Cyber-Physical Cloud Computing Implemented as PaaS

Clemens Krainer
Department of Computer Sciences
University of Salzburg, Austria
ckrainer@cs.uni-salzburg.at

Christoph M. Kirsch
Department of Computer Sciences
University of Salzburg, Austria
ck@cs.uni-salzburg.at

ABSTRACT

We describe a Platform-as-a-Service (PaaS) system for performing multi-customer information acquisition missions on unmanned vehicle swarms operated and maintained by a third party. Customers implement their missions completely unaware of each other and the available vehicle infrastructure. Vehicle swarm providers may add or remove vehicles unnoticed by customers for maintenance, recharging, and refueling. To achieve this, we apply the paradigm of cloud computing to virtualized versions of unmanned vehicles. Our implementation allows the simulation of multi-customer information acquisition missions as well as their execution on real hardware running the robot operating system (ROS).

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.m [**Software Engineering**]: Miscellaneous—*PaaS, Vehicle Virtualization, Cyber-Physical Systems*

General Terms

Cloud Computing, Vehicle Swarms, Sensor Networks

Keywords

Cyber-Physical Cloud Computing, Virtualization, Spatial Computing, Mobile Robotic Networks, Sensor Networks

1. INTRODUCTION

With the availability of low-cost unmanned vehicles, it is possible to use swarms of autonomous vehicles to perform information acquisition tasks over large areas. To benefit from the potential of vehicle swarms, it is necessary that practical application is safe, inexpensive, simple, and available to multiple clients.

The goal of this work is to examine strict separation of performing information acquisition tasks and operating autonomous vehicles. This work proposes a novel approach, in which owners of unmanned vehicle fleets provide their infrastructure to arbitrary, inexperienced clients. To achieve this, we take the paradigm of cloud computing and apply it in a cyber-physical system.

CyPhy'14, April 14 - 17 2014, Berlin, Germany

http://dx.doi.org/10.1145/2593458.2593461.

Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [12]. When computation happens not only in time but also in space, on-board computing units of servers moving in space can carry out the computation. [5] defines cloud computing in space and time as cyber-physical cloud computing (CPCC). In CPCC the servers (vehicles) can move in space and carry sensors and actuators. Examples of moving servers (vehicles) include collaborative sensing missions [16, 17], environment monitoring [8, 6], airborne computing clouds [10], in-vehicle smartphones as internet-connected sensors [7], and mobile phone sensing [13].

Analogous to regular cloud computing, CPCC customers get a virtual machine running on a real server [4]. The CPCC server is called a real vehicle (RV). The virtual machine is called a virtual vehicle (VV). We define VVs through the idea of a virtual vehicle monitor (VVM), analogous to a virtual machine monitor. VVs do the actual information acquisition, that is, requesting transportation to points in space, querying sensor values, and analyzing these values. VVs may migrate from their current host to another RV to approach desired points in space, but it is the VVM that assigns VVs to RVs. VVs that travel on-board RVs experience *physical mobility* and VVs migrating from one RV to another RV undergo *cyber-mobility.* We consider *physical mobility* as a larger-time-scale motion and *cyber-mobility* as a small-time-scale hop.

We implemented CPCC as a Platform-as-a-Service (PaaS) environment [12] in Java on top of the ROS framework [2]. This means that our implementation supports all platforms where ROS and Java is supported. VVs are JavaScript programs that use the VVM application programming interface (VVMAPI) to execute information acquisition tasks and to access their own private storage. VVs are completely unaware of the hosting RV's sensor equipment. VVs select abstracted sensors from a system wide catalog, provided by the VVMAPI. The queried sensors in conjunction with the required positions in space, specify an information acquisition task. VVs invoke the VVMAPI with the task and a callback function as parameters for execution. Depending on the task's currently requested position and required sen-

sors, the VVM decides whether the VV may stay on the current RV or must migrate to another RV. The VVM on the target RV schedules the pending tasks for execution and approaches the position of the first task to be processed. Every time the RV reaches a task position, the VVMAPI executes the according VV's callback function to hand over the current sensor values to the VV. After a VV terminates, the VVM migrates it to one of the ground stations.

