Mobile, Collaborative Augmented Reality using Cloudlets

Steven Bohez, Joeri De Turck, Tim Verbelen, Pieter Simoens and Bart Dhoedt
Department of Information Technology
Ghent University
Gaston Crommenlaan 8/201
9050 Ghent, Belgium
Email: {steven.bohez},{joeri.deturck},{tim.verbelen},
{pieter.simoens},{bart.dhoedt}@ugent.be

Abstract

The evolution in mobile applications to support advanced interactivity and demanding multimedia features is still ongoing. Novel application concepts (e.g. mobile Augmented Reality (AR)) are however hindered by the inherently limited resources available on mobile platforms (not withstanding the dramatic performance increases of mobile hardware). Offloading resource intensive application components to the cloud, also known as "cyber foraging", has proven to be a valuable solution in a variety of scenarios. However, also for collaborative scenarios, in which data together with its processing are shared between multiple users, this offloading concept is highly promising. In this paper, we investigate the challenges posed by offloading collaborative mobile applications. We present a middleware platform capable of autonomously deploying software components to minimize average CPU load, while guaranteeing smooth collaboration. As a use case, we present and evaluate a collaborative AR application, offering interaction between users, the physical environment as well as with the virtual objects superimposed on this physical environment.

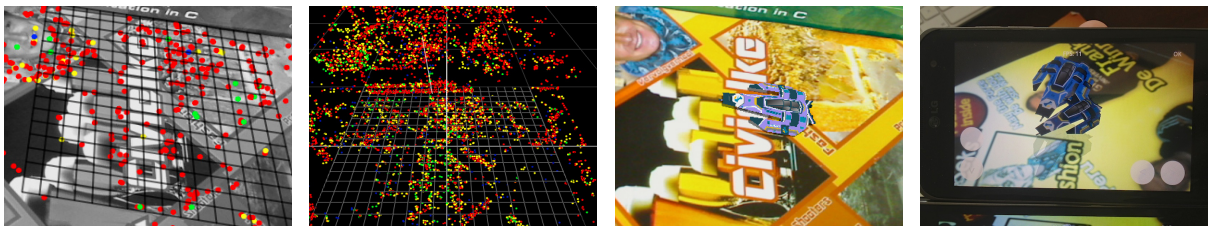
1. Introduction

In recent years, the mobile device market has been one of the fastest growing market segments, and sales predictions forecasting billions of smartphones and tablets to be sold during 2013 [1] confirm this market shows no sign of slowing down. The popularity of these devices not only stems from their portability, but also from their always-on nature, extensive connectivity and thousands of easy-to-install applications. These mobile applications have recently been evolving towards fully interactive multimedia experiences, such as immersive 3D games and design software. This evolution is however hindered by the limited capabilities of mobile devices, such as limited processing power and battery capacity.

To cope with these resource limitations, cyber foraging [2] was introduced, where infrastructure available in the near vicinity of the user is used to offload applications or parts thereof. Cyber foraging can be realised by a cloudlet [3], consisting of a collection of computing nodes (e.g. a server co-located with the wireless access point) that are sharing their resources with mobile terminals. Resource-intensive software components can then be offloaded to a computing node with ample free capacity, in order to reduce the execution time [4], [5], energy consumption [6], and/or improve throughput [7]. This set-up is similar to the cloud but does not suffer from large network delay, so it can be utilized to outsource delay-critical tasks.

Applications involving multiple users interacting with each other, so-called collaborative applications, are also getting more interest. Especially interactive and collaborative applications such as AR are gaining popularity fast (e.g. Niantic Labs' Ingress [8]). The cloudlet concept offers a number of opportunities for collaborative applications: instead of only sharing resources, users could also share results of calculations. In AR, for example, multiple users could use and expand the same map of the environment (see Section 3). However, none of the existing cloudlet or cyber foraging systems exploit this opportunity.

In this paper, we extend the cloudlet framework presented in [9], [10] to a "collaborative cloudlet" system that provides support for collaboration from the cloudlet middleware. Moreover, our system is capable of autonomous configuration and will automatically deploy components to minimize the overall load on all devices and the bandwidth consumed, while taking



(a) Tracker (b) Mapper (c) Renderer (d) Result

Figure 1. Output of the Tracker, Mapper and Renderer and the final result with multiple users.

into account all necessary state synchronization. As a use case, we developed a distributed AR game capable of collaboration and basic interaction with the environment.

2. Related Work

Early cyber foraging systems like Spectra [11] and Chroma [12] required substantial effort from the developer and pre-installed routines on the fixed nodes. In contrast, our solution does not require knowledge of network topology and can perform code migration at runtime.

