Poisson Process (PP) with exponentially distributed inter-arrival times and users participate an exponentially distributed time in the cloudlet. The exact distributions used during experiments are presented in Section 7.2. Node speed is assumed to follow a strictly-positive normal distribution. A separate distribution is used for fixed nodes and user nodes to represent the average difference in speed. The number of cores is uniformly distributed. The process modelling the start of each sequence is also a PP. However, as we want different sequences to have different arrival rates, we use a strictly-positive normal prior distribution on the parameter of the exponential distributions modelling the inter-arrival times. Similarly, the applications methods have an exponential prior distribution on the parameters of the strictly-positive normal distributions modelling the load, argument and result size of each method. The number of calls a method occurs in a sequence, as well as the number of different methods in the sequence, are both geometrically distributed.

The available bandwidth, number of application components and sequences follow a shifted Dirac distribution, i.e. they are a predetermined constant.

We further assume that for all distributions, including the inter-arrival times of the stochastic processes, all samples can be drawn independently of each other. While this may not always be the case for real-life applications, this suffices for evaluating and comparing the allocation algorithms. We also make the assumption that the $speed_d$ of all nodes and the *bandwidth* of the network can be measured in an operational system. Determining and updating these values at runtime would require periodic benchmarking, which adds additional load to the operational cloudlet. We instead assume that these values remain constant during the simulation. Furthermore the assumption is made that each user runs the same application in the cloudlet and interacts with it in the same way. This can however be extended in future work to multiple applications per user or different user behaviour for the same application.

6.3. Implementation

As discussed in Section 3, no existing cloud simulator provides the means to easily and efficiently simulate the long-lived sessions of mobile users and frequent calls that occur in a collaborative cloudlet middleware. Therefore, and because a Java implementation of the allocation algorithms was already available, we opted to implement our own cloudlet simulator in Java. For simplicity and efficiency, a custom discrete-event package was written, which however follows many of the design principles of discrete-event packages such as SimJava [43].

The cloudlet simulator has three important data-structures: an event queue, an observation blackboard and a parameter blackboard. The event queue is essentially a priority queue with the highest priority assigned to the event with the lowest time stamp. Events are pushed in and popped from the queue until the time stamp of the next event to be processed exceeds the desired simulated time. The observation blackboard contains, as the name implies, every sequence observation, as well as the properties of the nodes in the system and the variables X_{cd} and $Y_{c_i c_j}$. This blackboard provides the necessary input for the allocation algorithm. The parameter blackboard on the other hand contains all the parameters governing the dynamic cloudlet behaviour. This includes all the (prior) distributions, but also the generated application parameters such as the methods, sequence processes and component properties. These parameters are generated during initialization.

Random numbers are generated using the generator suggested by [44], which consists of two XOR-shift generators combined with a Linear Congruential Generator (LCG) and a multiply-with-carry generator. Using the parameters described in 7.2 and a default allocation algorithm, which only ensures correct state synchronization, we achieve a speed-up of about 70 compared to real-time at an event rate of about 8500 events/s on a Intel Core i5-3230M 2.6 GHz quad-core processor. Note that the simulator itself is implemented in a single thread, whereas the allocation algorithms are using multiple threads to achieve acceptable execution times.

7. Results and discussion

In this section, the performance of the allocation algorithms is evaluated using the model and discrete-event simulator presented in the previous section. While the main figures of merit are the average usage of the computational and network resources in the cloudlet, it is also important to minimize the number of constraint violations and the number of reallocation actions. Component reallocation affects user experience and introduces additional management overhead. First, we describe the algorithm configuration in Section 7.1 and simulated application scenarios in 7.2. These scenarios capture different application characteristics in terms of computational and network load. Next, we make a detailed comparison of SA and SD for a single scenario in Section 7.3 and discuss the effect of employing hysteresis for this scenario in Section 7.4. Finally, we extend the evaluation to multiple scenarios in Section 7.5.

7.1. Algorithm configuration

Both SA and SD have a number of parameters which need to be chosen carefully. The parameters have an influence on both the quality of the resulting deployment and the execution times needed to achieve this deployment. The goal of a heuristic allocation algorithm is to find the best possible solution in the least amount of time. While the execution time of the algorithm is not explicitly incorporated into the event-model of the simulation, it will have the largest impact when running in an operational cloudlet. Therefore the execution time still needs to be accounted for.

Table 1: Parameter settings for the SA algorithm.

Parameter	**Value**
Initial fraction accepted actions with loss	0.02
Temperature coefficient	0.05
Epoch coefficient	0.1
Fraction accepted actions threshold	0.5
Stop threshold	5

Initial values of the five parameters of the SA algorithm were selected based on previous work [12], where its performance and execution time were compared to an optimal, exhaustive algorithm. Tuning the SA parameters resulted in solutions within 2% of the optimum for an execution time of less than a second. However, the cloudlet sizes considered in [12] are smaller than the ones we want to simulate here, hence the parameters have to be further adapted to achieve feasible simulation times. The resulting SA-parameters are listed in Table 1. The only parameter of the SD algorithm, the iteration threshold, was set to 2. This results in similar execution times as the SA algorithm and ensures a fair comparison.

A default algorithm will be used to compare the performance achieved by the SA and SD algorithms. This algorithm will return a valid, but otherwise unoptimized, cloudlet configuration by selecting a random synchronization server for each component that shares state. When the user hosting the synchronization server leaves the cloudlet, another instance of the same type will be randomly selected as the new server. In other words, the default algorithm uses the minimal effort possible to establish collaboration. The comparison of our heuristics with this algorithm can show the benefits of simultaneously providing collaboration and runtime allocation optimization.

