View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Ghent University Academic Bibliography

Allocation Algorithms for Autonomous Management of Collaborative Cloudlets

Steven Bohez*, Tim Verbelen*, Pieter Simoens*,† and Bart Dhoedt*
*Department of Information Technology
Internet Based Communication Networks and Services (IBCN)
Ghent University - iMinds
Gaston Crommenlaan 8/201
B-9050 Ghent, Belgium
†Department of Industrial Technology and Construction
Ghent University College
Valentin Vaerwyckweg 1
B-9000 Ghent, Belgium
Email: {steven.bohez},{tim.verbelen},
{pieter.simoens},{bart.dhoedt}@intec.ugent.be

***Abstract*—Mobile applications are evolving towards support for advanced interactivity and resource-demanding multimedia features. Mobile platforms are however struggling to cope with these new innovative application concepts, such as Augmented Reality, due to the inherent limitations on their available resources, such as CPU, memory and battery power. Offloading resource-intensive calculations to nearby infrastructure or devices, also known as cloudlets, has emerged as a viable alternative in offering interactive and resource-intensive applications to mobile users.**

This concept of resource sharing provides promising opportunities for collaborative scenarios in which not only data processing, but also the data itself are shared between multiple users. In this paper, we investigate the challenges posed by offloading collaborative mobile applications. We describe the problem of autonomous management of collaborative cloudlets and propose two heuristic algorithms, Simulated Annealing and Steepest Descent, in order to solve this optimization problem. We observe that these heuristics yield an effectiveness of more than 90% for execution times that are three orders of magnitude lower when compared to a guaranteed-optimal approach.

I. Introduction

Mobile devices were among the fastest growing market segments in recent years, and with billions of smartphones and tablets to be sold in 2013 [1], there are no signs of this trend changing. The popularity of these devices has several reasons: not only are they portable and always-connected, but there are also hundreds of thousands of easy-to-install applications. These mobile applications have recently been evolving to highly interactive and media-rich experiences, such as immersive 3D games and Augmented Reality (AR). However, although recent advances in mobile processors and battery technology, mobile devices are struggling to cope with such applications due to their inherently limited resources. With the advent of wearable computers such as smart glasses and watches, resource limitations will continue to impact application capabilities.

To cope with these limitations, offloading (parts of) the application to infrastructure in the network becomes a necessity. Offloading to a distant cloud is however infeasible for highly interactive applications due to the large network delay. To offload such interactive applications, the concept op cyber foraging [2] was introduced, where a mobile user can benefit from resources available in his near vicinity, for example computing infrastructure co-located with the wireless access point. In this set-up, the user should not experience any significant network delay.

Cyber foraging can be realised using a cloudlet [3], a personalized Virtual Machine (VM) on a nearby, trusted server. The VM-based cloudlet concept has recently evolved to component-based cloudlets consisting of a group of computing nodes (both fixed and mobile) that are sharing resources with one another. Software components on the mobile device can then be redeployed at runtime to other nodes in the cloudlet according to some optimization criteria, such as the execution time [4], [5], energy consumption [6], and/or throughput [7].

Collaborative applications are also growing in number. These are applications involving multiple users interacting with each other, often in a real-time fashion. Niantic Labs' Ingress [8] is an example of a collaborative AR game with over half a million active users. The resource-sharing concept of component-based cloudlets offers a promising opportunity for collaborative applications: besides only sharing computing resources, users may also share data such as processing results and context information. For example, in a collaborative game, users could share the same virtual space and interact with the same virtual objects.

However, no existing cloudlet or cyber foraging system exploits this opportunity. In [9], we extend a component-based cloudlet middleware to a collaborative cloudlet by providing support for collaboration from the middleware. The middleware and collaboration are described in Sections III and IV.

An important aspect of a cloudlet system is autonomous management, the capability to decide at runtime how to allocate software components and configure collaboration. In this paper we propose two heuristic allocation algorithms, based on Simulated Annealing (SA) and Steepest Descent (SD), capable of minimizing the average device usage in the cloudlet while taking into account all necessary state synchronization. These algorithms are described in detail in Section V whereas in Section VI they are compared to an optimal approach.

II. Related work

Some of the first cyber foraging systems to be introduced, such as Spectra [10] and Chroma [11], required the application developer to pre-install routines on devices offering offloading. This approach quickly becomes infeasible for multiple applications, so runtime code migration becomes a requirement for realistic scenarios.

More recent cyber foraging and cloudlet middleware can be largely separated in VM-based and component-based approaches. Systems using VMs "copy" entire applications from mobile devices to nearby infrastructure. Either the mobile application is executed entirely in the dedicated VM, such as in the VM-based cloudlets in [12] and [13] or the decision is made on a per-routine basis, per example in CloneCloud [14] by using profiling and code analysis. COMET [15] uses a Distributed Shared Memory (DSM) approach which allows for easy migration of individual application threads between VMs.