Some systems use Virtual Machines (VMs) to "copy" applications from mobile devices to infrastructure. The VM-based cloudlets in [13] and [14] will execute the entire application in the VM. CloneCloud [15], on the other hand, uses profiling and code analysis to decide where to execute each routine separately. COMET [16] uses a Distributed Shared Memory (DSM) approach which allows for easy migrating of individual application threads between VMs.

Other recent approaches divide an application into different software components of which the resource intensive ones can be offloaded to infrastructure in the local network to reduce load. Using components instead of VMs provides more flexibility for application deployment. Often, these components are the methods themselves, such as in MAUI [6] and Scavenger [4]. These components can also be larger, such as Open Services Gateway initiative (OSGi) bundles in AlfredO [17] and AIOLOS [5]. Zhang et al. [18] use RESTful services called weblets. The algorithms used to partition the application are often based on well-known graph-partitioning algorithms [19], although e.g. [18] uses Naive Bayesian Learning techniques for runtime optimization. An in-depth comparison of cyber foraging and other mobile cloud computing systems is given in [20], [21].

Apart from multimedia applications, collaborative applications are also gaining interest. In collaborative applications, also called groupware, users work together in a common context or state to accomplish a common goal. Collaborative applications face additional challenges when executed in a mobile environment due to joining and leaving users. An example of a middleware system specifically focused on collaborative applications is MoCa [22].

No existing middleware framework however incorporates both aspects of cyber foraging and collaboration. We believe that collaborative applications can also benefit from the cloudlet concept by allowing users to share both resources and results. We integrate these concepts to adopt the component-based cloudlet for collaborative applications, autonomously deciding which components to offload or share.

3. Augmented Reality use case

A collaborative AR application is used for determining the main requirements of our framework and for evaluation. An AR application integrates virtual objects in (an image of) the real world [23]. Most AR applications use markers to identify the position of the user (camera), but there are also some methods that work without a priori knowledge of the environment. These algorithms create a map of the environment at runtime and use this map to track the position of the camera. Such a technique is described by Klein and Murray, who introduced Parallel Tracking and Mapping (PTAM) [24]. This algorithm was later extended by Castle et al. [25] who created Parallel Tracking and Multiple Mapping (PTAMM) that allows using multiple maps and automatic switching between these maps.

As a use case, an AR application is presented that is based on the PTAM algorithm (see [24] for details). Fig. 1 shows the collaborative AR application we developed: a simple game where every user can control his own spaceship and can interact with other users and the real environment. Fig. 1a shows the recognized feature points, 1b is the map, on which

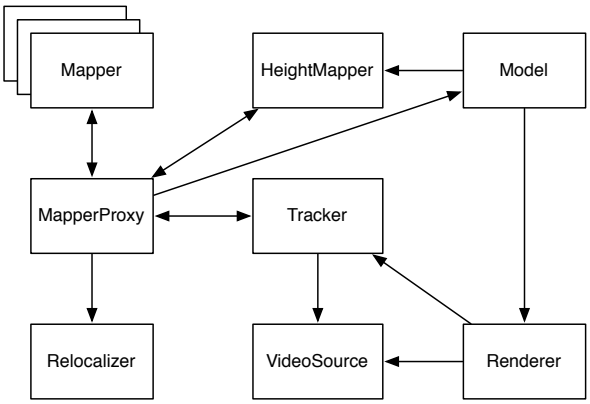


Figure 2. The components of our use case.

Figure 3. The height and shadow of the spaceship can be adjusted by using the information from the HeightMapper.

the feature points are projected. By determining the projection of the points, the position of the camera can be found. The camera position is then used to render the spaceship correctly, which can be seen in Fig. 1c. Fig. 1d shows the application running on a mobile device with multiple users visible. Users "looking" at the same map will appear in the same virtual space and can e.g. race each other. The height of the spaceship is adapted based on its position in the environment.

The application consists of the following components, as shown in Fig. 2:

Mapper By matching feature points between multiple keyframes of a video, the 3D position of the points can be estimated to create a 3D map. The Mapper receives new keyframes from time to time.

Tracker This component extracts feature points in the current keyframe and aims to determine the position of the camera. The position of the camera can be calculated by matching 2D feature points of the video with 3D points of the Mapper.

MapperProxy We extended PTAM to enable the use of multiple maps and automatically switch between them. The MapperProxy coordinates map selection between the different Mappers, the Tracker and the Relocalizer.