The performance of the algorithms will be evaluated by measuring the average usage, the generated traffic and the total relative constraint violation at each measure event. The total constraint violation is calculated as follows, with $\delta = \alpha + \beta + \gamma$.

$$\begin{aligned} f_{con} = & \frac{\alpha}{\delta} \cdot V\left(traffic, bandwidth\right) \\ & + \frac{\beta}{\delta} \cdot \sum_{d \in D} V\left(load_d, speed_d \cdot \#cores_d\right) \\ & + \frac{\gamma}{\delta} \cdot \sum_{d \in D} \sum_{s \in S} V\left(load_{sd}, speed_d\right) \end{aligned} \tag{17}$$

7.2. Application scenarios

Three different scenarios are simulated, which generate a different computational and network load. A timespan of one hour, or 3600 seconds, is simulated for all application scenarios. No users are present in the cloudlet at the start of the simulation. During simulation itself, events will be generated and processed as discussed in Section 6.1. At the plan events, one of three allocation algorithms (SA, SD or the default algorithm) will be executed and its suggested actions performed. As the focus of this paper lies on evaluating the performance of the allocation algorithms for different scenarios, we only perform a single simulation run for each configuration. As all the stochastic processes in the simulation are assumed to be memoryless, this should not affect the obtained results.

Table 2 shows the constant values used for all simulations. We assume a shared wireless network with a bandwidth of 200 Mbps and a single server co-located with the access point which can be used for offloading. The bandwidth was selected so that at each moment in time, all constraints can possibly be satisfied by correctly deploying components. The collaborative application that each user is executing is fixed to 8 components and 16 sequences.

A High-Load High-Traffic (HLHT) scenario, where an application generates a relatively high load and a large amount of traffic, is considered first. All stochastic variables and their distributions for this scenario are listed in Table 3. $\mathrm{U}(n)$ is the discrete uniform distribution in $[1, n]$, $\mathrm{Exp}(\lambda)$ is the exponential distribution with expected value $1/\lambda$. $\mathrm{Bern}(p)$ is the Bernoulli-distribution with expected value p, $\mathrm{G}_1(p)$ is the associated shifted geometric

Table 2: Selected values of all constants in the simulation.

Description	Symbol	Value
Simulator parameters		
Plan period (s)	T_p	30
Measure period (s)	T_m	1
Observation window (s)	T_o	30
Infrastructure parameters		
Fixed nodes	$\#D_0$	1
Fixed node speed (load/ms)	$speed_{d_0}$	4
Fixed node cores	$\#cores_{d_0}$	4
Bandwidth (Mbps)	$bandwidth$	200
Application parameters		
Application components	$\#C_i$	8
Application sequences	$\#S_i$	16

Table 3: Distributions of all stochastic variables for the HLHT application scenario. $\mathrm{F}(p_1, \mathrm{F}_2(p_2))$ represents a distribution with a constant parameter p_1 and a stochastic parameter which itself has a prior distribution $\mathrm{F}_2(p_2)$.

Description	Symbol	Distribution
User parameters		
User inter-arrival (ms)	T_a	$\mathrm{Exp}\left(1/60 \times 10^{-3}\right)$
User participation (ms)	T_d	$\mathrm{Exp}\left(1/600 \times 10^{-3}\right)$
User node speed (load/ms)	$speed_{d_i}$	$\mathrm{N}^+(1, 0.4)$
User node cores	$\#cores_{d_i}$	$\mathrm{U}(4)$
Application parameters		
Component is offloadable	N/A	$\mathrm{Bern}(2/3)$
Component shares state	N/A	$\mathrm{Bern}(1/2)$
Sequence size	$\#s$	$\mathrm{G}_1(1/4)$
Sequence parameters		
Sequence inter-arrival (ms)	T_s	$\mathrm{Exp}\left(\mathrm{N}^+\left(1 \times 10^{-4}, 1/4 \times 10^{-4}\right)\right)$
Method load (load)	$load_{\hat{m}^s_{c_i c_j}}$	$\mathrm{N}^+\left(\mathrm{Exp}\left(1/4 \times 10^{-1}\right), \mathrm{Exp}(1/6)\right)$
Method argument (bits)	$arg_{\hat{m}^s_{c_i c_j}}$	$\mathrm{N}^+\left(\mathrm{Exp}\left(1/8 \times 10^{-5}\right), \mathrm{Exp}\left(1/12 \times 10^{-4}\right)\right)$
Method result (bits)	$res_{\hat{m}^s_{c_i c_j}}$	$\mathrm{N}^+\left(\mathrm{Exp}\left(1/8 \times 10^{-4}\right), \mathrm{Exp}\left(1/12 \times 10^{-3}\right)\right)$
Method calls	$\#calls_{m^s_{c_j c_j}}$	$\mathrm{G}_1(1/4)$
Synchronization parameters		
State update inter-arrival (ms)	T_u	$\mathrm{Exp}\left(\mathrm{N}^+\left(1 \times 10^{-4}, 1/4 \times 10^{-4}\right)\right)$
Merge load (load)	$load_{\hat{m}^{merge}}$	$\mathrm{N}^+\left(\mathrm{Exp}\left(1/2 \times 10^{-1}\right), \mathrm{Exp}(1/3)\right)$
Set load (load)	$load_{\hat{m}^{set}}$	$\mathrm{N}^+\left(\mathrm{Exp}\left(1 \times 10^{-1}\right), \mathrm{Exp}\left(1/15 \times 10^{1}\right)\right)$
Update size (bits)	$arg_{\hat{m}^{merge}}$	$\mathrm{N}^+\left(\mathrm{Exp}\left(1/8 \times 10^{-5}\right), \mathrm{Exp}\left(1/12 \times 10^{-4}\right)\right)$

distribution. Finally, $\mathrm{N}^+(\mu, \sigma)$ represents the normal distribution $\mathrm{N}(\mu, \sigma)$, with expected value μ and variance σ^2, restricted to strictly-positive values.

