ESE CPCC Project

M. Kleber, C. Krainer, A. Schröcker, B. Zechmeister

Department of Computer Sciences
University of Salzburg
Austria

WS 2011/2012
February 10, 2012

Abstract

For the hands-on part of the Embedded Software Engineering Course, Winter 2011/2012 of Prof. C. Kirsch (Department of Computer Sciences, University of Salzburg) we decided to build a simulation system that demonstrates information-acquisition-as-a-service of mobile sensor networks for cyber-physical cloud computing.

This paper presents the main concepts of our project. First, this document describes the goals of this project, followed by implementation characteristics, limitations, and applied technologies. Second, this document outlines the system, reveals sensor simulation, explains configuration parameters, describes vehicle virtualization, and pictures cyber-mobility. Third, it elaborates on the class demonstrations, which exemplify data collection, vehicle migration, and varying sensor equipment. Finally, this paper summarizes the implemented system and depicts proposals for future enhancements.

Contents

1 Introduction **3**

2 Implementation **4**

2.1 System Overview 4

2.2 Sensor Simulation 5

2.3 Configuration 6

Pilot Web Application 6

Engine Web Application 8

Mapper Web Application 8

GM-Viewer Web Application 9

2.4 Vehicle Virtualization 9

Virtual Vehicle Programming Language 9

Parsing Virtual Vehicle Programs 10

Executing Virtual Vehicle Programs 10

Migration 11

Types of Mobility 11

2.5 Mapping 11

Registration Service 11

Mapper 12

3 Results **14**

3.1 Demonstration 1: Data Collection 14

3.2 Demonstration 2: Migration 15

3.3 Demonstration 3: Different Sensors 17

3.4 Demonstration 4: Multiple Virtual Vehicles 18

4 Conclusion **20**

References **21**

List of abbreviations **22**

1. Introduction

Our goal was to build a simulation system that demonstrates information-acquisition-as-a-service of mobile sensor networks for cyber-physical cloud computing (CPCC) as proposed in [4]. Based on the JNavigator project [6] our implementation provides

- the simulation of physical helicopter swarms,
- the simulation of sensors,
- the virtual abstraction of autonomous vehicles (virtual vehicles), and
- the migration of virtual vehicles among flying physical helicopters (real vehicles).

We consider this project as a first step into the domain of information-acquisition-as-a-service and therefore allow the following limitations:

- Real vehicles follow strict flight plans.
- There are no network bandwidth limits.
- There are no processing power limits.

In this project,

- we apply HTTP [1] as protocol for sensor abstraction and data exchange,
- we use Java as programming language,
- we implement the software as web applications, and
- we utilize Apache Tomcat [3] as web server and servlet container.

This document describes the highlights of the implemented software. Chapter 2 reveals the implementation details, chapter 3 describes the project results, and chapter 4 summarizes this paper and depicts proposals for future enhancements.

2. Implementation

This chapter presents the implementation of our system. After outlining the structural elements, it briefly describes the simulation of sensors and explains configuration parameters. Then, this chapter describes vehicle virtualization, and elaborates on cyber-mobility.

2.1 System Overview

Figure 2.1 presents an overview of the complete system containing the ground station and one simulated helicopter referred to as Real Vehicle (RV).

The simulated RV mainly comprises an Apache Tomcat web container, which executes a Pilot web application and an Engine web application. The Pilot web application consists of a model helicopter plant simulator, that is, the MockJAviator, a flight control system, an auto pilot, and a sensor simulator. The MockJAviator emulates the helicopter's flight dynamics and Inertial Measurement Unit (IMU), the flight control system operates attitude and altitude of the simulated vehicle, and the autopilot stirs the simulated vehicle along a Vehicle Control Language (VCL) script defined trajectory. The sensor simulator supports GPS receivers, belly mounted cameras, thermometers, barometers, and sonar sensors. The Engine web application consists of a Virtual Vehicle

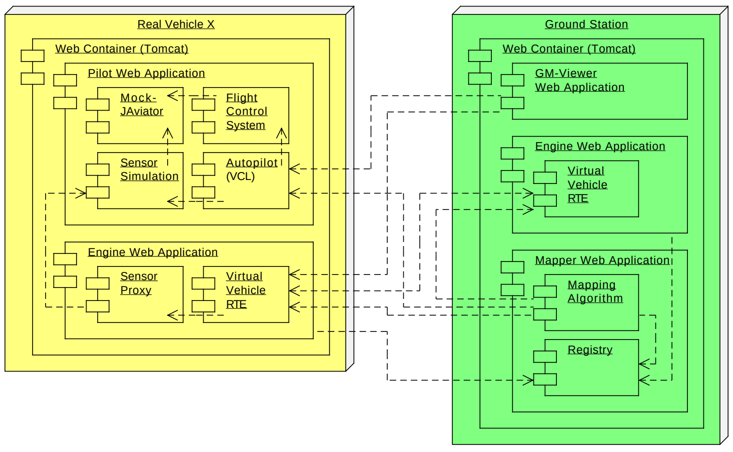


Figure 2.1: System Overview.

Runtime Environment (VV RTE) and a sensor proxy. The VV RTE handles Virtual Vehicle (VV) execution and the sensor proxy abstracts the access to sensors, as well as optimizes the network traffic for accessing the sensors.

The ground station executes an Apache Tomcat web container, which runs a Google Maps Viewer web application, a Engine web application, and a Mapper web application. The Google Maps Viewer web application allows an operator to supervise ongoing missions. The ground station Engine web application provides a VV RTE for uploading and downloading VVs. Registry and mapping algorithm are the main components of the Mapper web application. The mapping algorithm assigns VVs to RVs, based on fight plans and available sensors. Each Engine web application registers with the Mapper's registry.

2.2 Sensor Simulation

Figure 2.2 visualizes the simulation of sensors. The current implementation supports belly mounted cameras, random number generators for emulating thermometers and barometers, GPS position sensors, and sonar sensors.