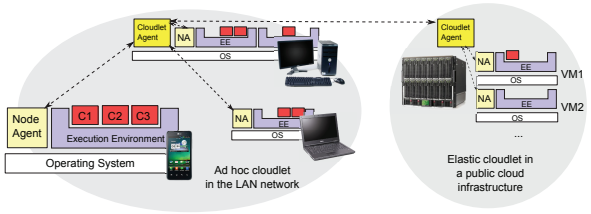


Figure 4. Architecture of the existing cloudlet framework [9].

Relocalizer When the Tracker cannot find the position of the camera, the Relocalizer makes a rough estimation of the camera position based on keyframe similarity. The Relocalizer is also responsible for the automatic switching between different maps.

HeightMapper Interaction with the environment is made possible by the HeightMapper component. This component creates a heightmap from the 3D points of the Mappers. The height of the spaceship is adjusted according to the estimated ground level, as in Fig. 3.

Model This component keeps track of the position, orientation and velocity of the spaceships.

VideoSource This component fetches videoframes from the camera and sends them to the Tracker.

Renderer The videoframes and virtual 3D objects are combined by the Renderer. The 3D objects are aligned with the videoframes by using the camera position of the Tracker.

This application has interesting characteristics and requirements with respect to the framework. Some components, like the HeightMapper and Mapper, are very CPU intensive, while others have real-time constraints; i.e. the Tracker and Renderer should process frames within 50 ms to achieve an acceptable user experience. It is also necessary to be able to efficiently exchange data between multiple users. When multiple users run the application at the same location, sharing data could also save computing power. The HeightMapper is an example of this: by sharing data, the heightmap of each map has to be constructed only once instead of on every device.

4. Middleware architecture

Our middleware framework is an extension of the previous work on cloudlets presented in [9], [10]. While this cloudlet framework already provides the necessary functionality for offloading software components, it lacks support for collaborative applications. Fig. 4 shows the main parts of the framework architecture: components, Execution Environments (EEs),

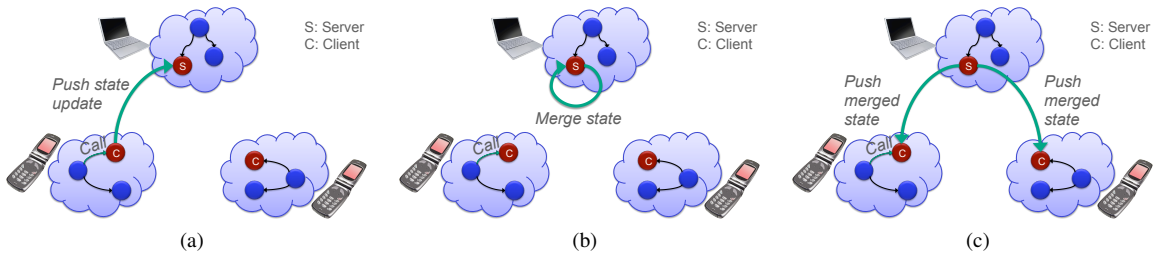


Figure 5. The three steps performed during synchronisation.

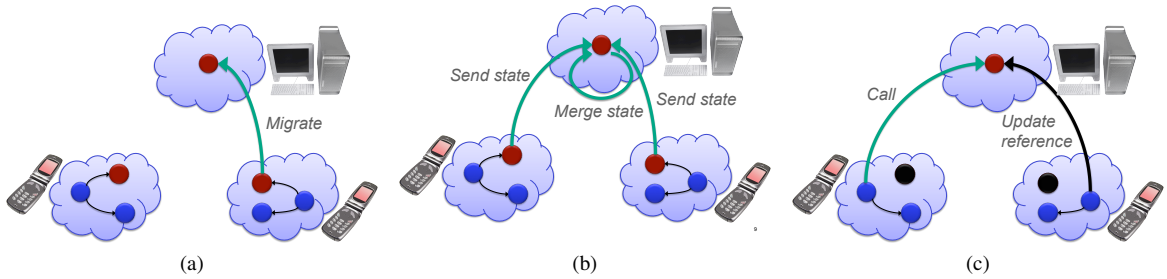


Figure 6. The three steps performed during shared offloading.

Node Agents (NAs) and a Cloudlet Agent (CA).

An application has to be split up in several loosely-coupled components that offer and require certain services (interfaces). These components are managed by the EE, which can start and stop components, and also makes sure that the components are linked together according to their offered and required interfaces. Components can be migrated from one EE to another. Function calls to components deployed on remote EEs are intercepted by proxies and executed using Remote Procedure Calls (RPCs).