The problem of coordinating and controlling swarms of autonomous robots has been previously investigated in a variety of contexts. Mathew et al. [11] present path-planning strategies for recharging autonomous vehicles performing persistent tasks. Royo et al. [15] employ a network centered, service oriented architecture to control unmanned aerial vehicle (UAV) fleets. Developers use a distributed application interface, called UAV service abstraction layer, to access predefined services over the network to implement civil missions. Pereira et al. [14] investigate the use of a structure aware computation model. Mobile robots are considered to be able to observe, control, compute, and communicate. They operate upon an abstraction of the structure of the world that entails location and connectivity. A case study of autonomous vehicles performing a environmental monitoring mission demonstrates the applicability of the computation model. Kirsch et al. [4] apply the paradigm of cloud computing to cyber-physical systems. The paper discusses some of the challenges and envisioned solutions, as well as outlines a prototype implementation based on XEN [3]. Huang et al. [9] explore possible performance guarantees for virtualized robots. The authors claim that providers supporting a given number of virtual vehicles need significantly fewer real vehicles to guarantee high performance isolation.

Our contribution is a PaaS CPCC implementation that provides a strict separation of operating robot infrastructure and performing missions. Customers implement their VVs as programs completely unaware of the available robot infrastructure. To ensure system dependability, platform providers are able to restrict operational area, altitude and maximum speed for each RV separately. Tasks that cannot be processed by any RV cause the VVM to migrate the concerned VV back to a ground station. The number of RVs affects the performance of the CPCC system, but not its applicability. Hence, providers may start with a small set of RVs and later add RVs achieving better performance or handling a growing number of customers. Naturally, adding RVs is possible at any time. Providers may also remove RVs from the fleet without customer notice for recharging or refueling. The concepts of [9] offer temporal isolation of VVs in a way that providers may even define soft-real-time service level agreements.

The remainder of this document is outlined as follows: Section 2 explains the application by means of an example VV program and simulation runs. Section 3 details the current implementation of our CPCC system. Section 4 offers conclusions and future work.

2. EXPERIMENTS

Figure 1 shows the program code of VV 1 that visits four locations in an altitude of 50 m over ground. VV 1 takes a picture and measures the temperature, as well as the CO_2 concentration at every location and archives the results in the VV storage. *Lines 1-3:* VV 1 queries the definitions of CO_2 sensor, thermometer, and belly mounted camera from

```
1.  var co2 = VV.sensor.get('CO2');
2.  var temp = VV.sensor.get('Thermometer');
3.  var camera = VV.sensor.get('Camera 640x480');
4.  var points = [
5.    new VV.types.LatLngAlt(48.1111, 12.8619, 50),
6.    new VV.types.LatLngAlt(48.1112, 12.8631, 50),
7.    new VV.types.LatLngAlt(48.1115, 12.8646, 50),
8.    new VV.types.LatLngAlt(48.1116, 12.8667, 50),
9.  ];
10. for (var int n=0; n < points.length; ++n) {
11.   VV.task.execute({
12.     type: 'point', position: points[n], tolerance: 2,
13.     sensors: [co2, temp, camera]
14.   },
15.   function(data) {
16.     for (var k=0; k < data.length; ++k) {
17.       VV.storage.store("p"+n+"-"+k, data[k]);
18.     }
19.   });
20. }
```

Figure 1: Program Code of Virtual Vehicle VV 1

the system-wide device catalog (SDC). The camera delivers 640 pixel by 480 pixel images. *Lines 4-9:* Variable `points` holds the four locations as an array of separate `LatLngAlt` objects, each containing the according location's latitude, longitude, and altitude over ground. Positive values for latitude and longitude indicate values in the north and east directions. South and west directions use negative values. *Line 10:* The VV program iterates over the locations in array `points`. *Lines 11-19:* For each location in `points` VV 1 compiles a task containing task type, location, precision to reach the location in meters (*Line 12*), required sensors (*Line 13*), and a callback function to handle the sensor values (*Lines 15-19*). The callback function receives the sensor values as an array, where the indexes of the values correspond to the indexes of the passed sensor definitions. Eventually, the callback function archives each sensor value in the VV storage (*Line 17*). For the first location, `p0-0` is the name of the CO_2 sensor value, `p0-1` is the name of the thermometer value, and `p0-2` is the name of the captured image. VVs may of course archive arbitrary objects in the VV storage and not only sensor values.