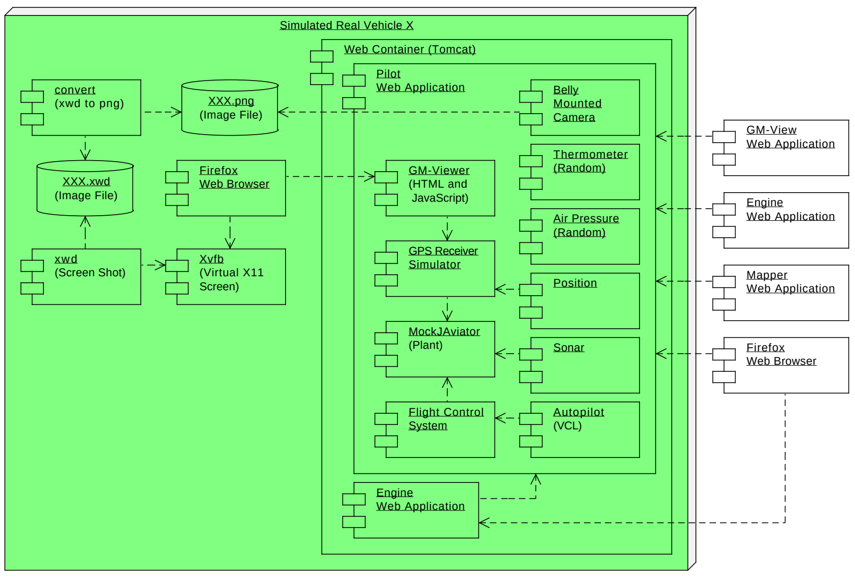


Figure 2.2: Sensor Simulation.

The belly mounted camera delivers a Google Maps satellite image indicating the position of the simulated vehicle by means of a visual marker, exemplified by Figure 2.3. To achieve this, the Pilot web application supplies a JavaScript enhanced HTML page.

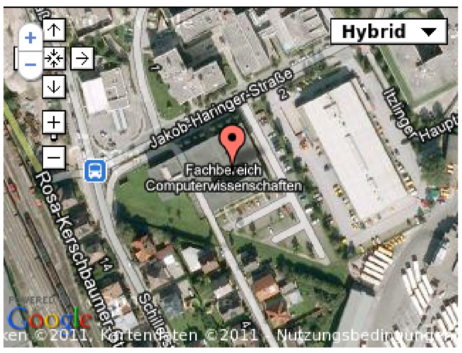

Hybrid
Jakob-Haringer-Straße
Fachbereich Computerwissenschaften
Itzlinger Haupt
Rosa-Kerschbaumer
Schillst
ken ©2011, Kartendaten ©2011 , Nutzungsbedingungen

Figure 2.3: A photo captured by the belly mounted camera.

Once the Firefox web browser selects this page, the embedded JavaScript program periodically polls the Pilot web application for the vehicle's current position. After that, the JavaScript program slides the center of the displayed satellite view to this position and repositions the marker. To allow several belly mounted cameras being simulated on the same machine, the utilized web browsers use X11 virtual frame buffer (Xvfb) devices as output screens. Whenever the belly mounted camera needs to deliver a photo, it applies the `xwd` utility to take a snapshot of the corresponding Xvfb device, converts this snapshot to an image in PNG format by using program `convert`, and delivers it via the Hyper Text Transport Protocol (HTTP).

Thermometer and air pressure sensors apply random number generators to simulate values. The position sensor queries the GPS receiver simulator for the current position of the vehicle, and the sonar sensor reads the current altitude over ground from the instantiated MockJAviator.

2.3 Configuration

This section covers the configuration of the web applications Pilot, Engine, Mapper, and GM-Viewer.

Pilot Web Application

The configuration of the Pilot web application consists of three parts. The first part is the configuration of the simulated model helicopter hardware, the second part is the configuration of simulated sensors, and the third part is the assigned VCL script.

Listing 2.1 depicts a fully simulated RV. With the property `plant.simulated` equal to `true`, the Pilot web application simulates the model helicopter's flight dynamics of the physical body including the IMU. The simulated model helicopter of type `MockJAviator` awaits instructions via UDP on address `localhost` port `9011` from a controller that operates attitude and altitude.

Listing 2.1: Virtual Hardware Configuration Example

```
plant.simulated = true
plant.type = MockJAviator
plant.listener = udp://localhost:9011
plant.location.system.type = gpssim
plant.location.system.listener = tcp://localhost:9012
plant.location.system.update.rate = 10

controller.simulated = true
controller.type = JControl

pilot.type = JPilot
pilot.name = Pilot One
pilot.controller.connector = udp://localhost:9014
```

The simulated model helicopter has a simulated GPS receiver onboard, which listens on `localhost` port `9012` for TCP clients. It delivers the vehicle positions at a rate of 10 updates per second.

In the configuration, shown in Listing 2.1, the Pilot web application also emulates the controller for operating attitude and altitude. The controller uses the property `plant.location.system.listener` to determine how to access the helicopter plant, and uses the property `pilot.controller.connector` to identify the parameters for configuring a connection for incoming commands.

The properties prefixed by `pilot` show the type and name of the autopilot component, as well as define the connection to the attitude and altitude controller.

Listing 2.2 depicts the configuration of simulated sensors. Property `sensor.list` defines the sensors to be simulated. The names in this list prepended by `sensor` are part of the configuration that follows.