A device (PC, smartphone, server) is called a node and is managed by a NA that can start and stop EEs, and monitors the (virtualized) hardware it runs on. The nodes in the same LAN network form a cloudlet that is managed by a single CA, which is chosen based on node speed (see Section 6.1). The CA communicates with the NAs and collects all the monitor information from the NAs and EEs. The CA can use this information to decide whether it has to offload certain components to achieve e.g. a lower (global) CPU usage.

We extend this middleware framework in two important ways. First, we add support for collaborative applications through a couple of mechanisms described in Section 5. Second, we introduce an allocation algorithm in Section 6 which will facilitate autonomous cloudlet configuration by determining a suitable component allocation and selecting the appropriate collaboration mechanisms.

5. Mechanisms for collaboration

To provide collaboration, sharing state between clients is essential. We extended the existing cloudlet framework with two mechanisms for collaborations that enable multiple instances of the same component to share data: synchronization and shared offloading.

5.1. Synchronization

Synchronisation is an active mechanism: messages are exchanged to keep a consistent state among component instances. We implemented a client-server synchronization mechanism, where one instance of the component is selected as a server. This instance holds the "ground truth" state and is responsible for the distribution of state updates. Fig. 5 shows this three-step process. The clients push their changed state to the server, after which the server merges the new state with the existing state. At this time the server can choose to buffer the update and check for consistency with pending state updates. How conflicts are resolved, is up to the application developer (see Section 7.1). Finally, the server pushes the (corrected) state update to all the clients if necessary, where the local state is overwritten. More advanced techniques such as incremental synchronization and revision histories are currently not offered by the framework, but could be implemented by the application developer.

5.2. Shared offloading

The second collaboration technique is shared offloading, where multiple application instances use a single, shared component instance. This mechanism builds on the migration feature already present in the cloudlet framework, and is again performed in three steps. First a shared instance is created. Depending on the location of this shared instance, nearly every method call will be an RPC. Only when the shared instance is allocated on a user device, will there still be local calls from the local components. Next, all other instances will push their state so it can be merged as specified by the application developer. Finally, all references are changed to refer to the shared instance. By sharing a component, its state will remain consistent across all application instances and collaboration is guaranteed. As no additional messages need to be exchanged, this is a passive mechanism.

6. Allocation algorithm

Cloudlets offer a number of opportunities to change component deployment, for example by migrating components or by changing the collaboration mechanism. By moving a resource-intensive component to a node with plenty of unused capacity, we can reduce the load on mobile nodes. Manual configuration is however impractical, therefore we adopted an autonomous Monitor-Analyze-Plan-Execute (MAPE) control loop [26], and developed a heuristic allocation algorithm to autonomously find an optimal configuration.

6.1. Model

The allocation algorithm is formulated as an optimization problem based on a theoretical model of the cloudlet in [10]. This model incorporates infrastructure, applications and behaviour and is extended to incorporate the collaborative aspects, such as synchronization servers and shared components.

6.1.1. Infrastructure. Every cloudlet consists of a number of nodes $d \in D$. Each node has a number of CPU cores, denoted by $\#CPUcores_d$ and each core can handle a certain amount of load per unit of time, the node speed $CPUspeed_d$.

6.1.2. Applications. Each application on the cloudlet consists of a number of active components $c \in C$. Some of these components are synchronization servers or shared instances, which are grouped in $H_{servers}$ and

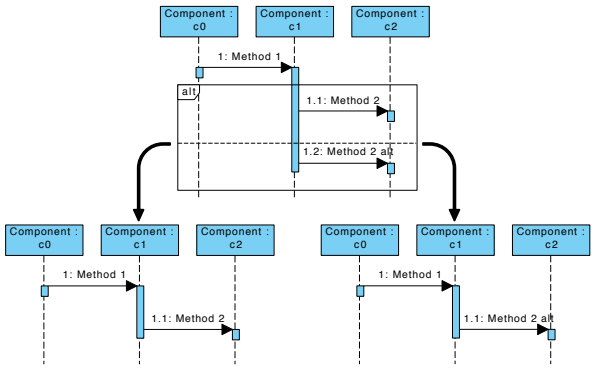


Figure 7. The transformation from a sequence diagram to separate sequences.

H_{shared} respectively. Furthermore, we define X_{cd} and h_{ij} as

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

$$h_{ij} = 1 - \sum_{d \in D} X_{c_i d} \cdot X_{c_j d}. \quad (2)$$

Hence, X_{cd} represents an allocation matrix and h_{ij} tells us whether c_i and c_j are allocated on different nodes.