Recently, more component-based systems have emerged. Using components instead of VMs provides more flexibility for application redeployment. 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 [6] and Scavenger [4]. The components can be larger, such as Open Services Gateway initiative (OSGi) bundles in AlfredO [16] and AIOLOS [5]. Weblets, RESTful services, are used by Zhang et al. [17]. An in-depth comparison of cyber foraging and other mobile cloud computing systems is given in [18].

Collaborative applications, also called groupware, are applications where multiple users work together in a shared context or state to accomplish a common goal. These applications face additional challenges when executed in a mobile environment (e.g. dynamically joining and leaving users). MoCa [19] is a middleware system specifically focused on mobile collaborative applications and uses a static client-server application architecture. In [20], it is however argued that a peer-to-peer architecture is more suited for mobile collaborative applications in order to remove the need for centralized infrastructure and improve flexibility.

The algorithms used to allocate software components are often based on well-known graph-partitioning algorithms [21], although other techniques for runtime optimization have been used successfully, such as Naive Bayesian Learning in [17]. Autonomously managing collaborative cloudlets, however, poses additional challenges, as we have to correctly decide on how to configure collaboration.

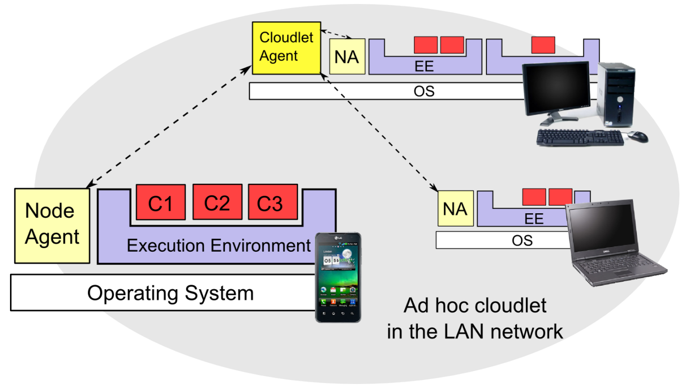

Fig. 1. General architecture of the cloudlet framework presented in [22].

III. Cloudlet middleware

We adopt the component-based cloudlet framework proposed in [22], [23] to create a collaborative cloudlet middleware. This cloudlet framework already provides the necessary functionality to offload software components from user devices to nearby computing infrastructure.

The general architecture of the framework is shown in Fig. 1. A cloudlet application is split up into several loosely-coupled software components. The components are managed by an Execution Environment (EE), which will connect them according to their offered and required services. Each application instance runs in its own EE. The EE can start, stop and reconnect components at runtime, which allows migrating of individual components between EEs (further called solitary offloading). Method calls between components deployed on different EEs are intercepted by proxies in order to execute them as Remote Procedure Calls (RPCs).

Every device in the cloudlet is represented as a node and runs a single Node Agent (NA). The NA starts, stops and manages the EEs on the device and collects monitoring information about the device. A single node in the cloudlet (e.g. the node with the greatest computing capacity) is selected to run a Cloudlet Agent (CA). This CA is responsible for autonomously managing the entire cloudlet and will aggregate the monitoring information from the different NAs and EEs in order to make decisions on when and where it needs to offload components. These decisions are made by an allocation algorithm, which will try to optimize a (combination of) metric(s) of the cloudlet, such as the average CPU usage of all nodes.

This middleware is extended with support for collaboration in Section IV by providing two mechanisms, state synchronization and shared offloading. However, these mechanisms introduce additional degrees of freedom to configure the cloudlet and to allocate components. In Section V, algorithms are discussed that are able to determine suitable component allocations and collaboration mechanisms.

IV. Collaboration

Collaboration between multiple users requires the exchange of information between different application instances. While this can be done by exchanging messages on the application level, it can be a tedious task for the application developer to implement message distribution and handle arrival and departure of users correctly. For this reason, collaboration is preferably supported by the middleware framework. Exchanging information between application instances can be abstracted to sharing some common state between application instances.

We extend the component-based cloudlet discussed in Section III with mechanisms to have a consistent state between multiple application instances. We propose two such mechanisms: shared offloading and state synchronization. To benefit from these mechanisms, the application developer only has to define the state of a component and how state updates of multiple sources can be merged in a consistent way.

A. Shared offloading

The cloudlet middleware allows us to dynamically change component references, even across application instances. This means a single component instance can easily be shared between multiple users by changing the required references. Sharing a component instance, or shared offloading, can be seen as an extension to solitary offloading, where now multiple users offload to the same component instance. Shared offloading is performed in three steps. First, a shared instance is created on the target node. Next, all other component instances push their current state to this shared instance where these states can be merged using an application- or component-specific method. Finally all references are changed to point to the newly-created shared instance. During offloading, all method calls to components are blocked until the process is finished to ensure no state information is lost.