Listing 2.2: Sensor Configuration Example

```
sensor.list = gps, sonar, temp, photo

sensor.gps.name = GPS receiver
sensor.gps.path = position
sensor.gps.uri = gps:///

sensor.sonar.name = Sonar
sensor.sonar.path = sonar
sensor.sonar.uri = sonar:///

sensor.temp.name = Thermometer
sensor.temp.path = temperature
sensor.temp.uri = rand:///18/22
sensor.temp.precision = 1

sensor.photo.name = Belly Mounted Photo Camera
sensor.photo.path = photo
sensor.photo.uri = x11:///:21
sensor.photo.type = snapshot
```

Each sensor configuration defines a `name`, a `path`, and an `uri` parameter. The parameter `name` of a sensor simply indicates its name, which is used for visualization only. The parameter `path` is a suffix to an URL dependent on the deployment context of the corresponding Pilot web application. Let's assume, for example, the Pilot web application is deployed in a web container's context `/pilot` listening on machine `nanook` port `9010`. To access the Pilot web application, an operator uses the URL `http://nanook:9010/pilot`. In this example the URL to access the above configured belly mounted photo camera is `http://nanook:9010/pilot/sensor/photo`.

GPS receiver and sonar sensor only have the protocol part of their URI specified to indicate their data source. The thermometer utilizes a random number generator to simulate values between 18 °C and 22 °C providing a precision of one decimal digit.

The belly mounted photo camera captures snapshots of the X11 screen identified by display number 21.

Listing 2.3 illustrates a VCL script example. The character `#` indicates lines containing comments. The first command in this example, “`go auto`”, switches the vehicle into autopilot mode. Without this command, all following commands are ignored. Command “`takeoff`” starts the vehicle’s engines and lifts it off the ground to an altitude of 1 m within 5 s. The “`fly to`” commands define waypoints the vehicle has to traverse by specifying latitude, longitude, and altitude above ground in absolute values. Additionally, this commands define a certain precision to determine when a waypoint has been reached, e.g., a sphere of 1 m radius. Furthermore, this commands assign an average velocity to approach waypoints, e.g., 2 m/s.

Listing 2.3: Vehicle Control Language Example

```
##
## @(#) real vehicle set course example
##
go auto
takeoff 1m for 5s
fly to (47.82204197, 13.04086670, 20)abs precision 1m 2mps
fly to (47.82206088, 13.04092035, 20)abs precision 1m 2mps
fly to (47.82195102, 13.04488063, 20)abs precision 1m 2mps
hover for 20s
land
go manual
```

As soon as all waypoints have been traversed, command “`hover`” instructs the vehicle to hold its position at the last waypoint for, in this example, 20 s. After that, command “`land`” directs the vehicle to land. Finally, command “`go manual`” switches back to manual control.

Engine Web Application

As depicted in Listing 2.4, the Engine web application configuration consists of two parameters. Parameter `pilot.url` specifies the Pilot web application the Engine travels with, and parameter `mapper.registry.url` defines the Mapper registry service to register with.

Listing 2.4: Engine Configuration Example

```
pilot.url = http://localhost:9010/pilot
mapper.registry.url = http://localhost:9040/mapper/registry
```

Mapper Web Application

Since every Engine web application registers with the Mapper, the Mapper web application needs no configuration parameters. The selection of mapping algorithms is built-in and not for configuration yet.

GM-Viewer Web Application

Currently, the Google Maps Viewer web application needs to know every Pilot web application and every Engine web application. As shown in Listing 2.5, the JSON [2] query service URLs for the current position, the set course's list of waypoints, and the list of currently assigned vehicles have to be configured for each RV carrying a Pilot and an Engine.

Listing 2.5: Google Maps Viewer Configuration Example

```
pilot.list = one, two

pilot.one.name = Pilot One
pilot.one.position.url = \
    http://localhost:9010/pilot/json/position
pilot.one.waypoints.url = \
    http://localhost:9010/pilot/json/waypoints
pilot.one.vehicle.status.url = \
    http://localhost:9010/engine/json/vehicle

pilot.two.name = Pilot Two
pilot.two.position.url = \
    http://localhost:9020/pilot/json/position
pilot.two.waypoints.url = \
    http://localhost:9020/pilot/json/waypoints
pilot.two.vehicle.status.url = \
    http://localhost:9020/engine/json/vehicle
```

2.4 Vehicle Virtualization

A Virtual Vehicle (VV) is a software implementation providing an isolated environment to the end user of the service. VVs are hosted on Real Vehicles (RVs) and offer an abstracted interface to the host's functionality. It should be possible for the end user to specify the behavior of VVs and the jobs that should be executed by them.

Virtual Vehicle Programming Language

For abstraction of services to end users a dedicated VV programming language has been defined. The language is targeted to be simple in usage. Listing 2.6 shows a program example. The program represents a list of commands. Each command consists of a point and a list of actions that have to be executed at this point. The point definition contains the geographic location (latitude, longitude, altitude) and the specification of a tolerance radius. The tolerance radius defines a sphere around the point in which the actions have to be performed.

Listing 2.6: Virtual Vehicle Sample Program

```
Point 47.82201946 13.04082647 1.00 tolerance 12.3
Picture
Temperature

Point 47.82203026 13.04084659 25.00 tolerance 100
Temperature

Point 47.82211311 13.04076076 30.00 tolerance 1.2
Picture
```

Parsing Virtual Vehicle Programs

When a VV is activated the program file is read and parsed. The implemented scanner allows extending the language by adding control structures like “`if (...) else`” or “`for (...)`” loops. Keywords can be added easily and parsing of doubles, integers, and identifier is already implemented. Parsing is done based on on the EBNF specification shown in Listing 2.7.

Listing 2.7: Virtual Vehicle EBNF

```
Command = Position Action-List
Position = Point Tolerance
Point = "Point" double double double
Tolerance = "Tolerance" double
Action-List = Action {Action}
Action = "Temperature" | "Picture" | "Airpressure" |
   "Altitude" | "Speed"
```

If an error occurs during program parsing, the parser stops and throws an exception. For easier debugging, the exception contains a detailed error description. If the parser succeeds parsing the program, it returns a list of command objects. Each of this objects contains a position and a list of actions. A position includes a point with a tolerance as described above.

