Computer Science • 14 (3) 2013 http://dx.doi.org/10.7494/csci.2013.14.3.403

Łukasz Nocuń
Michał Nieć
Paweł Pikuła
Aleksander Mamla
Wojciech Turek

CAR-FINDING SYSTEM WITH COUCHDB-BASED SENSOR MANAGEMENT PLATFORM

Abstract *The growing performance of low-cost mobile devices makes it possible to perform advanced processing on mobile sensors. This creates the need to building a management system for groups of sensors actively analyzing signals from hardware devices. In this paper, an architecture of a CouchDB-Based Sensor Management Platform is presented and its application for the problem of finding stolen cars is shown. Detailed performance tests of the platform as well as its application are provided.*

Keywords sensor networks, mobile devices, car plates recognition

1. Introduction

Many achievements of science and technology have been successfully utilized by law enforcement services to improve the rate of crime detection and increase public safety. One such crime which would likely be reduced by using recent scientific advancements is car theft.

Hundreds of thousands of cars are stolen in the EU each year [1]. Thieves often use the stolen car's license plates for some time, trying to move the car while hiding in heavy urban traffic. Automatic methods of recognizing the plates of moving cars based on a distributed monitoring of urban areas could help find such cars and reduce the number of thefts.

The problem requires the use of many mobile devices capable of acquiring such visual data. Assumptions might suggest utilizing one of the well-established sensor network management systems, like TinyDB [2] or SwissQM [3]. The domain of wireless sensor networks (WSN) focuses on using very simple devices for collecting raw data from the environment. Considered issues include limited communication ranges, network consistency, protocols optimization, and power management [4]. Typically, a WSN can be queried for data from a particular sensor or area. The data is collected or processes in the central unit of the network.

A distributed monitoring system which could successfully perform complex data analysis, like recognizing the plates of cars moving in a large urban area, requires a different approach. The amount of visual data collected by a large number of devices cannot be transferred to a central processing system and cannot be analyzed in a centralized manner. These issues have been previously discussed in [5].

A different approach (which was used in the system presented in this paper) assumes a distribution of data processing. Mobile devices are no longer reduced to functioning merely as data acquisition units, but to perform complex data analysis and notify the central server of the system only when important information is detected. The central server is responsible only for defining tasks for mobile devices and collecting information. The server is not required for the mobile devices to work – it can be temporarily switched off without any impact to the system. The creation of such systems is becoming possible due to the significant increase of computational power of low-cost mobile devices observed in recent years.

The particular task of finding cars, which is considered in this paper, requires an algorithm for identifying car plate numbers in images collected by mobile devices. The problems of automatic number plate detection [6] and recognition [7] have received enormous attention over the past few decades. A good survey on the methods can be found in [8]. Typically, the problem is divided into two consecutive stages: plate extraction and character recognition.

Plate extraction. The goal of the first stage is to select an appropriate region of interest within an image where a possible plate may be located. Two of the most popular ways to accomplish this are through morphological operations and signature matching. Morphological operations can enhance the plate area so it is possible to use connected component labeling to get several candidate polygons. Position, shape, and color filters are used to find only the correct car plate patterns. The signature matching method involves calculating image histograms after applying an edge detection filter. An area with more vertical lines is a good candidate for analysis.

Character recognition. Optical character recognition is a wide area of expertise not solely limited to car plate recognition. In some cases, any general purpose OCR algorithm could be used, but the results may be poor. A better idea is to use an algorithm which is designed with this specific requirement in mind. Custom designed neural net recognizers and kNN character classifiers are examples of techniques which take into account additional information on character size, position, and font.

The main contribution of this work is the novel architecture of the scalable and flexible sensor management platform. The simplicity of implementation combined with the provided features make this approach very promising. The car plate recognition algorithm is used as an example of a complex processing task which can be executed successfully on mobile active sensors. The algorithm used here does not go far beyond state-of-the-art in this domain.

The sensor management system presented in this paper is inspired by the Erlang-based sensor management framework presented in [9]. The assumptions and aims are similar; however, a new design of the architecture helped overcoming significant limitations. The new system has been successfully integrated with an image processing algorithm executed on mobile devices which is capable of detecting car plate numbers.