As multiple application instances are now using the same component instance, and hence the same component state, collaboration can be guaranteed. With no further exchange of information necessary after the initial set-up, shared offloading can be seen as a passive mechanism. As multiple nodes are now referring to the same component instance, this also implies that on all except one node, RPCs will need to be performed. Depending on the size of the argument and result values, this may cause significant traffic. Also, for some components, shared offloading is not feasible due to strict timing constraints or not possible due to hardware dependencies. To still guarantee a consistent state between instances of these types of components, a second mechanism is proposed in the form of state synchronization.

B. State synchronization

With the state synchronization mechanism, multiple component instances of the same type, but allocated on different nodes in the cloudlet, can still share their state by actively exchanging messages. This means that component instances can remain allocated on user devices, method calls can be executed locally, but users can still collaborate. However, as state changes can now happen in a distributed, asynchronous way, precautions need to be taken on how to distribute state updates.

We adopt a client-server synchronization mechanism, where one instance of the component type to be synchronized, is selected as the synchronization server. This instance then becomes responsible for correctly distributing the state updates of other components and resolving any conflicts. Other instances can notify the server at any time of a state update that needs to be propagated using a service provided by the middleware. The server uses a component-specific method to process these updates (e.g. buffer and merge them). Finally, the corrected state update will be propagated to all clients where their local state will be adjusted accordingly. The EE is able to transparently connect the clients with their respective servers. Techniques such as incremental updates and revision histories are not offered by the framework at this stage, but can be implemented on top of the state synchronization mechanism by the application developer.

V. Allocation algorithms

The collaborative cloudlets previously introduced offer various possibilities for configuring collaboration and component deployment. In addition to offloading individual software components, a suitable collaboration mechanism needs to be selected for components that need to share state. 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.

However, due to changes in the available devices in the cloudlet and the load produced by applications, the configuration of the cloudlet should be adaptable at runtime. This makes manual configuration typically infeasible and requires an autonomous control loop. A well-known approach is to use a Monitor-Analyze-Plan-Execute (MAPE) control loop as proposed by [24]. Here, monitoring information is periodically fed to an allocation algorithm which will decide which actions need to be taken to optimize the cloudlet.

A. Cloudlet model

The allocation algorithm can be seen as solving an optimization problem, where a certain objective function is (in this case) minimized. This optimization problem can be formulated using a mathematical model of the collaborative cloudlet. We extend the cloudlet model presented in [23] 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.

The infrastructure of the cloudlet consists of the devices, or computing nodes, and the network. The computing nodes can be grouped in a set D, with every node $d \in D$ having two properties: the speed $speed_d$ at which it can process a load on a single core and the number of cores $\#cores_d$. The assumption is made that the network consists of a single shared medium, so the network can be represented by its capacity $bandwidth$.

Note that while actual node speed and network capacity may change over time, we only consider the current value at the moment of optimization.

The application structure consists of a number of component instances c. All components in the cloudlet are grouped in the component set C. To track the current allocation of individual components, we define X_{cd} as an allocation matrix.

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

Using X_{cd}, we define H_{ij} as being 1 if and only if components c_i and c_j are allocated on different nodes. This implies that any communication between these components has to go over the network.

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

In order to incorporate shared offloading into the model, we define an additional variable Y_{ij} to be 1 if every call to c_i is actually performed by c_j, in other words if c_i is currently being substituted by c_j. After shared offloading a set components C' to a shared instance c_{shared}, it holds that $Y_{ij} = 1 \Leftrightarrow c_j = c_{shared}, \forall c_i \in C'$.

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

The behaviour is modelled using the concept of sequences. A sequence is a succession of method calls between component instances and follows a specific path in the control flow graph of the application. Sequences are assumed to be executed in a single thread. Each sequence $s \in S$ consists of a number of successive method calls $m_{sc_ic_j}$ between components c_i and c_j and occurs with a certain frequency $freq_s$. Every method call has a certain load $load_{m_{sc_ic_j}}$ that needs to be processed, an argument $arg_{m_{sc_ic_j}}$ and result $res_{m_{sc_ic_j}}$ size and occurs $\#calls_{m_{sc_ic_j}}$ times in the sequence (e.g. in a loop).

B. Objective

Based on the cloudlet model described above, an objective function can be defined that needs to be optimized. We choose the objective function to be the average relative usage of all the nodes in the cloudlet.

$$f = usage_{avg} \tag{4}$$

By minimizing this objective function, the load on the mobile devices can be reduced while not overloading the high-capacity nodes (to provide a safety margin). For the same relative usage on a mobile and high-capacity node, the high-capacity node is able to process more load as it has a higher speed and/or more cores.