Executing Virtual Vehicle Programs

A RV can host one or more VV. Each VVs executes in a separate thread. When a VV is scheduled the first time, it takes the first command from its command list, which was returned from the parser, and starts executing this command. A command can be executed if the position of the RV is within the tolerance sphere specified in its position, that means, the actions, like taking a picture, measuring the temperature, etc., can be performed. If it’s not possible to finish a command, because of a wrong position or actions can not be done, the VV keeps active.

Migration

VVs have the ability to suspend while they are executing their job. In this mode the entire internal state information is persisted to a file. While being in the suspended state it is possible to migrate the VV to another RV host and resume the operation there. To store the state the Java serialization is used. The whole command list is serialized to a file, that means the already finished commands and the command that have to be executed are written to disk. The results of the executed actions are stored in different files. After migration when the state is read back, the VV looks at the command list and begins again to execute commands. All commands which are already executed are skipped. The VV continues with the first unfinished command. Unfinished could also mean that some of the actions where already executed.

Types of Mobility

The migration process is denominated as *cyber mobility*, because no real vehicle movement is taking place when a VV changes its location. Movements that are carried out on board of a not changing RV host are known as *physical mobility*.

Each VV records its movements. This historical track is represented by a list of waypoints. Each waypoint contains a timestamp, the geographic location and the name of the host. The host name is necessary to differ between cyber mobility and physical mobility. Waypoints are stored when a VV executes a cycle and its location has changed more than a specific distance (physical mobility) or its host has changed (cyber mobility). With the information that is stored in the track detailed analysis of VV moving behavior will be possible, which means that there is no way to access this data by now.

2.5 Mapping

The Mapper is responsible for automatically mapping VVs to RVs. To accomplish this, the mechanism invokes the migration of VVs based on a mapping decision made by a mapping algorithm. Additionally to the Mapper itself with its mapping algorithm, a registration service is present.

The servlet stores all registration information persistent in a file, otherwise registered Engines would be lost in a restart. This is important because only registered Engines are considered during the mapping process.

Registration Service

An Engine registers itself with the registration service upon start up using its ID. If the registration was successful, the service fetches some useful information and stores it together with the Engine ID. The fetched information are available sensors and way points (flight plan), in our case, these is static information. If a registration attempt was not successful, the Engine keeps trying to register until it succeeds.

Mapper

The mapper periodically executes the following three steps:

1. Fetch the status of all VVs from all registered Engines. The returned message includes the current active action point, and its unfinished actions.
2. Fetch the status of all RVs on which an Engine is running. The returned message includes the current position, the next position, and the average velocity.
3. Execute the mapping algorithm.

At the time of writing, there are two algorithms available that can be set in the configuration file: a random mapping algorithm and a simple mapping algorithm.

Random Mapping Algorithm

This algorithm does not use any information concerning flight plans and available sensors. It randomly selects two different Engines. If one of these has VVs, then it selects one of this VVs and initiates a migration to the other Engine.

Simple Mapping Algorithm

Listing 2.8 reveals the simple mapping algorithm as pseudo code.

Listing 2.8: Simple Mapping Algorithm

```
for all virtual vehicles do
   if virtual vehicle program is complete
   then
      invoke migration to central engine
   else
      find fastest real vehicle with at least one fitting
      sensor and a distance between the line Current to Next
      and ActionPoint smaller than the tolerance
   endif
   if found vehicle
   then
      invoke migration to it
   endif
endfor
```

The fastest vehicle is the vehicle with the shortest flight time from its current position to the action point of the VV.

Figure 2.4 shows the current flight path of a RV, its current location C, and the end point of the current flight plan segment N. The flight path $\overrightarrow{CN}$ intersects with the tolerance sphere of action point AP, thus the distance is smaller than the tolerance and the RV is a candidate for migration. With the given speed of the RV, the mapping algorithm calculates the flying time of the RV to reach action point AP. The algorithm selects the RV with the smallest flying time to reach AP as a migration target for the

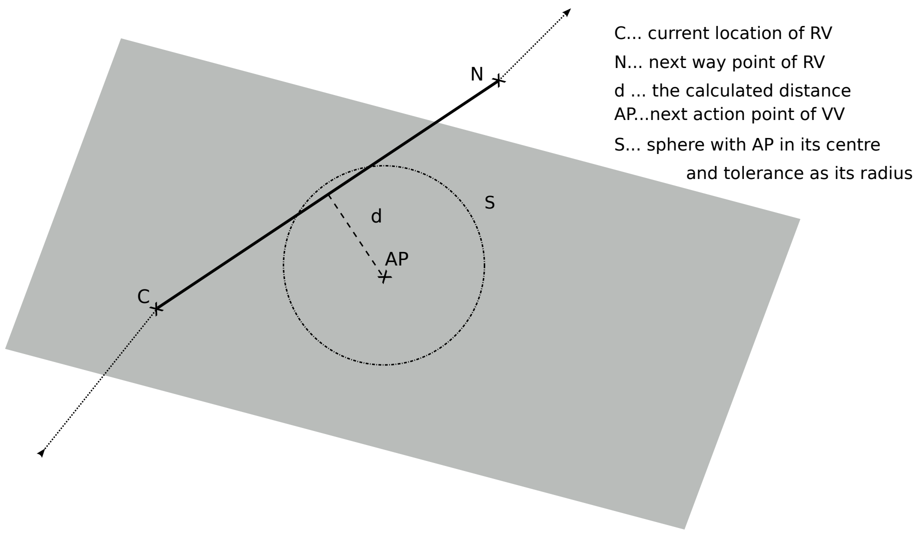

C... current location of RV
N... next way point of RV
d ... the calculated distance
AP...next action point of VV
S... sphere with AP in its centre and tolerance as its radius
N
C
d
AP
S

Figure 2.4: Intersection of flight path and next action point.

concerned VV. With this behavior of the algorithm, a VV stays on a RV or the central Engine until a RV enters a flight plan segment that satisfies the VV's sensor and location requirements.

3. Results

This chapter summarizes the four demonstrations held in class on January 24, 2012. The main goals of this demonstrations were to show data collecting VVs carried by RVs, as well as VVs migrating among RVs. All images shown in this chapter were rendered by utilizing Google Maps [5] in a web browser.