In the next two sections, the architecture of the CouchDB-based sensor management platform is presented in details, and its performance tests are presented. In the section 4, the car plates recognition algorithm is described, and basic performance results are provided. Section 5 contains results of real-life experiments which prove usability of the system in particular conditions and show its limitations.

2. Architecture of the CouchDB-Based Sensor Management Platform

The aim of the platform was to create a unified system to manage a network of mobile sensors. The system should provide tools to monitor the state of sensors, define tasks that can be executed by sensors, and provide storage for discovered information. The platform also should supply methods to browse, filter, and analyze information collected by the sensors. The general architecture is shown in Figure 1.

The system is strongly based on the CouchDB database. All crucial requirements of the platform, like information storage and robust communication, are supported by the database. Successful operation depends mostly on proper deployment and configuration of the CouchDB system.

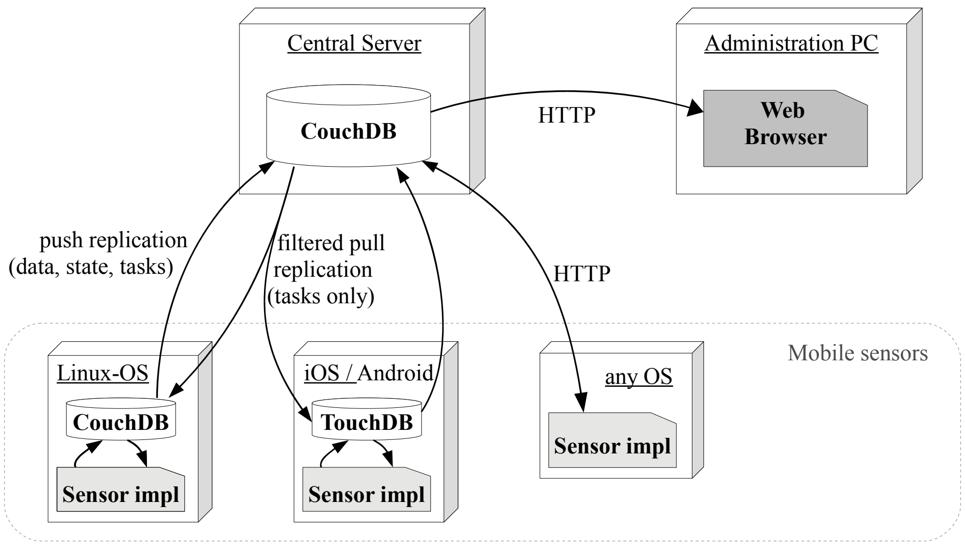


Figure 1. General architecture of the platform.

2.1. CouchDB

Apache CouchDB is a NoSql, schema-free database. It is a document-oriented database – data is stored as documents represented with the use of JSON standard. Data that cannot be represented as JSON object (e.g., pictures, binary files) is stored as an attachment.

Documents in the CouchDB database are stored in a flat address space. Each document is identified by a unique ID. There are no correlations between separate documents. Because of this, CouchDB introduces a new model of data filtering and querying. Data is retrieved from the database with use of views – special functions written in JavaScript. Views take a CouchDB document as an argument and determine if the document should be retrieved in view results. Views are also stored in the database as documents; but to distinguish them from other data, these documents are stored as design documents. Views act as the map part in a MapReduce system.

CouchDB has a built-in fault-tolerant data replication mechanism. It supports both master-slave and master-master replication that can be used between different, geographically-spread instances of CouchDB or on a single instance (e.g. to create backup). All interactions with CouchDB are done via a straightforward HTTP interface.

The unique features of CouchDB (such as document oriented storage, built-in replication, and universal HTTP interface) were the main reasons why it has been used as the core element of the platform. In the sensor management platform, it is used for:

- Providing Web GUI application using a built-in Web server.
- Managing sensors and their configuration. A sensor registers a local instance of CouchDB which causes the information replication in the central server. The same mechanism is used for providing configuration for the sensor.
- Storing tasks and distributing them among the sensors using filtered pull replication.
- Collecting information discovered by the senors. Each sensor can store data directly in the central database, using its HTTP interface. For more robust operations, it can also use a local database which will automatically synchronize with the central server when a network connection is available (push replication in Fig. 1).