The average relative usage $usage_{avg}$ is calculated by averaging the relative usage $usage_d$ of all the nodes in the cloudlet.

$$usage_{avg} = \frac{\sum_{d \in D} usage_d}{\#D} \tag{5}$$

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, which is the number of cores times the speed of a single core.

$$usage_d = \frac{load_d}{speed_d \cdot \#cores_d} \tag{6}$$

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

$$load_d = \sum_{s \in S} load_{sd} \tag{7}$$

The load per unit of time of a sequence s on a specific node d can be found by checking, for every method call $m_{sc_ic_j}$ in the sequence, where the actual component that processes the call is allocated. Only when that component is deployed on d is the generated load taken into account.

$$\begin{aligned} load_{sd} = & \sum_{c_i \in C} \sum_{c_j \in C} \sum_{c_k \in C} X_{c_k d} \cdot Y_{jk} \\ & \cdot load_{m_{sc_ic_j}} \cdot \#calls_{m_{sc_ic_j}} \cdot freq_s \end{aligned} \tag{8}$$

C. Constraints

Besides the objective we want to minimize, there are certain constraints that need to be taken into account while solving the optimization problem. These constraints reflect the limited capacity of the cloudlet infrastructure. A first constraint is that the total load imposed on any node cannot exceed its total capacity.

$$load_d \leq speed_d \cdot \#cores_d, \forall d \in D \tag{9}$$

The load generated by a single sequence is assumed to be processed in a single-threaded fashion, as it is a succession of methods in time. This means that the load of any single sequence may not exceed the core speed of any node in the cloudlet.

$$load_{sd} \leq speed_d, \forall s \in S, \forall d \in D \tag{10}$$

Finally, the network must be able to cope with the generated traffic, i.e. it must fit within the available bandwidth.

$$traffic \leq bandwidth \tag{11}$$

The amount of generated traffic can be calculated by summing the argument and result sizes of each observed RPC (i.e. $\forall m_{sc_ic_j} : H_{ij} = 1$, taking substitutions into account).

$$traffic = \sum_{s\in S}\sum_{c_i\in C}\sum_{c_j\in C}\sum_{c_k\in C}\sum_{c_l\in C} H_{kl}\cdot Y_{ik}\cdot Y_{jl} \cdot \left(arg_{m_{sc_ic_j}} + res_{m_{sc_ic_j}}\right) \cdot \#calls_{m_{sc_ic_j}} \cdot freq_s \quad (12)$$

These constraints are incorporated into the optimization problem as additional terms in the objective function. This gives the following, complete objective function f.

$$f = usage_{avg} + \alpha\cdot W\left(traffic, bandwidth\right) + \beta\cdot\sum_{d\in D} W\left(load_d, speed_d\cdot\#cores_d\right) + \gamma\cdot\sum_{d\in D}\sum_{s\in S} W\left(load_{sd}, speed_d\right) \quad (13)$$

The coefficients α, β and γ express the constraint penalties and are chosen such that the constraints are not violated in almost all cases (in our experiments $\alpha = \beta = \gamma = 100$). The function W expresses the relative violation of the constraints and is defined as follows.

$$W\left(a,b\right) = \begin{cases} \frac{a-b}{b} & \text{if } a > b \\ 0 & \text{otherwise} \end{cases} \quad (14)$$

D. Heuristic algorithms

The goal of an allocation algorithm is to improve the current allocation of the cloudlet by defining a set of actions to be performed. In a collaborative cloudlet which supports the mechanisms discussed in Section IV, valid actions belong to either one of the following types. If these actions are performed on a valid configuration of the cloudlet, i.e. all components are deployed and state sharing is possible, the result will also be a valid configuration.

Solitary offloading This type of action will migrate a single component instance c on d to another node d' in the cloudlet. This boils down to updating X: $X_{cd} \leftarrow 0$, $X_{cd'} \leftarrow 1$. These actions are only possible for offloadable components that 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 requires updating Y: $Y_{c_ic_i} \leftarrow 0$, $Y_{c_ic_{shared}} \leftarrow 1$. Shared offloading may also change the allocation of the shared instance. This will update X: $X_{c_{shared}d} \leftarrow 0$, $X_{c_{shared}d'} \leftarrow 1$. Shared offloading is only possible for offloadable components that require state sharing.

Switch server This will change the server for a component type that is currently using state synchronization to collaborate. State synchronization is also used as the default collaboration mechanism, and can be used for any component that requires state sharing, whether it can be offloaded or not. Switching the server requires updating the sequences representing the synchronization messages.

Shared2Sync Actions of this type will switch between sharing an instance and using state synchronization as the collaboration mechanism. The shared instance will become the new synchronization server.