3.1 Demonstration 1: Data Collection

In this demonstration one flying RV carries one VV, which collects data at four locations. Figure 3.1 a displays RV *Pilot One* with VV *VV 1.1* onboard. The color

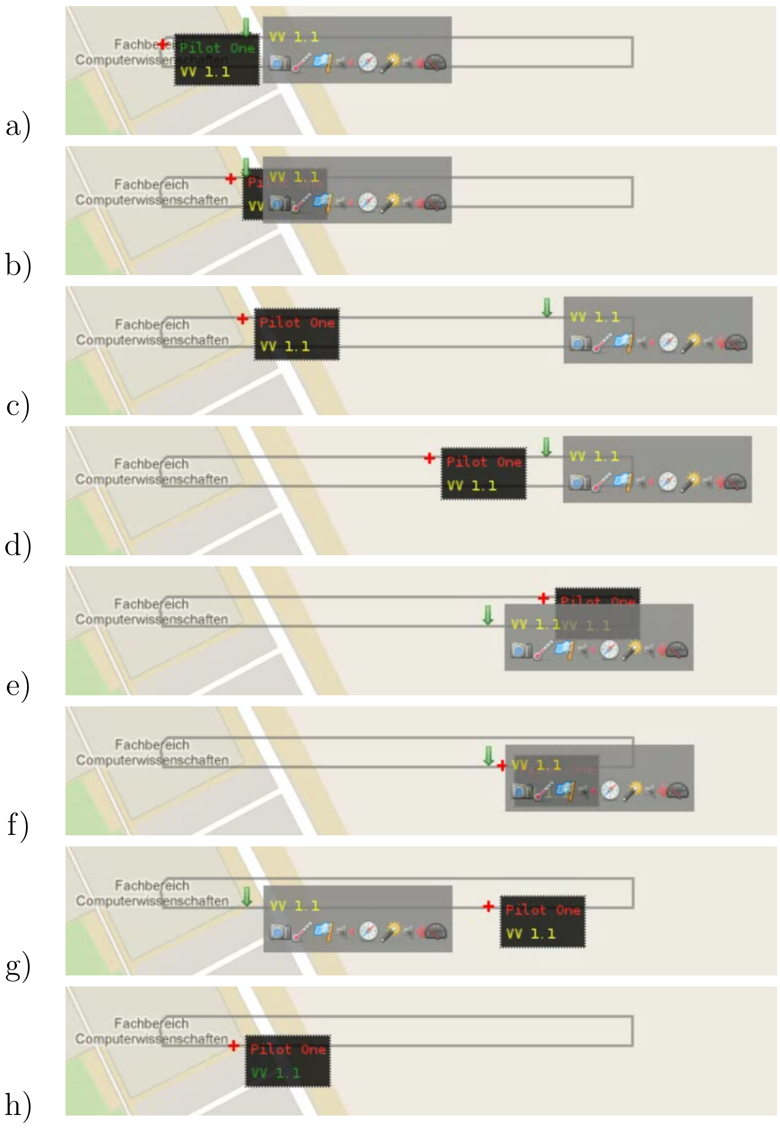


Figure 3.1: Data Collection Demonstration.

green of *Pilot One* indicates that the RV is still on the ground, and the color yellow of *VV 1.1* shows that the VV still has action points to process. The action points of *VV 1.1* command accessing the sensors belly mounted photo camera, thermometer, barometer, sonar, course over ground, random, GPS altitude, and speed over ground. In Figure 3.1 b *Pilot One* approaches the first action point. The red label indicates that the RV is flying. After completing the first action point *Pilot One* approaches the second action point of *VV 1.1*, as depicted in Figures 3.1 c and 3.1 d. Figures 3.1 e, 3.1 f, and 3.1 g visualize *Pilot One* heading for the third and fourth action point. After processing the fourth action point, the VV's label becomes green, which denotes the completion of *VV 1.1*'s mission.

3.2 Demonstration 2: Migration

In this demonstration two RVs fly along their set courses and one VV collects data at five locations. The blue line in Figure 3.2 suggests the virtual path of vehicle *VV 2.1*. Initially, RV *Pilot Two* carries *VV 2.1*. After the RVs take off, a migration

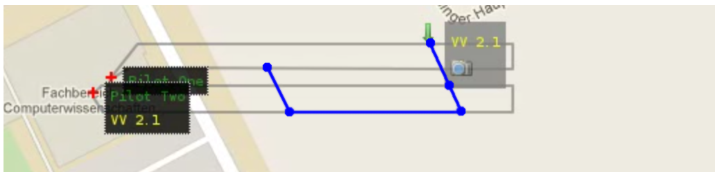


Figure 3.2: Path of Virtual Vehicle *VV 2.1*.

of *VV 2.1* from *Pilot Two* to *Pilot One* must take place to allow the VV to reach the first action point. The currently available mapping algorithm considers only the current set course segment of a RV for mapping decisions. In this demonstration the first action point resides on the second segment of the set course of *Pilot One*. As displayed in Figure 3.3, the mapper migrates *VV 2.1* as soon as *Pilot One* enters its second set course segment. *VV 2.1* resides on *Pilot One* until the RV reaches the

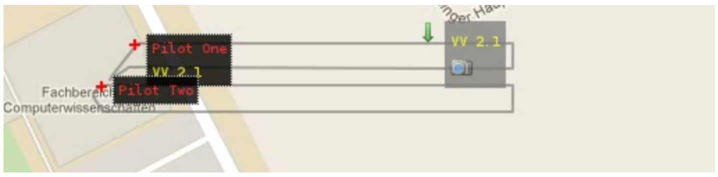


Figure 3.3: First migration of *VV 2.1* from *Pilot Two* to *Pilot One* initiated by a decision of the mapping algorithm.