2.2. Sensors

Sensor is a single application controlling a particular physical sensor or group of sensors. It acquires data from the hardware, performs a specific analysis, and delivers processed information to the database. Communication with the platform is done via HTTP requests; therefore, sensors can be implemented in any programming language supporting such requests. This flexibility allows us to build sensor applications on various devices like PCs, mobile phones (running Android/iOS/Windows Mobile), and other ARM-based devices.

Because of poor connection quality provided by sensor communication modules (such as GPRS), there is a need for caching collected data and then synchronizing it with the main database. The simplest way to achieve this is to start local CouchDB and use it in the same way as the main database. The only additional thing which should be configured is replication. As a result, we have a fault-tolerant, self-synchronizing system out of the box. We have tested the replication in a network with 80% packet drop and everything worked without any problems – all the data stored in a sensor database was correctly replicated in the main server.

CouchDB can be easily deployed on any Linux system which include ARM distributions; however, on operating systems installed on smartphones or tablets (iOS and Android), it has limited access rights (no access to the root account). There are two solutions to this issue: Couchbase Mobile (which is no longer supported) and TouchDB. Both are CouchDB-compatible database engines. The first one is Couchbase port for Android and iOS written in Erlang. TouchDB implementation is not based on CouchDB – it is a platform-specific implementation. TouchDB is characterized by low memory and CPU usage. It starts many times faster than Couchbase Mobile. TouchDB in comparison to CouchDB is like SQLite to MySQL.

The platform requires each sensor to create a sensor document and update it every 30 seconds to be considered as active. The sensor document contains fields describing sensor features such as unique id, type, state, name, version, available tasks, and configuration. When data is ready to delivery, a sensor should create a new sensor document, include the data, and fill proper fields identifying the sensor.

If the sensor adds a location field, then the results would be marked on the map available in the data view.

Tasks are defined in available_task field of sensor document. The field describes available tasks which, of course, may change during the sensor's lifetime. The simplest example is start and stop tasks – if the sensor is running, the start task should not be visible to user. Task description contains the name as well as in/out parameters defined by name and type. Thanks to this description, the GUI can display proper input form and it can be easily extended for non standard cases.

The sensor periodically queries the database for new tasks, processes them, and updates the task document (changing its state and attaching result or error description). To receive new tasks, the sensor can query the remote database using a suitable CouchDB view or setup filtered pull replication to its own local database. If a short response time is required and regular polling is a problem, long polling strategy on CouchDB changes feed can be used. Then, the sensor is notified when a new task arrives.

During platform development, four sample sensors were created. One of them is a reference sensor written in Python which reports the current load average of a CPU. The rest were created for the Android platform. Initially, Couchbase Mobile was used as a cache for data, but it was too CPU-consuming and unstable; thus, it was eventually replaced by TouchDB.

The sensors collect different types of data, including GPS location, camera images, and sound samples. Images and sounds are stored as document attachments. It is worth mentioning that the platform does not enforce any particular data structure: any JSON object combined with an optional binary attachment can be used.

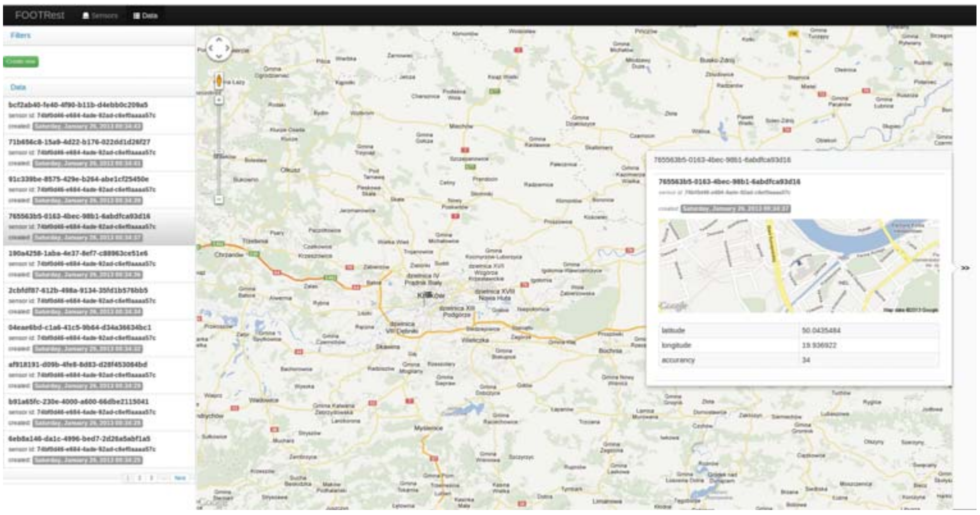
2.3. Cauchapp – Graphical User Interface