Sync2Shared These actions switch from using state synchronization for collaboration, to sharing a component instance. The synchronization server is chosen as the shared instance.

Algorithm 1 Steepest Descent

```
Current ← Initial
repeat
    K_max ← ∅
    G_max ← 0
    for ∀K ∈ allActions(Current) do
        G ← gain(K, Current)
        if G > G_max then
            G_max ← G
            K_max ← K
        end if
    end for
    if K_max ≠ ∅ then
        Current ← performAction(K_max, Current)
    end if
until the stop criterion is met
return Current
```

Suitable allocation algorithms are ones that can be used in an operational cloudlet, meaning that the execution time of the algorithm needs to be limited and be in relation to the period of the control loop. If we make the assumption of a cloudlet where every component is offloadable to every node, the total number of valid actions scales approximately as $O\left(\#D^{\#C}\right)$, where $\#D$ is the number of nodes in the cloudlet and $\#C$ is the total number of components. When looking at how the number of valid actions scales exponentially with the number of components, a brute-force, exhaustive approach is infeasible for runtime optimization. Even more intelligent optimal approaches such as Quadratic Programming (QP) do not scale well enough with the infrastructure and application size.

Heuristic allocation algorithms are hence needed for runtime optimization of the cloudlet. The goal is that, while the heuristic is not guaranteed to find the optimal solution, it is able to provide a sufficient approximation in limited execution time and scales better with problem size. 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 are search heuristics that explore the solution space by performing actions of the types described above.

1) Steepest Descent: A first search heuristic, SD, 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 is determined and the action with the

Algorithm 2 Simulated Annealing

```
Current ← Initial
Best ← Initial
T ← startTemperature(Initial)
L ← epochLength(Initial)
repeat
  for L times do
    K ← randomAction (Current)
    G ← gain (K, Current)
    if accepted with probability exp (G/T) then
      Current ← performAction (K, Current)
      if f (Current) < f (Best) then
        Best ← Current
      end if
    end if
  end for
  Decrease T
until the stop criterion is met
return Best
```

highest positive gain is accepted. The gain is defined as the difference in the objective function by performing the action. This process repeats itself until no further actions with a positive gain are found or a maximum number of actions, given by the iteration threshold parameter, have been accepted. SD is easily parallelizable, as the gain of every action during a single iteration can be calculated independently. In our experiments, however, we use a single-threaded implementation of the algorithm in order to make no assumption on the presence of a multi-core CA.

2) Simulated Annealing: A major disadvantage of SD is the high risk of converging to a non-global minimum, as only a limited part of the solution space is explored. To address this issue, an algorithm that performs a random instead of deterministic search can be used.

SA is is an action-based, intelligent random search 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 actions are randomly selected. A selected action is accepted with probability $\exp(G/T)$, where G is the gain in the objective function by executing the 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 large part of the solution space can be explored. Decreasing the temperature allows the algorithm to converge. In our implementation, the stop criterion depends on the fraction of actions that are accepted during an epoch. The different parameters of SA are described below and are determined in Section VI-B.

Initial fraction accepted actions with loss The initial temperature is selected so that, during the first epoch, a given expected fraction of actions with a negative gain is accepted.

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

Epoch coefficient The number of selected actions during each epoch is proportional with the total number of valid actions, scaled with an epoch coefficient.

Fraction accepted actions threshold When the fraction of actions that got accepted during an epoch falls below a threshold, a stopcounter is increased. This stopcounter is reset when a globally better solution is found.

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

VI. RESULTS

The heuristic allocation algorithms are compared to an Exhaustive (EX) allocation algorithm with regards to their effectiveness (i.e. how well they can approximate the solution of the EX algorithm) and execution times. The EX algorithm is guaranteed to be optimal and will perform a brute-force search for the global minimum by testing every reachable allocation in the solution space. An allocation is reachable when it is the result of applying a set of valid actions on the initial allocation. To evaluate the algorithms in a number of different scenarios, a set of random input problems is generated. All measurements were performed on a Intel Core i5-3230M 2.6 GHz quad-core processor.

A. Problem generation

We generated a total of 500 different cloudlet configurations. The cloudlets consist of two mobile users and a single fixed node. Node speeds are exponentially distributed, the number of cores uniformly. Each user runs an application instance consisting of 8 components. These components have a 50% chance of being offloadable, and also a 50% chance of needing to share their state. A total of 16 observed sequences are generated, which consist of a geometrically distributed number of methods and have an exponentially distributed frequency. Method load, argument and result size are also exponentially distributed.