Simulation with these parameters results in variations of the user count as illustrated in Fig. 3. The user count varies between 0 at the beginning of the simulation to 16 after an hour of simulation. A total of 54 user arrivals are

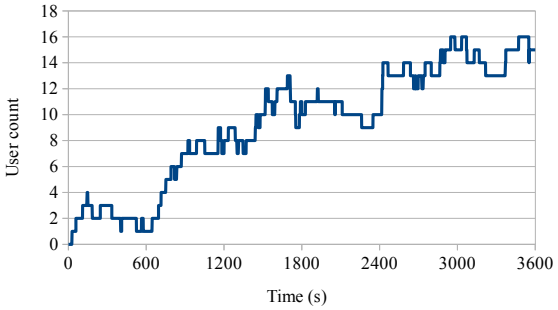


Figure 3: User count for all simulated scenarios.

Table 4: Selected distributions of all stochastic variables in the HLLT application scenario.

Description	**Symbol**	**Distribution**
Method argument (bits)	$arg_{\hat{m}^s_{c_i c_j}}$	$\mathrm{N}^+\left(\mathrm{Exp}\left(1/4\times10^{-5}\right),\mathrm{Exp}\left(1/6\times10^{-4}\right)\right)$
Method result (bits)	$res_{\hat{m}^s_{c_i c_j}}$	$\mathrm{N}^+\left(\mathrm{Exp}\left(1/4\times10^{-4}\right),\mathrm{Exp}\left(1/6\times10^{-3}\right)\right)$
Update size (bits)	$arg_{\hat{m}^{merge}}$	$\mathrm{N}^+\left(\mathrm{Exp}\left(1/4\times10^{-5}\right),\mathrm{Exp}\left(1/6\times10^{-4}\right)\right)$

Table 5: Selected distributions of all stochastic variables in the LLHT application scenario.

Description	**Symbol**	**Distribution**
Method load (load)	$load_{\hat{m}^s_{c_i c_j}}$	$\mathrm{N}^+\left(\mathrm{Exp}\left(1/2\times10^{-1}\right),\mathrm{Exp}\left(1/3\times10^{0}\right)\right)$
Merge load (load)	$load_{\hat{m}^{merge}}$	$\mathrm{N}^+\left(\mathrm{Exp}\left(1\times10^{-1}\right),\mathrm{Exp}\left(1/15\times10^{1}\right)\right)$
Set load (load)	$load_{\hat{m}^{set}}$	$\mathrm{N}^+\left(\mathrm{Exp}\left(2\times10^{-1}\right),\mathrm{Exp}\left(1/75\times10^{2}\right)\right)$

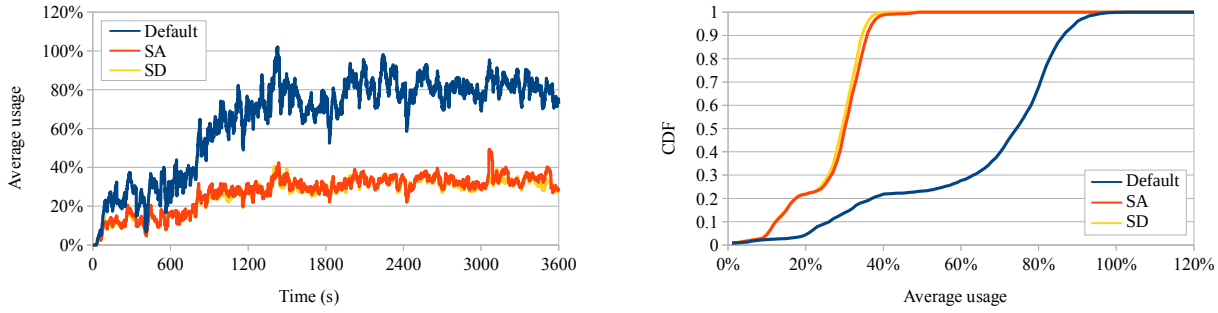
observed during this period, with an average of 9 users present at each time in the cloudlet. The long-term average according to Little's law is slightly higher, with 10 users.

Two other scenarios will be evaluated. In the High-Load Low-Traffic (HLLT) application scenario, the same load is generated as in the HLHT scenario but the amount of traffic is reduced so that the available bandwidth is plentiful. The opposite happens in the Low-Load High-Traffic (LLHT) scenario, where the amount of traffic stays the same as in the first scenario but the generated load is reduced. The used parameters for these scenarios are shown in Table 4 and Table 5 respectively. Parameters that are not mentioned in these tables are the same as in the HLHT scenario. The variation in user count remains the same as in Fig. 3.

7.3. Evaluation of the High-Load High-Traffic scenario

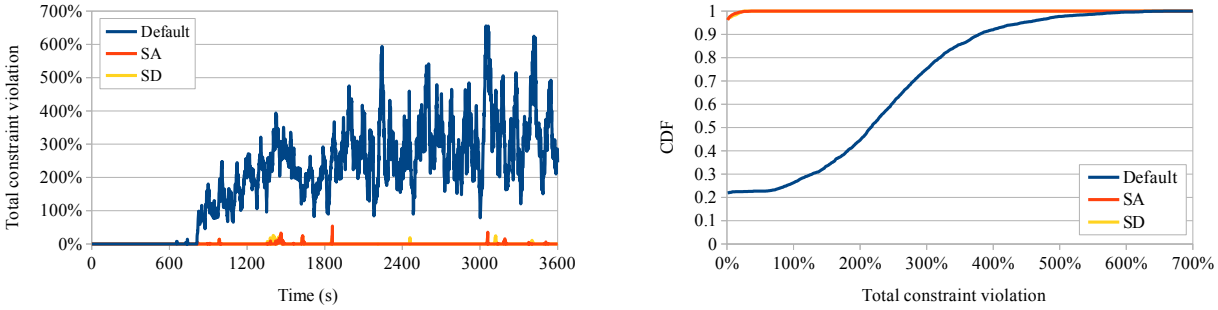
First a detailed comparison of the allocation algorithms is made for the HLHT scenario. Fig. 4a shows how the main optimization criterion, the average usage, varies as a function of time for the three algorithms. No hysteresis is used at this time. The default algorithm shows the average usage without optimization. We get an average of 65.2% over the entire hour. Note that the average usage may exceed 100% when nodes in the cloudlet become overloaded. When using the SA and SD algorithms, a significant decrease in the average usage is achieved, with 27.3% and 26.6% for the SA and SD algorithm. SD only performs slightly better than SA in terms of average usage, but their performance is generally very similar. This is confirmed when looking at the Cumulative Distribution Function (CDF) in Fig. 4b.