One of the platform elements is a Rich Internet Application serving as user interface (Fig. 2). The application helps the user to interact with the platform and the registered sensors. It can be used to:

- monitor sensors state – display sensor details (e.g. sensor's name, type, state etc.),
- manage sensors – display or change configuration, submit new tasks, list all submitted tasks,
- view data collected by sensors – view data from particular sensor or browse data collected by all sensors.

GUI simplifies sensors management and monitoring. Whenever one of the sensors updates its state or sends new data, it is reflected in the GUI. All sensors that provide their location are marked on the map. The user can define new task for a specific sensor, check/update its configuration, or browse recent data sent by it.

Furthermore, GUI provides tools to handle data collected by sensors. It presents any JSON formatted data, displays images, and can playback sound files. To browse data more efficiently, the user can specify his/her own customizable filters.



bcf2ab40-fe40-4f90-b11b-d4ebb0c209a5
sensor id: 74bf0d46-e684-4ade-92ad-c6effaaaa57c
created: Saturday, January 26, 2013 00:34:43

71b656c8-15a9-4d22-b176-022dd1d26f27
sensor id: 74bf0d46-e684-4ade-92ad-c6effaaaa57c
created: Saturday, January 26, 2013 00:34:41

91c339be-8575-429e-b264-abe1cf25450e
sensor id: 74bf0d46-e684-4ade-92ad-c6effaaaa57c
created: Saturday, January 26, 2013 00:34:39

765563b5-0163-4bec-98b1-6abdfca93d16
sensor id: 74bf0d46-e684-4ade-92ad-c6effaaaa57c
created: Saturday, January 26, 2013 00:34:37

190a4258-1aba-4e37-8ef7-c88963ce51e6
sensor id: 74bf0d46-e684-4ade-92ad-c6effaaaa57c
created: Saturday, January 26, 2013 00:34:36

2cbfdf87-612b-498a-9134-35fd1b578bb5
sensor id: 74bf0d46-e684-4ade-92ad-c6effaaaa57c
created: Saturday, January 26, 2013 00:34:34

04eae6bd-c1a6-41c5-9b64-d34a36634bc1
sensor id: 74bf0d46-e684-4ade-92ad-c6effaaaa57c
created: Saturday, January 26, 2013 00:34:32

af918191-d09b-4fe8-8d83-d28f453084bd
sensor id: 74bf0d46-e684-4ade-92ad-c6effaaaa57c
created: Saturday, January 26, 2013 00:34:29

b91a65fc-230e-4000-a600-66dbe2115041
sensor id: 74bf0d46-e684-4ade-92ad-c6effaaaa57c
created: Saturday, January 26, 2013 00:34:29

6eb8a146-da1c-4996-bed7-2d28a5abf1a5
sensor id: 74bf0d46-e684-4ade-92ad-c6effaaaa57c
created: Saturday, January 26, 2013 00:34:29

765563b5-0163-4bec-98b1-6abdfca93d16

765563b5-0163-4bec-98b1-6abdfca93d16
sensor id: 74bf0d46-e684-4ade-92ad-c6effaaaa57c
created: Saturday, January 26, 2013 00:34:37

latitude	50.0435484
longitude	19.936922
accurancy	34

Figure 2. GUI screen with list of data entries collected by sensors. Each item from the list on the left represents a single result sent by one of the sensors. When the user chooses one of the items, entry details are displayed in a separate block. When GPS coordinates are delivered with data, location will be marked on a map. The user can define custom filters to effectively analyze the collected data.