first action point (Figure 3.4 a). As soon as *VV 2.1* has captured an image via the belly mounted camera, the mapping algorithm decides to initiate a migration of the VV to *Pilot Two* (Figure 3.4 b). Then, *VV 2.1* takes a picture on the second action point. Now, there are no action points in the current set course segments of both RVs. In such a case the currently implemented mapping algorithm can not decide whether migrating the VV would be beneficial or not. So, the VV stays on board of *Pilot Two*

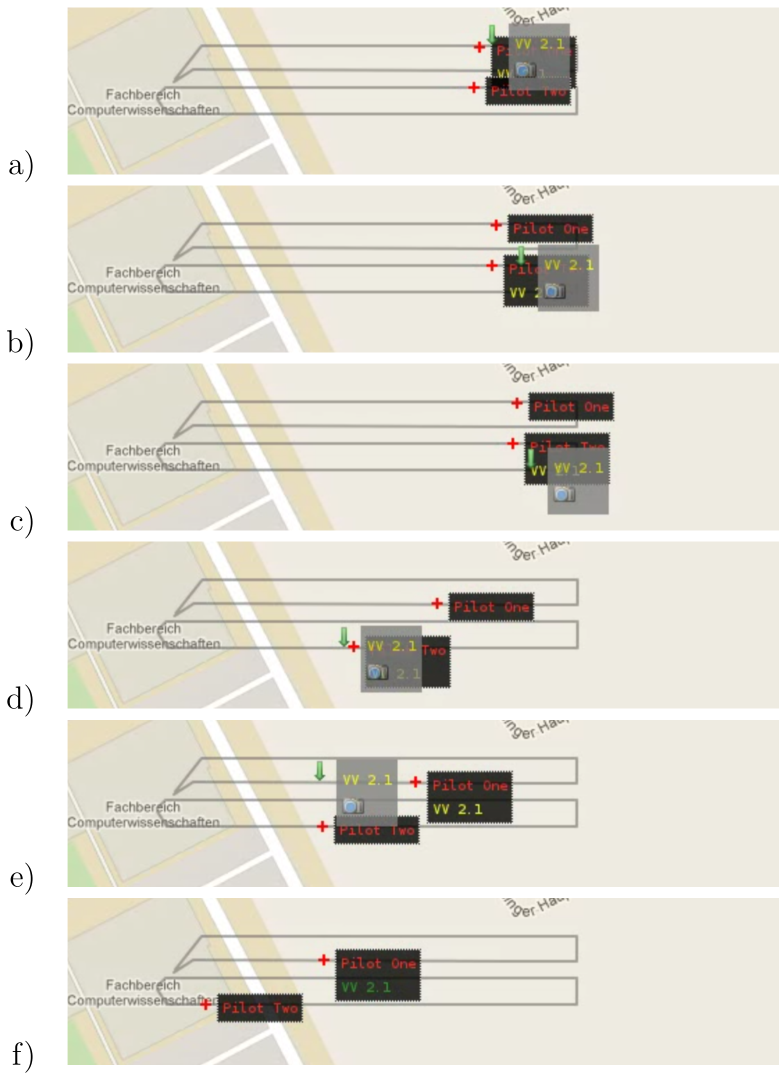


Figure 3.4: *Pilot One* and *Pilot Two* mutually carrying *VV 2.1* caused by migration decisions of the mapping algorithm.

until the mapping algorithm decides otherwise. The RVs continue to traverse their set courses until *Pilot Two* enters the fourth segment. The mapping algorithm detects that *VV 2.1* already is onboard *Pilot Two* and suppresses a migration. In Figures 3.4 c and 3.4 d *VV 2.1* captures photos at the third and at the fourth action point. For the fifth action point, the mapping algorithm decides a migration of *VV 2.1* back to *Pilot One*, as visualized in Figure 3.4 e. After taking a picture at the last action point, *VV 2.1* has completed its mission. To indicate this, the label of *VV 2.1* turns green (Figure 3.4 f). Finally, the mapping algorithm decides a migration back to the central Engine.

3.3 Demonstration 3: Different Sensors

The RVs in this demonstration do not have the same set of sensors. The first RV, *Pilot One*, carries a thermometer, the second RV, *Pilot Two*, ferries a barometer, and the third RV, *Pilot Three*, transports a belly mounted camera. All three RVs follow the same set course in sequence and keep a flying distance of 10 s. The task list of *VV 3.1* consists of two action points, which require capturing photos, temperature values, and air pressure values.

Figure 3.5 a shows all three RVs approaching the first action point. Since *Pilot One* arrives first and provides a required thermometer sensor, the mapper algorithm already initiated a migration of *VV 3.1* to *Pilot One*. At this point in time, the indicator of the first action point visualizes all three required actions as incomplete. After *VV 3.1* has completed the temperature measurement, *Pilot Two* is the next in line to reach the action point supplying a fitting sensor. Therefore, the mapping algorithm commands a migration to *Pilot Two*. As shown in Figure 3.5 b, the action point indicator no more views the thermometer.