6.1.3. Behaviour. To model the behaviour, we introduce the concept of a sequence: a specific succession of method calls along a certain path through the control flow graph. Fig. 7 shows this concept. A single sequence diagram containing a conditional statement is split in two distinct paths or sequences, only one of which will be followed at runtime. Each sequence $s \in S$ occurs with a certain frequency $\#calls_s$ and contains a number of method calls $m_{sc_ic_j}$ from component c_i to c_j. Every method has a certain load $load_{m_{sc_ic_j}}$, an argument size $A_{m_{sc_ic_j}}$, a result size $B_{m_{sc_ic_j}}$ and occurs $\#calls_{m_{sc_ic_j}}$ times within the sequence.

6.2. Objective

Based on this model, we minimize a weighted sum of the average CPU usage of all devices in the cloudlet and the required total bandwidth:

$$f_{objective} = \alpha \cdot CPUusage_{avg} + \beta \cdot bandwidth \quad (3)$$

By minimizing this objective function, we not only leverage the larger capacity of fixed nodes for reducing load on mobile nodes, we also keep the necessary

bandwidth low. By adjusting the parameter α, we are able to either put more emphasis on minimizing average CPU load or on bandwidth usage.

To obtain $CPUusage_{avg}$, we first calculate $load_{sd}$, which is the load per unit of time generated by sequence s on device d.

$$load_{sd} = \sum_{c_i \in C} \sum_{c_j \in C} X_{c_j d} \cdot load_{m_{sc_ic_j}} \cdot \#calls_{m_{sc_ic_j}} \cdot \#calls_s \quad (4)$$

Summing $load_{sd}$ over all sequences gives us $load_d$, the total load per unit of time imposed on device d.

$$load_d = \sum_{s \in S} load_{sd} \quad (5)$$

This allows us to calculate the average CPU usage on device d as:

$$CPUusage_d = \frac{load_d}{CPUspeed_d \cdot \#CPUcores_d} \quad (6)$$

The total average CPU usage is then given by:

$$CPUusage_{avg} = \sum_{d \in D} CPUusage_d / \#D \quad (7)$$

The bandwidth used can be found by summing the size of the argument and return values for each call that is executed by components c_i and c_j that are allocated on different devices (i.e. $h_{ij} = 1$).

$$bandwidth = \sum_{s \in S} \sum_{c_i \in C} \sum_{c_j \in C} h_{ij} \cdot \left(A_{m_{sc_ic_j}} + B_{m_{sc_ic_j}} \right) \cdot \#calls_{m_{sc_ic_j}} \cdot \#calls_s \quad (8)$$

Additional constraints are added to the objective function as not to exceed the single-threaded and multi-threaded capacity of each node. As every method within a sequence is executed sequentially, the load of that sequence on a specific core cannot exceed the speed of that core.

$$load_{sd} \leq CPUspeed_d, \forall s \in S, \forall d \in D \quad (9)$$

Similarly, the load of all the sequences in parallel on a specific node, cannot exceed the total capacity of that node.

$$load_d \leq CPUspeed_d \cdot \#CPUcores_d, \forall d \in D \quad (10)$$

Algorithm 1 Simulated Annealing

```
Current ← Initial
Best ← Initial
T ← startTemperature(Initial)
L ← epochLength(Initial)
repeat
  for L times do
    Select possible move k ← selectMove(Current)
    Calculate gain G ← g(k, Current)
    if accepted with probability e^(G/T) then
      Current ← performMove(k, Current)
      if f(Current) < f(Best) then
        Best ← Solution
      end if
    end if
  end for
  Decrease T
until the stop criterion is met
return Best
```

6.3. Heuristic

There are a number of possible actions to change the allocation of the cloudlet: migrating components, changing synchronization servers and switching between synchronizing and sharing. The number of valid allocations scales approximately as $O\left(D^C\right)$, where D is the number of nodes and C is the number of components. Algorithms to find the global optimum such as a brute-force approach or Integer Linear Programming (ILP) do not scale well with the infrastructure and application size. Hence heuristics are required for runtime optimization in realistic scenarios.

Inspired by the results presented in [19], Simulated Annealing (SA) is adopted in our framework. This is a move-based, intelligent random search using a control factor defined as the temperature. The procedure is shown in Algorithm 1. A randomly selected move is accepted with probability $\exp(G/T)$, where G is the gain in the objective function by executing the move and T is the temperature. Moves with a positive gain will always be accepted, but a fraction of moves with a negative gain will also be accepted depending on the temperature. The algorithm advances through a number of epochs, in which a fixed number (proportional to the number of possible moves) of random moves are tested with a fixed temperature.