B. Parameter selection

In order to compare the heuristic allocation algorithms to the EX algorithm, their parameters settings first need to be optimized. The goal is to simultaneously maximize their effectiveness and minimize their execution time. The effectiveness is calculated as

$$e_{SA,SD} = \frac{\sum_{p \in problems} gain_{SA,SD}(p) / gain_{EX}(p)}{\#problems}. \quad (15)$$

Note that all input problems are generated in such a way that $gain_{EX}(p) > 0$.

TABLE I
PARAMETERS SETTINGS OF THE DIFFERENT CONFIGURATIONS OF THE SA ALGORITHM.

Conf.	Init. fr.	Temp.	Epoch	Fr. acc.	Stop
1	0.9	0.5	1	0.5	1
2	0.9	0.5	1	0.5	4
3	0.9	0.2	1	0.5	4
4	0.05	0.2	1	0.5	4
5	0.05	0.2	0.5	0.5	4
6	0.05	0.2	0.5	0.1	4
7	0.05	0.2	0.5	0.1	8
8	0.05	0.2	0.33	0.1	8

1) Simulated Annealing: The SA algorithm has 5 different parameters as described in V-D2. In order to efficiently find suitable parameters, an iterative optimization approach is used. Starting from a base configuration, a single parameter is varied at a time. When done for all parameters, the parameter is selected that yields either the highest increase in effectiveness for the lowest increase in execution time or the highest decrease in execution time for the lowest decrease is effectiveness. The process is repeated until no significant improvement is possible.

This resulted in the 8 configurations listed in Table I. As SA is a stochastic algorithm, each configuration is tested 10 times. The average effectiveness and execution time as well as their standard deviation are shown in Fig. 2. The base configuration, configuration 1, performs reasonably well with an average effectiveness of 81.19% for an average execution time of 169 ms. After optimization of the parameters an effectiveness of 98.37% is achieved for an execution time of 133 ms with configuration 8. In further experiments, configuration 8 of the SA algorithm is used.

2) Steepest Descent: The SD algorithm only has a single parameter to select: the iteration threshold. Fig. 3 shows how the average effectiveness and execution time of the SD algorithm vary for different values of this threshold. For an iteration threshold of 8, we obtain an effectiveness of 92.88% for an execution time of 161 ms. This value is chosen as it most closely approximates the chosen SA configuration.

C. Comparison

The EX, SA and SD algorithms are compared with respect to different metrics. The SA algorithm is again executed 10 times and all metrics shown are averages. Fig.4 shows the execution times of the three algorithms for each of the input problems, sorted by the execution time of the EX algorithm. The average execution time for SA is 131 ms, which is slightly faster than SD which has an average execution time of 161 ms. Both are much faster than EX, which on average needs 176 seconds. As expected, both heuristic algorithms find a solution much faster than the EX algorithm, up to four orders of magnitude for the most complex problems. We see that SA

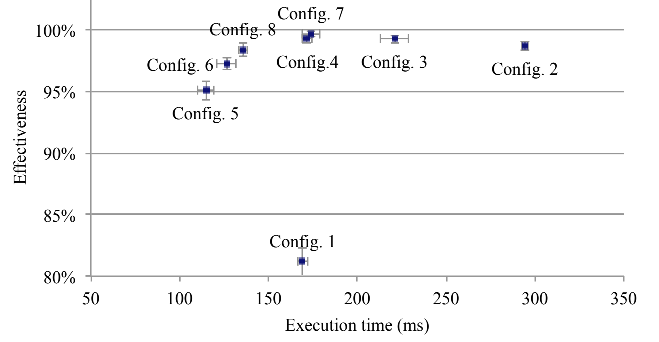

Fig. 2. Execution times and effectiveness for the different parameters configurations of the SA algorithm. Average and standard deviation of 10 iterations is shown.

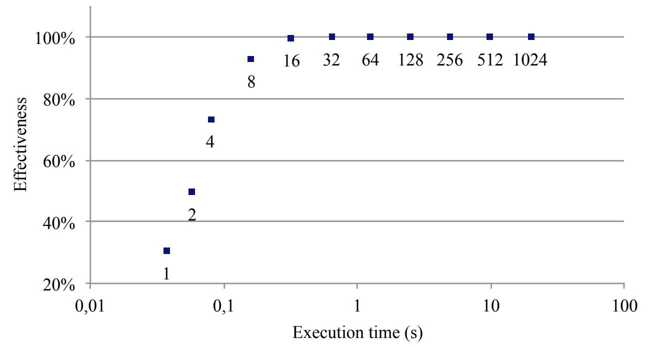

Fig. 3. Execution times and effectiveness for different values of the iteration threshold of the SD algorithm.

and SD both scale better with problem size as well, having a lower average gradient than EX.