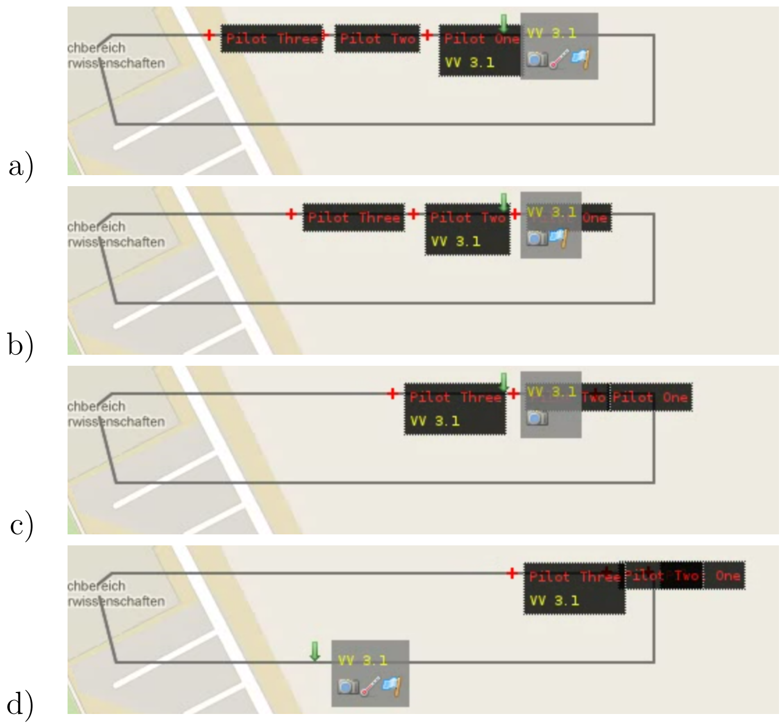


Figure 3.5: RVs *Pilot One*, *Pilot Two*, and *Pilot Three* mutually carrying *VV 3.1* repeatedly to the first action point caused by migration decisions of the mapping algorithm.

Figure 3.5 c presents the situation after *VV 3.1* has measured the air pressure. The mapper algorithm already ordered a migration of *VV 3.1* to *Pilot Three* and the action point indicator views the remaining action for taking a photo. Then, *VV 3.1* stays onboard of *Pilot Three*, because the mapping algorithm can not find a eligible RV for processing the next action point.

In Figure 3.6 a *Pilot One* enters a set course segment that leads to the next action point. Since *Pilot One* facilitates an adequate sensor and is the only one, as the mapping algorithm considers it, to reach the action point, the mapper algorithm directs a migration of *VV 3.1* to *Pilot One*.

As *Pilot One* catches the action point, *VV 3.1* queries the thermometer and the mapper algorithm orders a migration to *Pilot Two* (Figure 3.6 b). Again, the action point indicator no more views the thermometer.

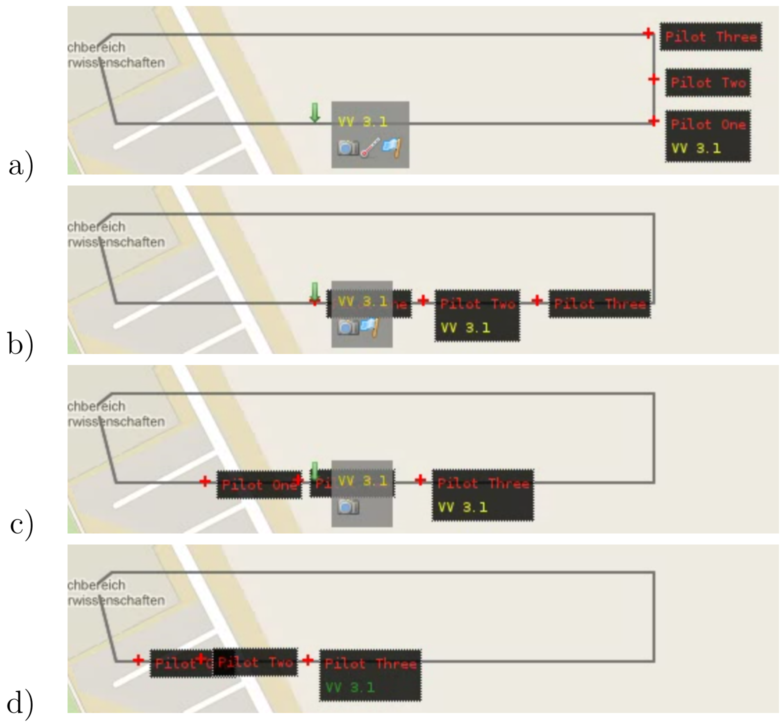


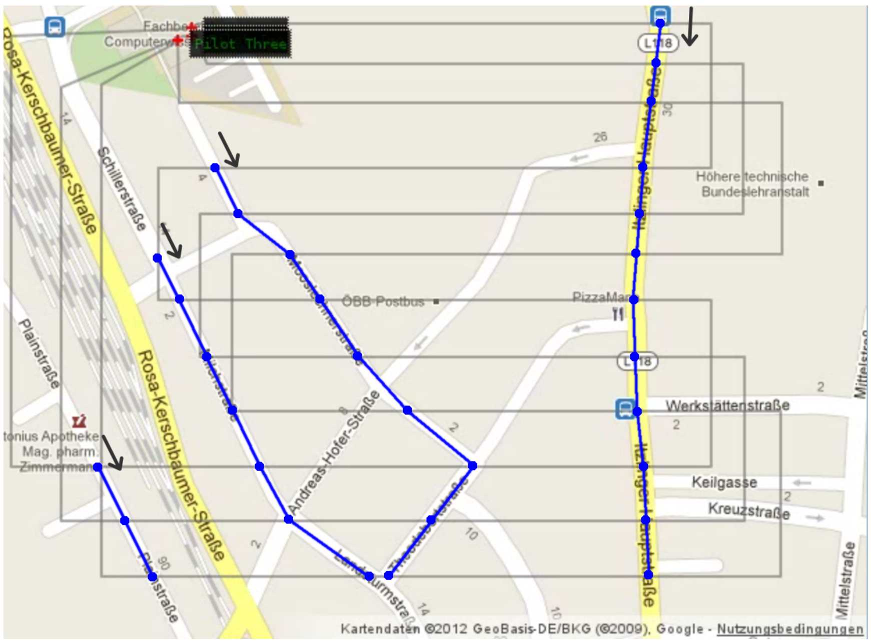
Figure 3.6: RVs *Pilot One*, *Pilot Two*, and *Pilot Three* mutually carrying *VV 3.1* repeatedly to the second action point caused by migration decisions of the mapping algorithm.