A high initial temperature allows the algorithm to explore a large area of the search space, while decreasing the temperature geometrically as the algorithm advances, ensures it converges to a local optimum. The initial temperature is determined so that a given fraction (typically $> 60\%$) of moves with negative gain is accepted for the initial solution. When the fraction of accepted moves in a certain epoch falls below a given

threshold, a counter is increased. When this counter exceeds another given stop threshold, the algorithm terminates. As SA is stochastic, it also makes sense to run the algorithm for multiple iterations and select the most optimal solution found.

7. Implementation

To evaluate our design, we developed prototype versions of the collaborative cloudlet framework and the AR use case.

7.1. Collaborative cloudlet framework

The prototype of the cloudlet framework is developed in Java and is based on the OSGi specification. OSGi already provides functionality for installing, removing and moving components at runtime. Apache Felix [27] is used as implementation of OSGi. The migration of components and remote execution of method calls is made possible by an open source, lightweight implementation of the OSGi Remote Service Admin specification [28].

The synchronization mechanism is implemented as an extension of the EE. In order to enable state synchronization in a component, the application developer has to provide the following three methods: `Serializable getState()`, `void setState(Serializable state)` and `Serializable mergeState(Serializable state)`. The component can decide when to push its state to the server by calling the `syncNow` method on the EE. The `mergeState` method is called on the server to combine the old and new states, after which the server will call the `setState` method on each client to push the new state.

When migrating a component, a suitable EE needs to be selected in order to isolate different application instances and shared or offloaded components on the same node. When no empty EE is available, a new one is created to host the instances needing isolation. Shared offloading is implemented as migrating a component to an EE where a component instance is already active. When such a component instance is detected, the state is merged using the `mergeState` method. If no instance is active, a new one is started and its state is set using `setState`.

The allocation algorithm is integrated into the CA. The CA runs two control loops. A fast control loop with period 5 s will gather the monitoring information and process component properties to check for shareable and offloadable components. A slow control loop with period 30 s will execute the SA algorithm and execute the necessary commands to configure the cloudlet based on its solution. The slower loop allows the cloudlet to stabilize after executing commands so the next iteration will use valid monitoring information. To cope with nodes joining or leaving the cloudlet, an additional check in the fast loop will call the SA algorithm immediately when the nodes have changed.

7.2. Augmented Reality use case

The PTAM algorithm in [29], written in C++, is split into different components (Mapper, Tracker and Relocalizer) that are wrapped in Java using using Java Native Interface (JNI), and in turn into OSGi bundles. By providing binaries for multiple architectures in the bundle, the components are still portable. The MapperProxy component is added to the application to be able to use multiple Mappers without having to change the code of the Mapper.

The Mapper and HeightMapper are both computationally intensive and can be offloaded. The HeightMapper, Model and MapperProxy have a shareable state. By sharing the state of the HeightMapper, only one heightmap per Mapper has to be created, otherwise every client creates all of the heightmaps independently. The Model performs incremental synchronization by only sending the position of the spaceship tied to the local user. The Model is responsible for the interaction between the spaceships and the environment. When spaceships come in each other's vicinity, their height is adjusted to make sure they don't collide. The Model also fetches the ground level of the spaceship's position from the HeightMapper to adjust the shadow and the height of the spaceship.

The HeightMapper will create a heightmap for each Mapper instance based on its 3D pointcloud. To create the heightmap, this pointcloud is projected onto the ground plane. The projection is then triangulated after which each triangle is assigned the minimum height of its vertices in the original pointcloud. We use a Delaunay-triangulation [30] to get a regular shape and the incremental construction algorithm [31] to create this triangulation in practical time.

8. Results

The evaluation is done in two parts. First, the effectiveness of the allocation algorithm is determined by comparing it to an optimal algorithm. By making a trade-off between effectiveness and execution time, we can determine the parameters of the algorithm. Second, we evaluate the collaboration mechanisms and

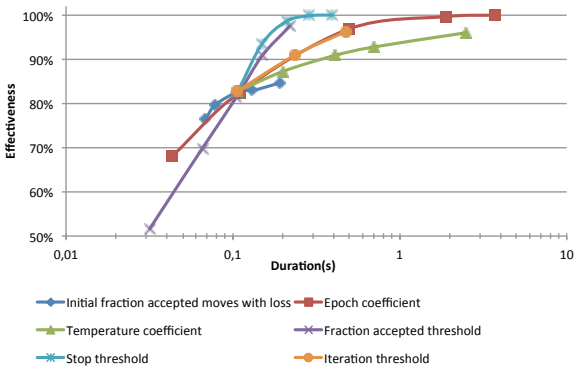


Figure 8. Comparison of the allocation algorithm parameters and their influence on the effectivity and execution time.

the allocation algorithm in a runtime scenario with the use case application.