2.4. Features of the platform

The created platform has a very simple architecture; however, it provides several unique features which are worth discussing.

Multiplatform. The only requirement to set up a platform is to run CouchDB instance. CouchDB is available for Linux (both regular PC and ARM based), MS Windows, and OSX. Furthermore, when the platform is up and running, the only thing that the user needs to interact with it is an up-to-date web browser.

Support for heterogeneous sensor devices. Due to CouchDB's HTTP interface, there is no need for any dedicated solutions for communication between the sensors and the server. The sensors use only straightforward HTTP requests/responses.

Extensibility. CouchDB is a document oriented, NoSQL database. Thanks to this, the platform is very flexible in regards to the possible data formats that can be collected by sensors.

Resistance to poor network quality. If the sensors use a local instance of CouchDB, then the built-in fault-tolerant data replication mechanism resolves all issues related to data synchronization between the server and the sensors. Replication is handled by the database without any special actions required from the sensor. Replication is also resistant to connection problems that are quite common in mobile devices.

3. Sensor Management Platform performance

The platform was stress-tested to see how it behaves under a heavy load. The heavy load in our context means thousands of sensors simultaneously sending data every few seconds to the platform's server.

3.1. The premises

The CouchDB database was started in Amazon Elastic Compute Cloud. The machine was a paravirtualized of type "m1.xlarge". Table 1 shows the detailed configuration.

Table 1
Parameters of machine used to run CouchDB instance.

Cores	4
Ram	16 GB
Storage	Amazon EBS (network storage)
Operating System	Ubuntu Server 12.04 (Linux)
CouchDB	1.2.1 (using Erlang OTP R15B03-1)
Network interfaces	1

The sensors were simulated by two other amazon ec2 instances with parameters described in Table 2. Traffic was generated by Tsung [10] program running in distributed mode.

Table 2
Parameters of single machine used to simulate sensor.

Cores	2
Ram	4 GB
Storage	Amazon EBS
Operating System	Amazon Linux
Tsung	1.4.1 (using Erlang OTP R15B03-1)
Network interfaces	1

3.2. Test scenarios

Four scenarios have been prepared and tested in order to test various aspects of platform performance.

5000 concurrent sensors writing to database

This scenario assumed that only sensors were interacting with database uploading data regularly. The mock sensors were being created over the time of 10 minutes with 0.03 seconds gap. Each sensor performed the following steps:

1. generate random sensor id,
2. submit sensor description document,
3. enter loop (repeat 30 times):
    (a) submit data document with random GPS data,
    (b) wait from 1 to 10 seconds.

The platform GUI was not used by any users to prevent CouchDB from generating map-reduce views. This scenario was designed to show how the database performs without any querying from the sensors or users. The test was also a baseline for further testing.

5000 concurrent sensors writing to database while user query the results in real-time

The second scenario was the same as the first one; however, during the test, the user browsed the GUI application, watching sensors which triggered continuous-view generation. The CouchDB map-reduce engine traversed though every new document and generated views which then were stored on the disk. This operation demands computational power. This test was aimed to check how generating a view from a large collection of new or updated documents could affects the overall server performance.

Sensors uploading large attachments while user query the results in real-time

In this scenario, the mock sensors attached a file (30 000 bytes) to every data document. This test was designed to measure the performance for a platform which handles imaging and other sensors which can store large data. The rest of the test parameters were similar to the previous ones.

Maximum load

The last scenario was introduced to determine the maximum number of sensors which the platform can handle. The test was similar to the second one; however, this time, the number of sensors was increased to 12 000 and the gap between spawning each new sensor was reduced to 0.01 second.

3.3. The results

The first scenario ran correctly without any disturbing events from the database. As shown in the Figure 3, the average time of the request (which is saving data document to the server) was between 10 and 15 ms (which is a very satisfying number from the sensor perspective).