Figure 2 shows a snapshot of the simulated flight of one RV serving three VVs. In this experiment the RV starts from its depot at the upper left of Figure 2 and processes tasks from all VVs by applying the nearest neighbor algorithm. Green lines show VV movements in space, that is, when transported by the RV, light-blue lines visualize the intended VV paths, and gray lines mark planned RV trajectories. The black rectangle on the right of the RV shows the RV's name and the VVs it carries. The VV programs generate tasks one by one at run-time, starting with the first task on the left side of Figure 2. Red dots represent already processed tasks and blue dots unprocessed tasks. The green down-arrows show that the unprocessed tasks are scheduled for execution and the gray oblongs display the related VVs and the sensor values to be measured. In Figure 2 the RV has already processed the first three tasks of each VV. The program code of VV 1 is shown in Figure 1. The programs of VV 2 and VV 3 are similar, only the task coordinates differ.

Figure 3 presents a simulated flight of two RVs serving three VVs. In this experiment the RVs operate in separate

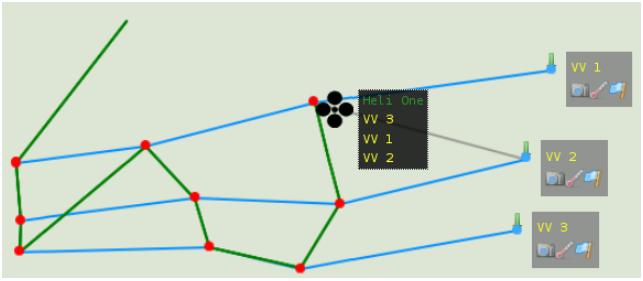

Figure 2: A simulated flight of one RV serving three VVs. The RV (black) starts from its depot (top) and processes tasks from all VVs by utilizing the nearest neighbor algorithm. Green lines reveal VV movements in space, light-blue lines visualize intended VV paths, and gray lines show planned RV trajectories.

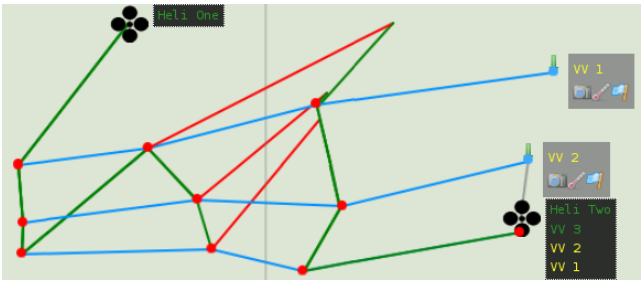

Figure 3: A simulated flight of two RVs serving three VVs. The red lines visualize VV cyber-mobility (migrations) and the vertical gray line indicates the border between the RV's operation areas.

areas. The vertical gray line indicates the border between these areas. RV *Heli One* processes tasks on the left side and RV *Heli Two* is responsible for the right side. Both RVs apply the nearest neighbor algorithm to determine the next task for processing. *Heli One* has already completed the six tasks in its area and has returned to its depot position. In this experiment the first and second tasks of the VVs are located in the operating area of *Heli One*. The remaining tasks are located in *Heli Two's* cell. After *Heli One* has completed the second task of VV 1 the VV migrates to *Heli Two* for the execution of its third task. This is indicated by the upper slanted red line. VV 2 and VV 3 migrate at a later point in time for the same reason to *Heli Two*.

3. SYSTEM

Our system implements information-acquisition-as-a-service of mobile sensor networks as proposed in [5]. Figure 4 depicts an overview of the CPCC system, which consists of customers, administrators, a set of RVs, and one or more ground stations (GSs). The ground station's web frontend (GWF) allows customers to upload their VV programs, as well as view and download results from executed VVs. Administrators employ the GWF to configure the CPCC system and connected RVs. They also manage customer access and the number of VVs a customer is allowed to execute in parallel. The ground station's virtual vehicle monitor (VVM) processes an active VV until it requires the execution of a information acquisition task. Considering the task's position and the required sensors, the VVM decides which

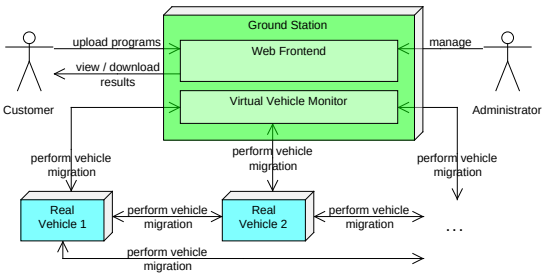

Figure 4: System Overview.