8.1. Allocation algorithm

In order to evaluate the performance of the allocation algorithm, we will compare the SA technique to an optimal brute-force algorithm. We consider both effectiveness and execution time. The effectiveness is defined as the fraction of the gain in the objective function using the SA-approach to the one using the brute-force approach. We generated and averaged over 500 randomly generated applications which reflect the complexity of the AR use case. An average execution time of 88 s for finding the optimal solution through a brute force strategy motivates the need for a heuristic approach. First, the necessary coefficients in the objective function are determined using the optimal algorithm. After setting α to 1, we tune the other coefficients in such a way that the average CPU load and bandwidth have the same weight and none of the constraints are violated.

Secondly, the parameters of the SA algorithm are refined in order to improve the quality of the solutions obtained when running the SA algorithm for 1 s, as this value allows to perform SA-based optimisation at runtime. Fig. 8 shows the different parameters and how they vary in both effectiveness and execution time. The effects of these different parameter settings are evaluated in reference to a default configuration, which is the crossing point of all the curves. Due to the stochastic nature of the algorithm, these values are averaged over 10 runs for each configuration. The default configuration gets an effectiveness of 82,5% for an execution time of 105 ms. We observe that changing the stop threshold increases the effectiveness the most for the smallest increase in execution time. By increasing this threshold, we get an effectiveness of 98,6% for an execution time of 213 ms. Further optimizations only result in excessive execution times without significant improvement of the solution quality.

A hysteresis coefficient is added to ensure a certain minimal amount of gain is achieved before applying the solution. This will help to stabilize the cloudlet. We observe that a coefficient of 5% has no significant effect on the average effectiveness.

8.2. Augmented Reality use case

To evaluate the performance of the collaboration mechanisms and the allocation algorithm on the use case in a runtime scenario, two Android devices (Samsung Galaxy SII with a 1.2 GHz dual-core ARM Cortex-A9 and LG Optimus 2X with a 1 GHz Dual-core ARM Cortex-A9) and an extra free node (MacBook Pro with a 2.4 GHz Intel Core 2 Duo running Ubuntu Linux) are used. These devices are connected to a IEEE 802.11n wireless acces point with no other wireless devices present.

There are two stages in the test: the shareable components are first synchronized manually, and afterwards the allocation algorithm is activated. The measured CPU usage and bandwidth is visible on Fig. 9, as well as the important events. The allocation algorithm decides in less than a second to offload the HeightMapper and Mapper instances to the free node (and does this consistently in multiple test runs); the migration itself takes about 20 seconds due to pending function calls to the HeightMapper. The CPU usage decreases from 100% to around 85% on the Android devices and increases on the server. The CPU usage gain on Android is modest because there is no cap on the Frames Per Second (FPS) and the Tracker profits from the freed resources to process more frames.

Beside the CPU usage, we also monitored the execution times of the most important methods: those that process map updates (in the Mapper and HeightMapper) and videoframes (in the Tracker). Fig. 10 shows the average execution times and standard deviations before and after the offloading. The Tracker component is not offloaded, but the execution time decreases by 41% due to the extra available CPU cycles after offloading the Mapper and HeightMapper. The execution times of the Mapper and HeightMapper decrease with 49% and 95% respectively, including the network delay. The major difference in execution times of the HeightMapper before and after offloading is caused by the limited floating point support of ARM processors.

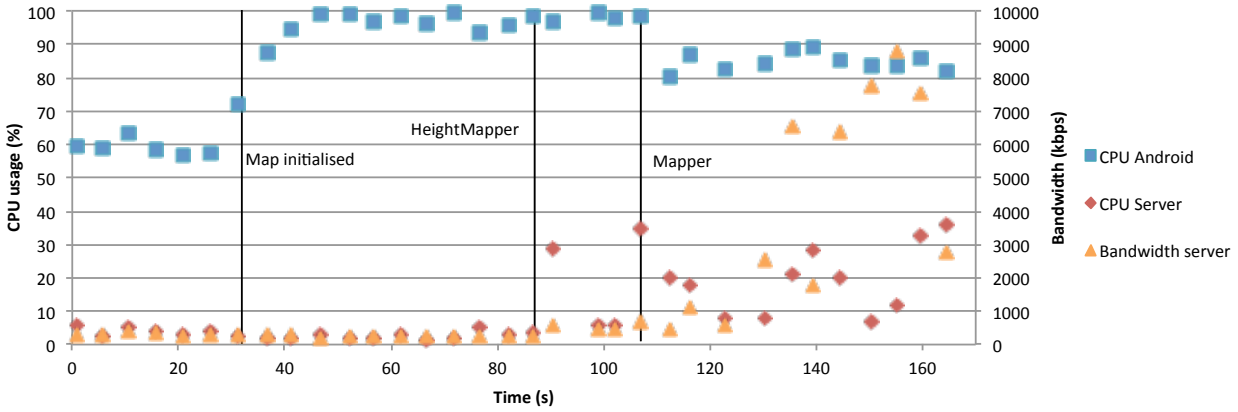
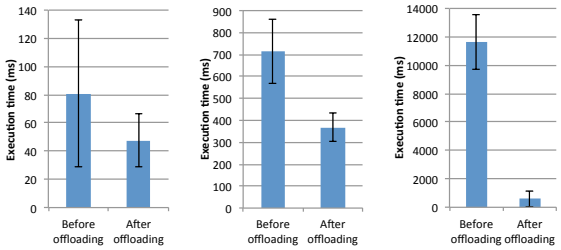