While the average usage is the main optimization criterion, it is also important to look at when, and to what extent, the constraints are violated. Fig. 5a shows the total constraint violation varying in time, Fig. 5b shows the accompanying CDF. We see that severe violations of the constraints occur when no optimization is in place, with



(a) (b)

Figure 4: The effect of using the SA and SD optimization algorithms on the average usage for the HLHT scenario. The usage varying in time is shown on the left. The right plots shows the CDF, which gives the fraction of the time the average usage is less then or equal to a given value.



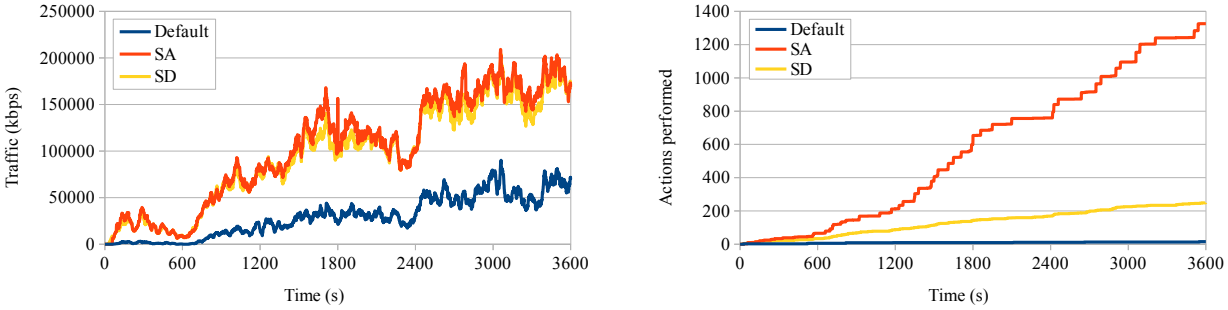
(a) (b)

Figure 5: The total constraint violation for the HLHT scenario is shown as a function of time on the left for all tested algorithms. The CDF is shown on the right.

violations occurring 78.3% of the time, and total violations reaching 655%. When using the SA or SD algorithm, constraint violations only occur 3.8% and 3.4% of the time, with total violations never exceeding 53% and 25% respectively. We again observe that SD performs slightly better than SA.

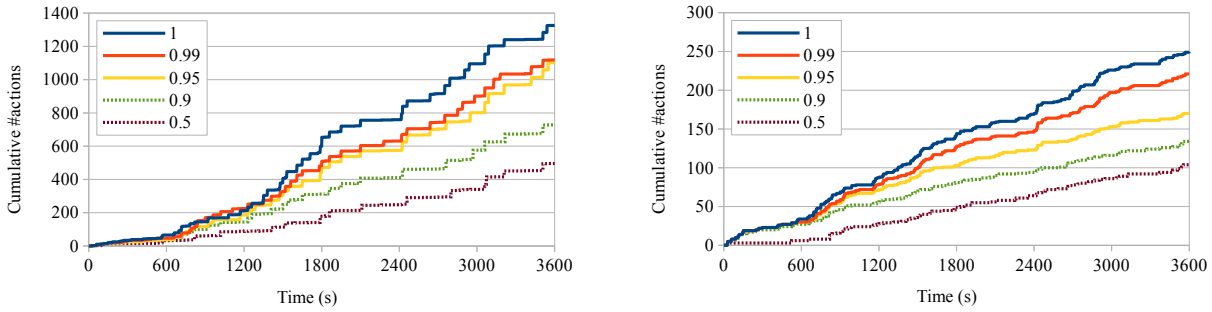
The generated traffic is shown in Fig. 6a. With an average of 96 Mbps for SD compared to 103 Mbps for SA, one could again conclude that SD performs slightly better, although the amount of traffic is only incorporated into the optimization problem as a constraint and is not a direct minimization objective.

A metric that is not included into the optimization problem, but is important for the cloudlet stability, is the number of actions that are performed. Fig. 6b shows the cumulative number of actions for each algorithm. With the default algorithm, a total of 16 actions need to be performed to maintain a correct configuration of the cloudlet. The difference between SA and SD becomes very clear: SA performs a total of 1326 actions, more than five times as much as SD, which only performs 250 actions. Clearly, SA will result in more unstable behaviour than SD, even though their performance is very similar otherwise. This is due to the random nature of SA: while the final result of the algorithm may be as good as SD, the path to find this solution is random and may contain many unnecessary actions that have little to no impact on the overall performance. The goal is now to improve the stability of the cloudlet by employing hysteresis and removing these unnecessary actions.



(a) (b)

Figure 6: The plot on the left shows the amount of generated traffic as a function of time for the HLHT scenario. The right plot shows the cumulative number of actions performed.