An important metric is the number of actions an allocation algorithm suggests. While the optimization problem does not incorporate an explicit cost to each action, the actual cost can be a significant (e.g. components may be temporary unavailable during migration). Fig. 5 shows the number of actions each algorithm suggests for each problem, sorted by the number of actions of the EX algorithm. EX suggests an average of 6.6 actions. We observe that the number of actions of the SD algorithm is always equal (for simpler problems) or less (for more complex problems) than the number suggested by the EX algorithm, with an average of 5.2 actions. Due to the iteration threshold, the maximum number of actions of the SD algorithm is limited to 8. For the SA algorithm, we see that it sometimes proposes more actions than the EX algorithm, which is the result of its random nature. With an average of 5.8 actions however, it suggests almost one action less. This implies that the possible remaining 1% increase in effectiveness of SA requires, on average, that the number of actions to be performed increases with 1 as well.

The relative gain for each problem and algorithm is show in Fig. 6. The gains of the SA algorithm are closer to the

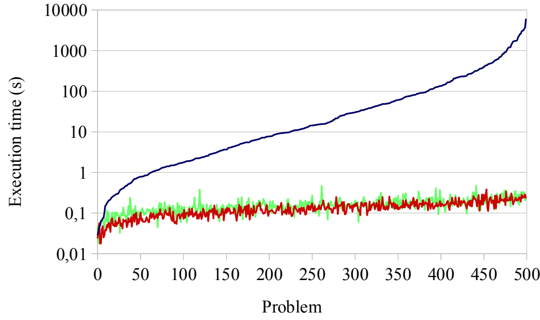

Fig. 4. Execution times of each allocation algorithm for all problems. Problems are sorted by execution time of the EX algorithm.

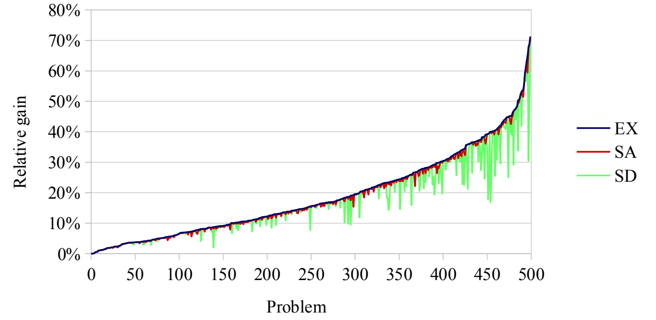

Fig. 6. Relative gain achieved by each allocation algorithm for all problems. Problems are sorted by relative gain achieved by the EX algorithm.

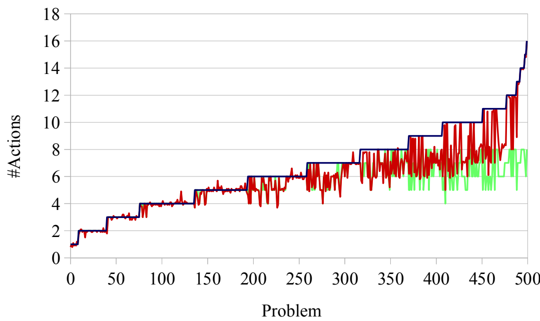

Fig. 5. Number of actions proposed by each allocation algorithm for all problems. Problems are sorted by number of actions proposed by the EX algorithm.

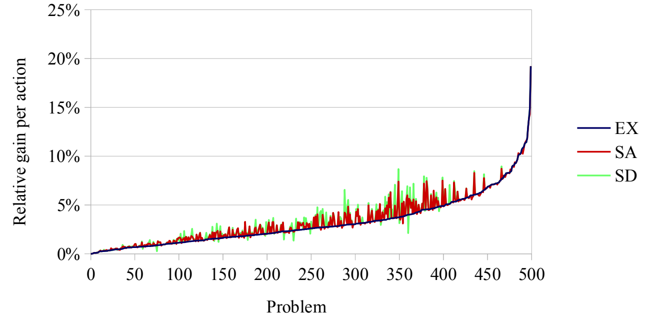

Fig. 7. Relative gain per action achieved by each allocation algorithm for all problems. Problems are sorted by relative gain per action achieved by the EX algorithm.

optimal ones than the gains obtained by the SD approach. This is to be expected as SA has a higher average effectiveness. On average, EX yields a relative gain of 18.86%, 0.35% more than SA (18.51%) and 0.93% more than SD (17.58%).

Finally we look at the relative gain obtained per action for each algorithm, which can be seen in Fig. 7. SD obtains the highest average gain per action with 3.59%, but this is expected as the algorithm only selects the actions that result in the highest gain at any given time. The difference with SA is however small, which has an average of 3.43% gain per action. With the gain per action of the EX approach being even lower at 3.22%, this again confirms that the additional gains of the optimal algorithm require relatively many actions. Compared to SA, the 1.89% increase in gain of the EX algorithm requires 12.47% additional actions.