After that, *Pilot Two* arrives at the action point, *VV 3.1* measures the air pressure and the mapper algorithm commands a migration to *Pilot Three* (Figure 3.6 c). Consequently, the air pressure symbol vanishes from the action point indicator. Now, *Pilot Three* gets to the action point and *VV 3.1* completes its mission by taking a picture, as shown in Figure 3.6 d. The VV's label turns green to show this. At last, the mapping algorithm initiates a migration back to the central Engine.

3.4 Demonstration 4: Multiple Virtual Vehicles

In this demonstration three RVs fly along their set courses and four VVs collect data at several locations. Each RV provides the same set of sensors. Initially, the VVs idle on the central Engine and wait for the mapping algorithm to assign an eligible RV. The blue lines in Figure 3.7 a display the virtual paths of the VVs. All VV paths progress top-down, as indicated by black arrows.

Figure 3.7 b shows an advanced stage of this demonstration mission displaying all four VVs in action.

a)

Pilot Three
Kartendaten ©2012 GeoBasis-DE/BKG (©2009), Google - Nutzungsbedingungen

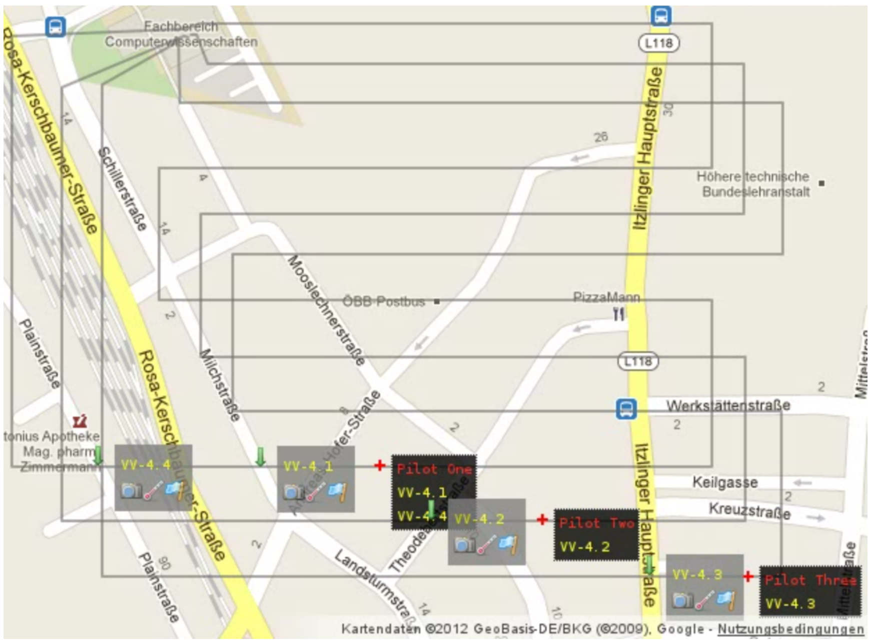
b)


Figure 3.7: Demonstration 4: a) Virtual Vehicle Paths. b) Multiple VVs in action.

4. Conclusion

This work has presented an implementation of a simulation system that demonstrates information-acquisition-as-a-service of mobile sensor networks for CPCC as proposed in [4]. This chapter concludes the paper by summarizing the current situation, and providing suggestions for future enhancements.

Our goal was to implement a flexible and scalable simulation system. We applied the Java programming language and standard Internet technologies like web services to meet this challenge.

The implemented system currently allows the simulation of helicopter fleets of several dozens of vehicles and supports the simulation of sensors like GPS receivers and photo cameras. To simulate air-pressure sensors, temperature sensors, etc. the system utilizes random number generators, which deliver values in a defined range and precision. Support for hardware-in-the-loop testing is available for flight control systems and helicopter plants.

Simulated helicopters follow strict flight plans, but do not access the onboard sensors for data collection. It is a virtual abstraction of autonomous vehicles, Virtual Vehicles (VVs) for short, that gathers data. One helicopter is able to carry several VVs. To complete their missions, VVs may migrate between helicopters.

Future works could cover the following topics:

- The implemented mapping algorithm considers only the current flight plan segment for migration decisions. Future implementations should include all helicopter set course segments.
- Flight plans for helicopters should be derived from VV mission requirements.
- Although the implementation is able to simulate dozens of helicopters, the network traffic between helicopters and ground station needs optimization to achieve higher scalability.
- More advanced camera sensors may allow for directing the sensors towards defined targets, which requires extending the VV programming language.
- In VV missions action points define where to capture sensor values. Video cameras and other streaming sources need new VV programming language commands to trigger recordings.

References

[1] *RFC 2616, Hypertext Transfer Protocol – HTTP/1.1*, June 1999. http://www.ietf.org.

[2] *RFC 4627, The application/json Media Type for JavaScript Object Notation (JSON)*, July 2006. http://www.ietf.org.

[3] Apache Software Foundation: *Apache Tomcat Project*, 2012. http://tomcat.apache.org.

[4] Craciunas, S.S., Haas, A., Kirsch, C.M., Payer, H., Röck, H., Rottmann, A., Sokolova, A., Trummer, R., Love, J., and Sengupta, R.: *Information-acquisition-as-a-service for cyber-physical cloud computing.* In *Proc. Workshop on Hot Topics in Cloud Computing (HotCloud)*. USENIX, 2010.

[5] Google Inc.: *Google Maps*, 2012. http://maps.google.at.

[6] Krainer, Clemens D.: *JNavigator - An Autonomous Navigation System for the JAviator Quadrotor Helicopter.* Master's thesis, University of Salzburg, Austria, 2009.

List of Abbreviations

CPCC cyber-physical cloud computing

HTTP Hyper Text Transport Protocol

IMU Inertial Measurement Unit

JSON JavaScript Object Notation

RV Real Vehicle

TCP Transmission Control Protocol

UDP User Datagram Protocol

URL Unified Resource Locator

URI Unified Resource Identifier

VCL Vehicle Control Language

VV Virtual Vehicle

VV RTE Virtual Vehicle Runtime Environment

Xvfb X11 virtual frame buffer