(a) SA (b) SD

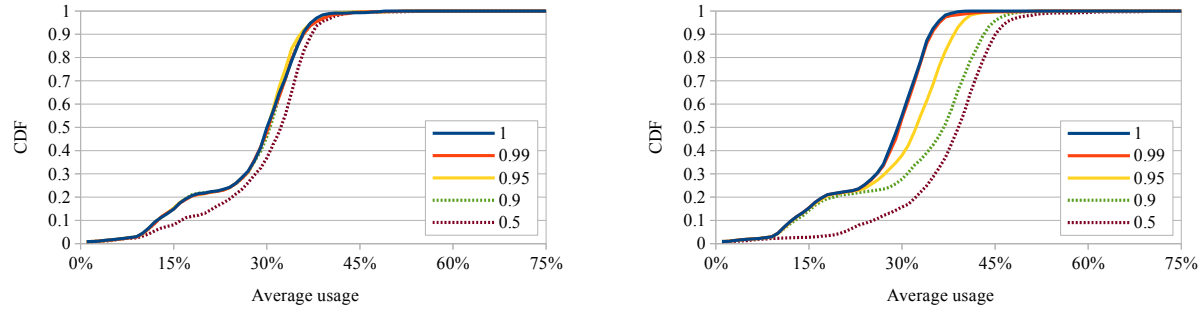
Figure 7: The effect of using a hysteresis coefficient $\eta \leq 1$ on the cumulative number of actions for the HLHT scenario.

7.4. Improving stability

Fig. 7 shows the cumulative number of actions for the HLHT scenario when using a hysteresis coefficient $\eta \leq 1$. Fig. 7a shows that for the SA algorithm, a minimum relative gain of only 1% can reduce the total number of actions by 15.7%. By further decreasing the hysteresis coefficient, the total number of actions is reduced even more. The amount of actions does however not decrease linearly with the hysteresis coefficient. For example, when decreasing η from 0.95 to 0.9, the number of actions is reduced by 374, while further decreasing η to 0.5 only further reduces the number of actions by 232. This implies that there is only a subset of actions that realize a significant decrease in the objective function. By choosing a small η, the performed actions are effectively reduced to this set. Fig. 7b shows the same observations for the SD algorithm. Note that for $\eta = 0.5$, SA still performs almost double the amount of actions as SD performs without hysteresis.

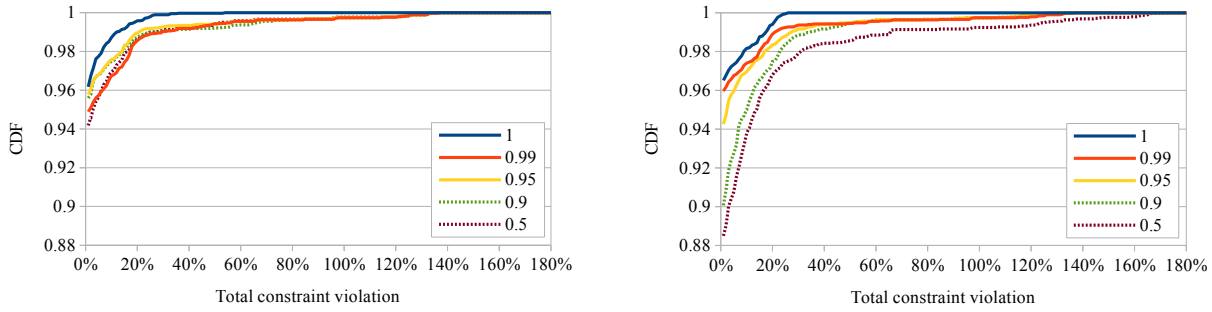
While reducing the number of performed actions will indeed increase the stability of the cloudlet, this impacts the objective minimisation and the constraints. Figure 8 shows what happens to the average usage CDF when applying hysteresis to both SA and SD. For SA, an impact is only noticeable when employing relatively extreme values of η. By requiring that a proposed solution has a minimum relative gain of 50% or more, solutions are only accepted when the cloudlet is already in a state of high constraint violations and/or high device usage and these kinds of gains in the objective function are possible. This effect is even more visible when looking at SD, as the impact on the average usage is already noticeable for a minimum relative gain of 5%. By construction, SD will suggest less actions to be performed and the quality of the result is hence more influenced by the hysteresis coefficient.

Fig. 9 shows the effect of hysteresis on the constraint violation CDF. For SD, both the fraction of time that



(a) SA (b) SD

Figure 8: While the use of a hysteresis coefficient with SA (left) does not noticeably affect the achieved average usage except for very low values, its impact is much greater with the SD (right) algorithm. Values shown are for the HLHT scenario.



(a) SA (b) SD

Figure 9: As on the average usage, the hysteresis coefficient only has a small impact on the total constraint violation when using the SA algorithm compared to SD. Values shown are for the HLHT scenario.

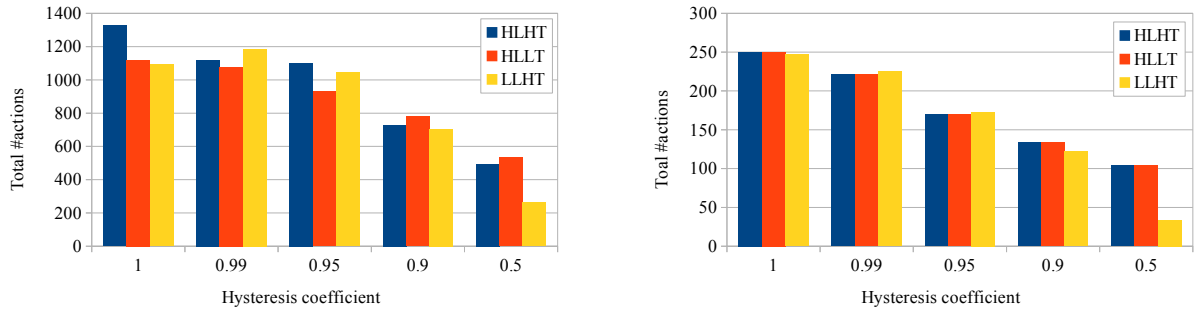
constraints are violated and the maximum observed violation increase with decreasing η, which is to be expected as less optimization takes place. As these increases are significant, the hysteresis coefficient should not be set any lower than 0.99. This still lowers the number of actions by 11.6%, down to 221. For SA, more erratic behaviour is observed, as for example $\eta = 0.99$ results in more frequent constraint violations than when $\eta = 0.9$. For $\eta \geq 0.9$ however, the constraints are only violated 1.3% of the time more than when no hysteresis is used. Keeping in mind the previous results, a hysteresis coefficient of 0.9 is best suited for the SA algorithm, which reduces the number of actions down to 728, a reduction of 45.1%.