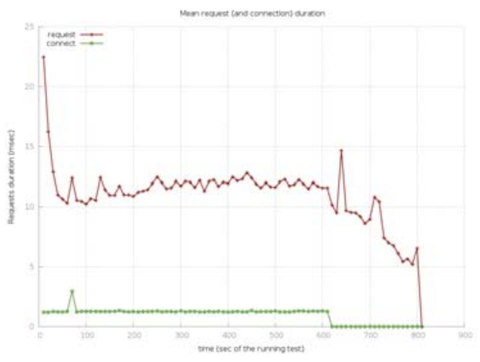


Figure 3. Mean time of saving data document in the database during the first test.

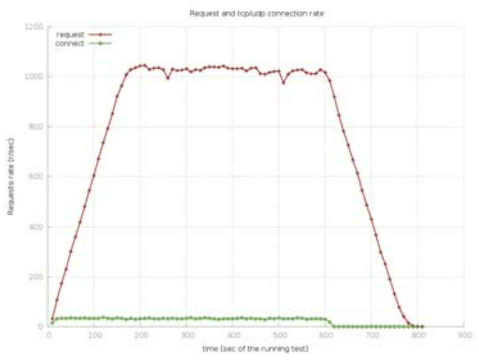


Figure 4. Data submission rate during the 1st test.

During the time when all 5000 sensors were up and sending data, the platform reached a speed of 1,000 requests per second and sustained it during the whole test (Fig. 4). Processor utilization sustained at around 60% and memory consumption was less than 1GB during the whole test (Figure 5 and 6).

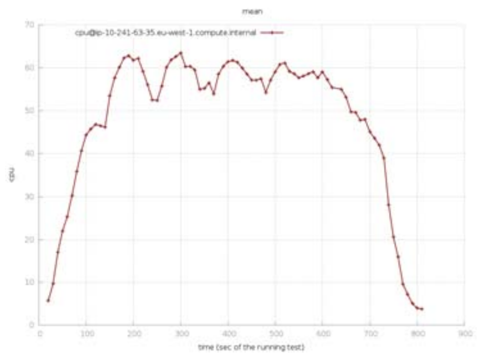


Figure 5. CPU utilization percentage during the 1st test.

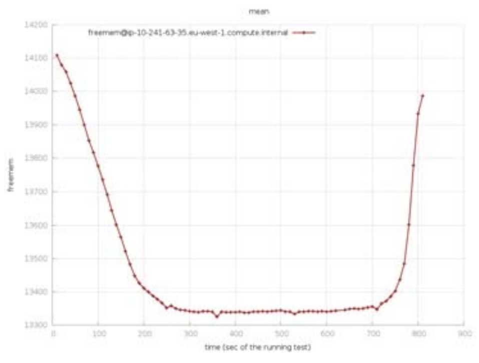


Figure 6. Memory consumption in KB during 1st test

The second scenario was also performed correctly, but there were several moments when response time to sensor requests rose to 600 ms (Fig. 7). Also, the CPU utilization was higher and sustained at 80% (Fig. 8).

Despite occasional performance issues, overall speed was similar and allowed the platform to maintain around 1000 data submissions per second (Fig. 9). Memory usage did not change (Fig. 10). None of the sensor requests ended with an error.

The next scenario – attaching files to the data documents – performed in a completely different manner. It seems like our test machines did not have sufficient

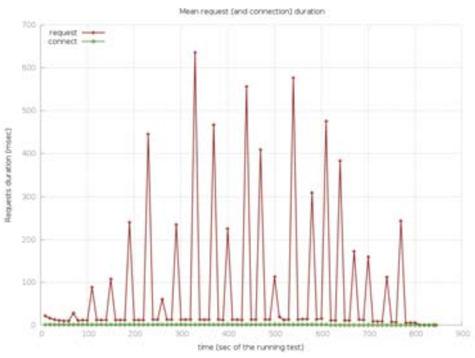


Figure 7. Mean time of saving data document to the database during the second test.