VII. Conclusion

In this paper we presented a collaborative cloudlet middleware as an extension of a component-based cloudlet system with two collaboration mechanisms, shared offloading and state synchronization. We describe two heuristic algorithms for runtime optimization, Simulated Annealing (SA) and Steepest Descent (SD), and after fine-tuning their parameters, compared them to an optimal approach with regards to their execution times, obtained gains and suggested number of actions.

Not only do SA and SD solve the optimization problem with more than 90% effectiveness with execution times that are three orders of magnitude lower that the exhaustive approach, they are able to find a solution with fewer required actions. When comparing SA with SD, we observe that while SD results in fewer actions, SA is able to find a solution slightly faster and with a gain much closer to the optimum.

In order to further improve the autonomous management of collaborative cloudlets, the optimization problem should be extended to include the cost of performing the suggested actions. Moreover, the algorithms should be evaluated for larger problem size involving dozens of users and hundreds of application components. The SD approach can be further optimized by using a parallelized implementation. While this will not improve the effectiveness of the algorithm, it will further reduce the execution time on a multi-core CA.

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”.

REFERENCES

[1] "Gartner Press Release," 2013. [Online]. Available: http://www.gartner.com/newsroom/id/2408515

[2] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang, "The case for cyber foraging," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC - EW10*. ACM Press, July 2002, p. 87.

[3] M. Satyanarayanan, "Mobile computing: the next decade," in *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services Social Networks and Beyond*. ACM Press, June 2010, pp. 1–6.

[4] M. Daro Kristensen, "Scavenger: Transparent development of efficient cyber foraging applications," in *Proceedings of the 2010 IEEE International Conference on Pervasive Computing and Communications*. IEEE, Mar. 2010, pp. 217–226.

[5] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt, "AIOLOS: middleware for improving mobile application performance through cyber foraging," *Journal Of Systems And Software*, vol. 85, no. 11, pp. 2629–2639, 2012.

[6] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*. ACM Press, June 2010, p. 49.

[7] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*. ACM Press, June 2011, p. 43.

[8] Niantic Labs, "Ingress." [Online]. Available: http://www.ingress.com/

[9] S. Bohez, J. D. Turck, T. Verbelen, P. Simoens, and B. Dhoedt, "Mobile , Collaborative Augmented Reality using Cloudlets," in *Proceedings of the 6th International Conference on Mobile Wireless Middleware, Operating Systems and Applications*, 2013, pp. 1–10.

[10] J. Flinn and M. Satyanarayanan, "Balancing performance, energy, and quality in pervasive computing," in *Proceedings 22nd International Conference on Distributed Computing Systems*. IEEE Comput. Soc, 2002, pp. 217–226.

[11] R. K. Balan, M. Satyanarayanan, S. Y. Park, and T. Okoshi, "Tactics-based remote execution for mobile computing," in *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*. ACM Press, May 2003, pp. 273–286.

[12] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The Case for VM-Based Cloudlets in Mobile Computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, Oct. 2009.

[13] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan, "Just-in-time provisioning for cyber foraging," in *Proceeding of the 11th International Conference on Mobile Systems, Applications, and Services*. ACM Press, 2013, p. 153.

[14] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: elastic execution between mobile device and cloud," in *Proceedings of the Sixth Conference on Computer Systems*. ACM Press, Apr. 2011, p. 301.

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

[16] J. S. Rellermeyer, O. Riva, and 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 New York, Inc., Dec. 2008, pp. 22–41.

[17] X. Zhang, A. Kunjithapatham, S. Jeong, and S. Gibbs, "Towards an Elastic Application Model for Augmenting the Computing Capabilities of Mobile Devices with Cloud Computing," *Mobile Networks and Applications*, vol. 16, no. 3, pp. 270–284, Apr. 2011.

[18] M. Sharifi, S. Kafaie, and O. Kashefi, "A Survey and Taxonomy of Cyber Foraging of Mobile Devices," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 4, pp. 1232–1243, 2012.

[19] V. Sacramento, M. Endler, H. Rubinsztejn, L. Lima, K. Goncalves, F. Nascimento, and G. Bueno, "MoCA: A Middleware for Developing Collaborative Applications for Mobile Users," *IEEE Distributed Systems Online*, vol. 05, no. 10, pp. 2–2, Oct. 2004.

[20] G. Cugola and 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.

[21] T. Verbelen, T. Stevens, F. De Turck, and B. Dhoedt, "Graph partitioning algorithms for optimizing software deployment in mobile cloud computing," *Future Generation Computer Systems*, vol. 29, no. 2, pp. 451–459, Feb. 2013.

[22] T. Verbelen, P. Simoens, F. De Turck, and 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.

[23] T. Verbelen, P. Simoens, F. De Turck, and 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*, 2012, pp. 1–14.

[24] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.