While we are able to nearly halve the number of actions suggested by the SA algorithm without noticeably decreasing its performance, this is still more than double the number of actions suggested by the SD algorithm when no hysteresis is employed, while the same or slightly better performance is achieved.

7.5. Evaluation of the other scenarios

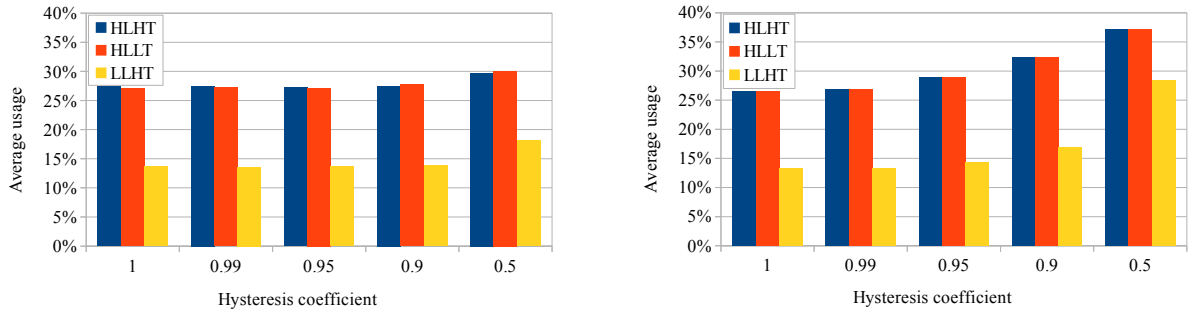
Fig. 10 shows the total number of actions that are performed for all the simulated scenarios, for both allocation algorithms and for different values of the hysteresis coefficient. While the number of actions of the SA algorithm vary more between scenarios compared to SD, it decreases in a similar way for each scenario with decreasing hysteresis coefficient for both SA and SD.

The impact on the global average usage of each scenario is shown in Fig. 11. For each scenario, a smaller hysteresis coefficient increases the global average usage. However, the impact is larger in the LLHT scenario. Using $\eta = 0.5$



(a) SA (b) SD

Figure 10: The total number of actions performed for each value of the hysteresis coefficient and the different scenarios.



(a) SA (b) SD

Figure 11: The effect on the average usage over time of different hysteresis coefficients for the three scenarios.

compared to no hysteresis, the global average usage increases by 2.3% (SA) and 10.5% (SD) for the HLHT scenario, 3.1% and 10.5% for the HLLT scenario and 4.5% and 15.1% for the LLHT scenario. This happens as a system put under relatively low load will most likely also have less infrastructure constraints that are violated. This implies that the relative share of the average usage in the objective function increases and thus more gains needs to be obtained by reducing the average usage rather than reducing constraint violations. This also implies that the hysteresis will have a greater impact on the average usage for low-load scenarios.

When comparing the SA algorithm to SD, the same observations from Section 7.3 hold. While the impact of hysteresis on the performance of SA is less severe than on SD, the number of moves can not be lowered enough to achieve the same quality of SD is equal or less moves.

8. Conclusion

In this paper we evaluated algorithms for the runtime optimization of a collaborative cloudlet middleware. The runtime optimization problem is formulated based on a theoretical model of the cloudlet, which describes the cloudlet infrastructure, application structure and application behaviour. In order to find a solution for this problem in an operational cloudlet system, two heuristic allocation algorithms based on Steepest Descent (SD) and Simulated Annealing (SA) are proposed. In order to evaluate these algorithms for different application scenarios and with dynamic cloudlet behaviour, a model for discrete-event simulation is proposed that captures the dynamics of the cloudlet per method call. We compare these algorithms by how well they minimize the average CPU usage, how well they satisfy the resource constraints and by the stability of the resulting allocation.

When simulating different application scenarios, we found that SD and SA yield very similar performance, with little difference in the average CPU usage and constraint violations. However, compared to SD, SA requires over five times more actions (e.g. component migrations) to be performed in total. This causes unstable behaviour, i.e. small changes in the observed application behaviour may result in many reallocations.

To improve the stability of the allocation, we introduced a hysteris factor enforcing a minimum gain before a reallocation is accepted, at the cost of a less optimal deployment. For SA, we found that the number of performed actions can almost be halved without any significant impact on performance. For SD, the number of actions can only be reduced by about 10% before the performance suffers noticeably. Moreover, the impact of employing hysteresis is greater for application scenarios with a relatively low generated load. In summary, to achieve the same quality of allocation compared to SA, SD requires less than halve as many actions to be performed.

In future work, other application scenarios can be evaluated, as well as scenarios involving different applications running in the cloudlet simultaneously. In this paper we assumed the network bandwidth to be constant. An interesting scenario would be to let the bandwidth vary over time as well, instead of only the user count. To further increase the stability of the cloudlet, the theoretical model used for runtime optimization could be extended to include the cost associated with the suggested actions.