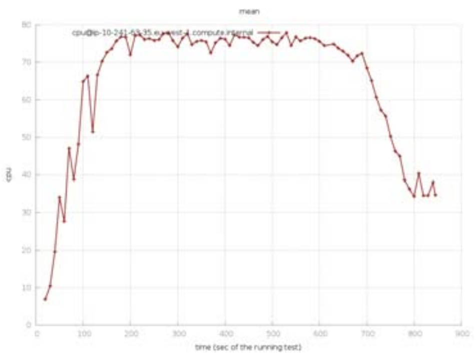


Figure 8. CPU utilization during the second test.

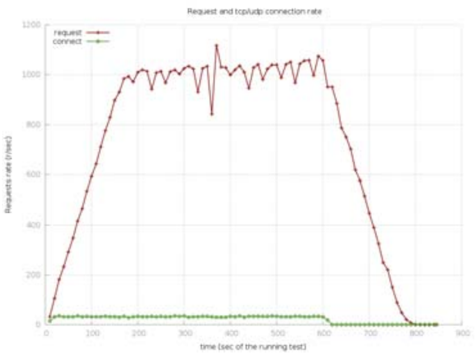


Figure 9. Data submission rate during the 1st test.

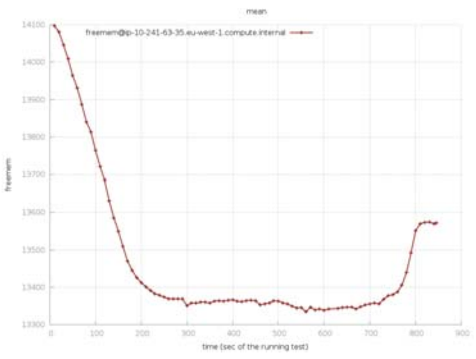


Figure 10. Memory consumption during the 2nd test.

bandwidth to deal with the simulation of many concurrent uploads. So, the load generators could not generate more them 2000 sensors actively uploading documents.

Figure 11 shows clearly the bandwidth issue. The response time constantly increased while the sensors were continuously created and began to upload data. The available bandwidth was divided between more and more concurrent connections. As a result, the upload requests lasted longer. The significant drop in request duration around 600 seconds of test is the thime when most sensors finished uploading their last data document and started to turn off. Therefore, they released bandwidth to the remaining sensors which could upload the rest of their data very efficiently.

The average request rate was stable during the whole test; around the 70–80 data submission per second (Fig. 12).

The processor and memory utilization was 30% (Fig. 13) and memory consumption was 0.5 GB (Fig. 14). These values are much lower than in previous tests and

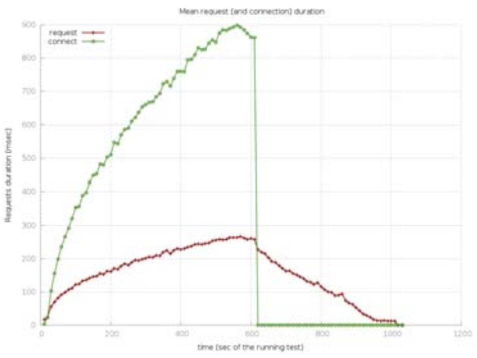


Figure 11. Mean time of saving data document to the database during the third test.

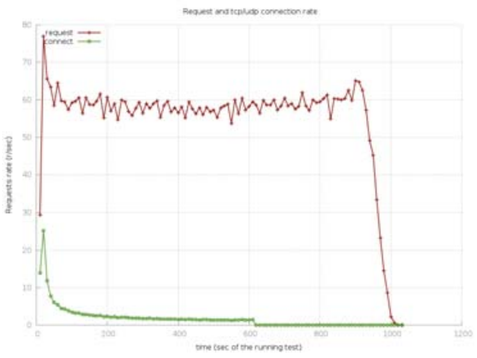


Figure 12. Data submission rate during the third test.

shows that computational power was not a problem in these cases. The network average utilization during the test was 15.14 Mb/sec, and the total size of data sent to server was 1.55 GB.