Figure 9. The change in CPU usage and bandwidth in our testing scenario. At 30 s the map is initialised, at about 90 s the allocation algorithm is activated. Offloading of the HeightMapper is completed 20 s later, after which the Mapper is offloaded.

9. Conclusion

In this paper we proposed a mobile middleware framework that extends the cloudlet concept to support collaboration using two mechanisms: synchronization and shared offloading. An AR application based on PTAM was also extended to incorporate collaboration and allow basic interaction between users and with the environment. An allocation algorithm was designed to allow autonomous configuration of the cloudlet.

By evaluating the framework we were able to determine the benefits and drawbacks of both collaboration mechanisms. Which mechanism is more suitable depends on constraints imposed on bandwidth, CPU load, execution times and/or the migration of components. We were also able to fine-tune the parameters in the allocation algorithm to get the most effectiveness for the given time frame. When running in combination with the AR use case, we were able to decrease execution times of key components by 41% to up to 95%, while still guaranteeing collaboration.



(a) Tracker (b) Mapper (c) HeightMapper

Figure 10. Execution times before and after offloading to a server. The Tracker keeps running locally, the Mapper and HeightMapper are offloaded to the free node.

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 - MCS '10*. ACM Press, June 2010, pp. 1–6.

[4] M. Daro Kristensen, "Scavenger: Transparent development of efficient cyber foraging applications," in *2010 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. 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 - MobiSys '10*. 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 - MobiSys '11. ACM Press, June 2011, p. 43.

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

[9] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt, "Cloudlets: bringing the cloud to the mobile user," in *3rd ACM Workshop on Mobile Cloud Computing and Services, Proceedings*. Ghent University, Department of Information technology, 2012, pp. 29–35.

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

[11] 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.

[12] 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 - MobiSys '03*. ACM Press, May 2003, pp. 273–286.

[13] 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.

[14] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan, "Just-in-time provisioning for cyber foraging," *Proceeding of the 11th annual international conference on Mobile systems, applications, and service (MobiSys '13)*, p. 153, 2013.

[15] 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 - EuroSys '11*. ACM Press, Apr. 2011, p. 301.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] D. Kovachev, Y. Cao, and R. Klamma, "Mobile Cloud Computing: A Comparison of Application Models," *CoRR*, vol. abs/1107.4, 2011.

[22] 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.

[23] R. T. Azuma, "A Survey of Augmented Reality," *Presence*, vol. 6, no. August, pp. 355–385, 1997.

[24] G. Klein and D. Murray, "Parallel Tracking and Mapping for Small AR Workspaces," in *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*. IEEE, Nov. 2007, pp. 1–10.

[25] R. O. Castle, G. Klein, and D. W. Murray, "Wide-area augmented reality using camera tracking and mapping in multiple regions," *Computer Vision and Image Understanding*, vol. 115, no. 6, pp. 854–867, June 2011.

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

[27] "Apache Felix," 2013. [Online]. Available: http://felix.apache.org/

[28] T. Verbelen, "RemoteServiceAdmin," 2013. [Online]. Available: http://code.google.com/p/remoteserviceadmin/

[29] G. Klein and D. Murray, "Parallel Tracking and Mapping on a camera phone," in *2009 8th IEEE International Symposium on Mixed and Augmented Reality*. IEEE, Oct. 2009, pp. 83–86.

[30] B. N. Delaunay, "Neue darstellung der geometrischen kristallographie," *Z. Kristallo-graph*, vol. 84, pp. 109–149, 1932.

[31] L. J. Guibas, D. E. Knuth, and M. Sharir, "Randomized incremental construction of Delaunay and Voronoi diagrams," *Algorithmica*, vol. 7, no. 1-6, pp. 381–413, June 1992.