Acknowledgment

Tim Verbelen is funded by Ph.D grant of the Fund for Scientific Research, Flanders (FWO-V). This project was partly funded by the UGent BOF-GOA project "Autonomic Networked Multimedia Systems".

[1] Gartner, Gartner Says Smartphone Sales Accounted for 55 Percent of Overall Mobile Phone Sales in Third Quarter of 2013 (2013) [cited 23 December 2013].
URL http://www.gartner.com/newsroom/id/2623415

[2] Samsung, Galaxy Gear V7000 (2013) [cited 23 December 2013].
URL http://www.samsung.com/nl/consumer/mobile-phone/mobile-phones/galaxy-gear/SM-V7000ZKAPHN/

[3] Sony, SmartWatch (2013) [cited 23 December 2013].
URL http://www.sonymobile.com/nl/products/accessories/smartwatch/

[4] Google, Glass (2013) [cited 23 December 2013].
URL http://www.google.com/glass/start/

[5] M. Satyanarayanan, Mobile computing: the next decade, in: Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services Social Networks and Beyond (MCS '10), ACM Press, 2010, pp. 1–6. doi:10.1145/1810931.1810936.

[6] Nokia Solutions and Networks, Increasing Mobile Operators' Value Proposition With Edge Computing, Tech. rep. (2013).
URL http://nsn.com/portfolio/liquid-net/intelligent-broadband-management/liquid-applications

[7] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, H.-I. Yang, The case for cyber foraging, in: Proceedings of the 10th workshop on ACM SIGOPS European Workshop (EW '10), ACM Press, 2002, pp. 87–92. doi:10.1145/1133373.1133390.

[8] M. Satyanarayanan, P. Bahl, R. Caceres, N. Davies, The case for VM-based cloudlets in mobile computing, IEEE Pervasive Computing 8 (4) (2009) 14–23. doi:10.1109/MPRV.2009.82.

[9] T. Verbelen, P. Simoens, F. De Turck, B. Dhoedt, AIOLOS: middleware for improving mobile application performance through cyber foraging, Journal Of Systems And Software 85 (11) (2012) 2629–2639.

[10] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, G. Alonso, Calling the cloud: Enabling mobile phones as interfaces to cloud applications, in: J. Bacon, B. Cooper (Eds.), Middleware 2009, Vol. 5896 of Lecture Notes in Computer Science, Springer-Verlag, 2009, pp. 83–102. doi:10.1007/978-3-642-10445-9_5.

[11] Niantic Labs, Ingress (2012) [cited 23 December 2013].
URL http://www.ingress.com/

[12] S. Bohez, J. D. Turck, T. Verbelen, P. Simoens, B. Dhoedt, Mobile , Collaborative Augmented Reality using Cloudlets, in: Proceedings of the 6th International Conference on Mobile Wireless Middleware, Operating Systems and Applications (MobilWare '13), Bologna, Italy, 2013, pp. 1–10.

[13] T. Verbelen, P. Simoens, F. De Turck, B. Dhoedt, Cloudlets: bringing the cloud to the mobile user, in: Proceedings of the 3rd ACM Workshop on Mobile Cloud Computing and Services, ACM Press, 2012, pp. 29–35.

[14] J. Flinn, P. SoYoung, M. Satyanarayanan, Balancing performance, energy, and quality in pervasive computing, in: Proceedings of the 22nd International Conference on Distributed Computing Systems, IEEE Computer Society, Vienna, Austria, 2002, pp. 217–226. doi:10.1109/ICDCS.2002.1022259.

[15] R. K. Balan, M. Satyanarayanan, S. Y. Park, T. Okoshi, Tactics-based remote execution for mobile computing, in: Proceedings of the 1st international conference on Mobile systems applications and services (MobiSys '03), ACM Press, San Francisco, California, USA, 2003, pp. 273–286. doi:10.1145/1066116.1066125.

[16] K. Ha, P. Pillai, W. Richter, Y. Abe, M. Satyanarayanan, Just-in-time provisioning for cyber foraging, in: Proceeding of the 11th annual international conference on Mobile systems, applications, and service (MobiSys '13), ACM Press, Taipei, Taiwan, 2013, pp. 153–166. doi:10.1145/2462456.2464451.

[17] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, A. Patti, CloneCloud: elastic execution between mobile device and cloud, in: Proceedings of the sixth conference on Computer systems (EuroSys '11), ACM Press, Salzburg, Austria, 2011, p. 301. doi:10.1145/1966445.1966473.

[18] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, X. Chen, COMET: code offload by migrating execution transparently, in: Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation, USENIX Association, 2012, pp. 93–106.

[19] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, P. Bahl, MAUI: making smartphones last longer with code offload, in: Proceedings of the 8th international conference on Mobile systems, applications, and services (MobiSys '10), ACM Press, 2010, pp. 49–62. doi:10.1145/1814433.1814441.

[20] M. Daro Kristensen, Scavenger: Transparent development of efficient cyber foraging applications, in: Proceedings of the 2010 IEEE International Conference on Pervasive Computing and Communications (PerCom '10), IEEE, 2010, pp. 217–226. doi:10.1109/PERCOM.2010.5466972.

[21] J. S. Rellermeyer, O. Riva, G. Alonso, AlfredO: an architecture for flexible interaction with electronic devices, in: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware, Springer-Verlag, 2008, pp. 22–41.

[22] X. Zhang, A. Kunjithapatham, S. Jeong, S. Gibbs, Towards an Elastic Application Model for Augmenting the Computing Capabilities of Mobile Devices with Cloud Computing, Mobile Networks and Applications 16 (3) (2011) 270–284. doi:10.1007/s11036-011-0305-7.