RV should perform the task and migrates the VV to this RV. After a RV has executed a VV's task, the VV remains on the RV until it requires the execution of another task. If the current RV can perform the task, the VV remains on the RV. If the current RV is not able to achieve the task, the RV's VVM decides which RV should perform the task and migrates the VV to this RV. If a VV has terminated, the current RV migrates the VV back to one of the GSs.

From the software point of view GSs and RVs are very similar. We consider a GS as an immobile RV that has no sensors and actuators. Hence, a GS basically has the same functionality as a RV. The main differences between GS and RV are in handling of completed VVs and the administrative part of the GWF. Figure 5 presents the main hardware and software components available in RVs as well as in GSs.

RVs and GSs use an off-the-shelf Linux operating system that is able to execute ROS and suitable driver software needed to access real, as well as simulated, sensors and actuators. The VVM mainly consists of the Mozilla Rhino JavaScript virtual machine (JSVM) [1] and the components quota manager, loader and migrator, vehicle mapper, and task executor. The JSVM executes all VV programs isolated in separate threads. It provides essential functionality to take snapshots of running programs in the form of byte arrays. Snapshots may be stored or transferred to remote machines for later continuation. The loader and migrator (LM) performs the initial load of VVs and arranges VV migrations. The task executor (TE) is responsible for accomplishing VV tasks. After a VV issues a task the TE employs the vehicle mapper to decide whether a migration to another RV should happen or not. In case of a migration the task executor invokes the LM to initiate the migration. If the VV stays on the RV, the TE considers all open tasks, creates a new flight plan, and approaches the first task position. When the RV reaches the task position, the TE triggers the JSVM to read sensor values and resume the VV's execution.

The quota manager is responsible for preventing VVs to utilize resources too much. It terminates VVs that consume too much CPU power or disk storage.

A VV consist of its VV program, JavaScript VM context (CTX), management data (MD), and document storage (DS). The CTX contains the *state* of the VV program, that is, a complete snapshot including program, program counter, and variables. MD contain parameters necessary for a VV's execution, like quota limits, and allowed API calls. A VV may store objects in its DS for later retrieval. The web frontend handles all incoming network traffic and ensures encrypted connections.

VVs do not access sensors and actuators directly for information acquisition. Instead, VVs access virtual abstractions defined in a system-wide device catalog (SDC). The

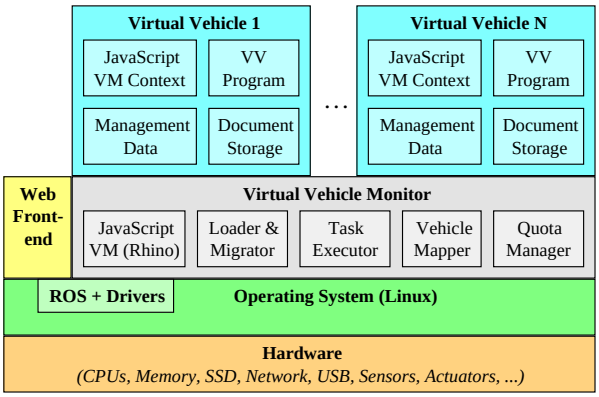

Figure 5: Hardware and Software Components Overview of a Real Vehicle or Ground Station.

SDC specifies for each sensor type and optional supplemental parameters. Currently available sensor types are camera, GPS, thermometer, NO_x, CO_2, barometer, altimeter. For cameras the SDC defines the following supplemental parameters: alignment to the RV, e.g., *heading*, inclination, e.g., *45 degrees down*, as well as width and height in pixels. When adding RVs to the CPCC system, administrators connect SDC definitions to each sensor configuration. This enables VVs to read the values of the desired sensor without knowing the actual RV's hardware configuration. By requesting a task to be executed, VVs control actuators indirectly. It is the responsibility of the TE to control the RV's actuators.