[23] M. Sharifi, S. Kafaie, O. Kashefi, A Survey and Taxonomy of Cyber Foraging of Mobile Devices, IEEE Communications Surveys & Tutorials 14 (4) (2012) 1232–1243. doi:10.1109/SURV.2011.111411.00016.

[24] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, R. Govindan, Odessa: enabling interactive perception applications on mobile devices, in: Proceedings of the 9th international conference on Mobile systems, applications, and services (MobiSys '11), ACM Press, Bethesda, Maryland, USA, 2011, pp. 43–56. doi:10.1145/1999995.2000000.

[25] T. Verbelen, T. Stevens, F. De Turck, B. Dhoedt, Graph partitioning algorithms for optimizing software deployment in mobile cloud computing, Future Generation Computer Systems 29 (2) (2013) 451–459. doi:http://dx.doi.org/10.1016/j.future.2012.07.003.

[26] I. A. Moschakis, H. D. Karatza, Towards scheduling for internet-of-things applications on clouds: a simulated annealing approach, Concurrency and Computation: Practice and Experiencedoi:10.1002/cpe.3105.

[27] J. Zhang, X. Zhu, B. Ying, A task scheduling algorithm considering bandwidth competition in cloud computing, in: M. Pathan, G. Wei, G. Fortino (Eds.), Internet and Distributed Computing Systems, Vol. 8223 of Lecture Notes in Computer Science, Springer-Verlag, 2013, pp. 270–280. doi:10.1007/978-3-642-41428-2_22.

[28] M. R. Rahimi, N. Venkatasubramanian, S. Mehrotra, A. V. Vasilakos, Mapcloud: Mobile applications on an elastic and scalable 2-tier cloud architecture, in: Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing, UCC '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 83–90. doi:10.1109/UCC.2012.25.

[29] V. Sacramento, M. Endler, H. Rubinsztejn, L. Lima, K. Goncalves, F. Nascimento, G. Bueno, MoCA: A Middleware for Developing Collaborative Applications for Mobile Users, IEEE Distributed Systems Online 05 (10) (2004) 2–2. doi:10.1109/MDSO.2004.26.

[30] G. Cugola, G. Picco, Peer-to-peer for collaborative applications, in: Proceedings 22nd International Conference on Distributed Computing Systems Workshops, IEEE Computer Society, 2002, pp. 359–364. doi:10.1109/ICDCSW.2002.1030795.

[31] M. Endler, G. Baptista, L. D. Silva, R. Vasconcelos, M. Malcher, V. Pantoja, V. Pinheiro, J. Viterbo, Contextnet: Context reasoning and sharing middleware for large-scale pervasive collaboration and social networking, in: Proceedings of the Workshop on Posters and Demos Track of the ACM/USENIX Middleware Conference, ACM Press, 2011, pp. 2:1–2:2. doi:10.1145/2088960.2088962.

[32] L. David, R. Vasconcelos, L. Alves, R. Andre, G. Baptista, M. Endler, A communication middleware for scalable real-time mobile collaboration, in: IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2012, pp. 54–59. doi:10.1109/WETICE.2012.70.

[33] J. M. Marques, X. Vilajosana, T. Daradoumis, L. Navarro, LaCOLLA: Middleware for Self-Sufficient Online Collaboration, IEEE Internet Computing 11 (2) (2007) 56–64. doi:10.1109/MIC.2007.43.

[34] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, R. Buyya, Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Software: Practice and Experience 41 (1) (2011) 23–50. doi:10.1002/spe.995.

[35] R. Buyya, M. Murshed, Gridsim: a toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing, Concurrency and Computation: Practice and Experience 14 (13-15) (2002) 1175–1220. doi:10.1002/cpe.710.

[36] D. Kliazovich, P. Bouvry, S. Khan, Greencloud: a packet-level simulator of energy-aware cloud computing data centers, The Journal of Supercomputing 62 (3) (2012) 1263–1283. doi:10.1007/s11227-010-0504-1.

[37] S.-H. Lim, B. Sharma, G. Nam, E. K. Kim, C. Das, Mdcsim: A multi-tier data center simulation, platform, in: Proceedings of the 2009 IEEE International Conference on Cluster Computing and Workshops (CLUSTER '09), IEEE Computer Society, 2009, pp. 1–9. doi:10.1109/CLUSTR.2009.5289159.

[38] A. Nez, J. Vzquez-Poletti, A. Caminero, G. Casta, J. Carretero, I. Llorente, icancloud: A flexible and scalable cloud infrastructure simulator, Journal of Grid Computing 10 (1) (2012) 185–209. doi:10.1007/s10723-012-9208-5.

[39] G. Sakellari, G. Loukas, A survey of mathematical models, simulation approaches and testbeds used for research in cloud computing, Simulation Modelling Practice and Theory 39 (0) (2013) 92–103, s.I. Energy efficiency in grids and clouds. doi:10.1016/j.simpat.2013.04.002.

[40] J. Kephart, D. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50. doi:10.1109/MC.2003.1160055.

[41] T. Verbelen, P. Simoens, F. De Turck, B. Dhoedt, Adaptive application configuration and distribution in mobile cloudlet middleware, in: Proceedings of the 5th International Conference on Mobile Wireless Middleware, Operating Systems and Applications, Berlin, Germany, 2012, pp. 1–14.

[42] D. S. Johnson, C. R. Aragon, L. A. McGeoch, C. Schevon, Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning, Operations Research 37 (6) (1989) 865–892.

[43] R. Mcnab, F. W. Howell, Using java for discrete event simulation, in: in Proc. Twelfth UK Computer and Telecommunications Performance Engineering Workshop (UKPEW), Univ. of Edinburgh, 1996, pp. 219–228.

[44] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, Numerical Recipes 3rd Edition: The Art of Scientific Computing, 3rd Edition, Cambridge University Press, New York, NY, USA, 2007.