4. CONCLUSIONS AND FUTURE WORK

We have presented a Platform-as-a-Service (PaaS) implementation of cyber-physical cloud computing (CPCC), as proposed in [5], that allows the separation of operating vehicle fleets from performing information acquisition missions. Providers may add RVs to the system for better performance and may also remove RVs without service disruption for maintenance, recharging, or refueling. Clients use in their JavaScript coded VVs the VVM application programming interface to read from abstracted sensors and to access their own private storage.

Possible future work include image processing in VVs, message passing between VVs, automation of RV recharging, better temporal isolation as suggested in [9], utilization of real hardware, and integration of unmanned ground vehicles, as well as autonomous underwater vehicles.

5. ACKNOWLEDGMENTS

This work has been supported by the National Science Foundation (CNS1136141) and by the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund S11404-N23).

6. REFERENCES

[1] Mozilla Rhino. http://www.mozilla.org/rhino.

[2] The Robot Operating System (ROS). http://www.ros.org.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. Symposium on Operating Systems Principles (SOSP)*. ACM, 2003.

[4] C. Kirsch et al. Cyber-Physical Cloud Computing: The Binding and Migration Problem. In *Proc. International Conference on Design, Automation and Test in Europe (DATE)*. EDA Consortium, 2012.

[5] S. Craciunas, A. Haas, C. Kirsch, H. Payer, H. Röck, A. Rottmann, A. Sokolova, R. Trummer, J. Love, and R. Sengupta. Information-Acquisition-as-a-Service for Cyber-Physical Cloud Computing. In *Proc. Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[6] E. Pereira at al. A Networked Robotic System and its Use in an Oil Spill Monitoring Exercise. *Short talk at the International Workshop on the Swarm at the Edge of the Cloud*, 2013.

[7] A. Ghose, P. Biswas, C. Bhaumik, M. Sharma, A. Pal, and A. Jha. Road condition monitoring and alert application: Using in-vehicle Smartphone as Internet-connected sensor. In *Proc. International Conference on Pervasive Computing and Communications Workshops (PERCOM)*. IEEE, 2012.

[8] H. Chen et al. Cloud computing on wings: Applications to air quality. In *Proc. American Astronautical Society Guidance and Control Conference (AASGNC)*. AAS, 2012.

[9] J. Huang, C. M. Kirsch, and R. Sengupta. Scalability of Vehicle Networks through Vehicle Virtualization. *Poster at the International Workshop on the Swarm at the Edge of the Cloud*, 2013.

[10] C.-K. Lin. *Coding-based System Primitives for Airborne Cloud Computing*. PhD thesis, Harvard University, Cambridge, MA, USA, 2012. AAI3495619.

[11] N. Mathew, S. L. Smith, and S. L. Waslander. A Graph-Based Approach to Multi-Robot Rendezvous for Recharging in Persistent Tasks. In *Proc. International Conference on Robotics and Automation (ICRA)*. IEEE, 2013.

[12] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), Gaithersburg, MD, September 2011.

[13] N. Lane et al. A survey of mobile phone sensing. *IEEE Communications Magazine*, 48(9):140–150, September 2010.

[14] E. Pereira, C. Potiron, C. Kirsch, and R. Sengupta. Modeling and Controlling the Structure of Heterogeneous Mobile Robotic Systems: A BigActor Approach. In *Proc. International Systems Conference (SysCon)*. IEEE, 2013.

[15] P. Royo, J. López, E. Pastor, and C. Barrado. Service Abstraction Layer for UAV Flexible Application Development. In *Proc. Aerospace Sciences Meeting and Exhibit (ASM)*. AIAA, 2008.

[16] A. Ryan and J. Hedrick. A mode-switching path planner for UAV-assisted search and rescue. In *Proc. Conference on Decision and Control, European Control Conference (CDC-ECC)*. IEEE, 2005.

[17] A. Ryan, J. Tisdale, M. Godwin, D. Coatta, D. Nguyen, S. Spry, R. Sengupta, and J. K. Hedrick. Decentralized control of unmanned aerial vehicle collaborative sensing missions. In *Proc. American Control Conference (ACC)*. IEEE, 2007.