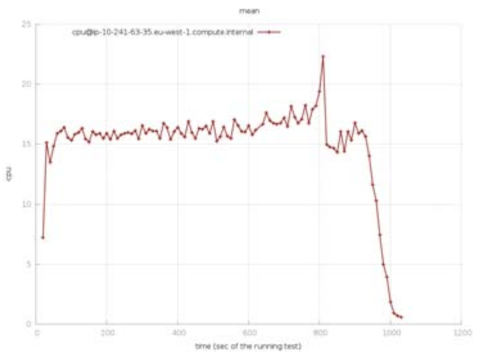


Figure 13. CPU utilization during the third test.

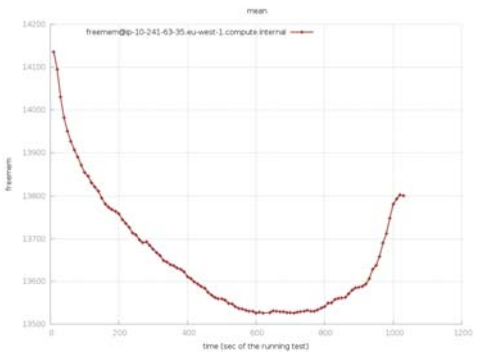


Figure 14. Memory consumption during the third test.

To confirm that networking is the problem, the scenario was rerun with attachment sizes reduced to 10,000 bytes and 20,000 bytes. In both cases, there was no significant change in the request rate nor in CPU and memory utilization. Network utilization, however, was still around the 15 Mbits/sec.

During the last scenario, the maximum number of sensors which could be handled by this setup was reached. The database started to return errors of code 500 to sensors which is unacceptable behavior (Fig. 15). The situation occurred around 300 seconds after the test started. The number of connected sensors went almost to desired 12 000 sensors at that moment.

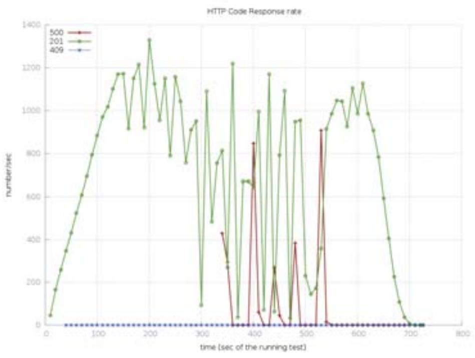
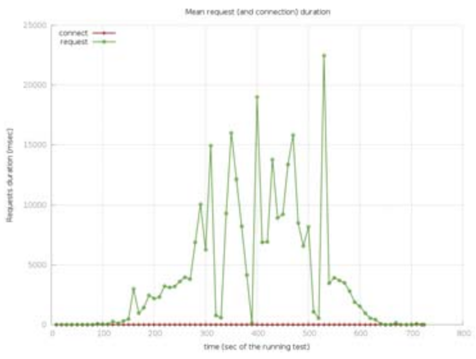


Figure 15. HTTP code rate returned to sensors during 4th test.

Figure 16 and 17 show that the requests times and request rates deteriorated when all 12,000 sensors tried to upload data to the server. After some sensors finished uploading all 30 documents, the performance returned to an acceptable level. CPU utilization (Fig. 18) was high during the whole test, and it was similar to the utilization from the second test. Memory consumption went up to 2 GB (Fig. 19), which is almost twice than in the previous tests.



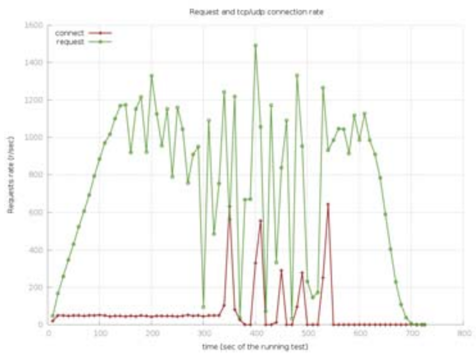


Figure 16. Mean time of saving data document to database during the fourth test.

Figure 17. Data submission rate during the fourth test.

During the last 3 tests, the user interface was responsive and enabled the user to see the incoming data and sensor activity. Due to slow view generation, there was a delay between the state shown in GUI and the actual state of the database. Nevertheless, it never took more than a minute to process all of the documents, and the GUI was usually just a couple seconds behind.

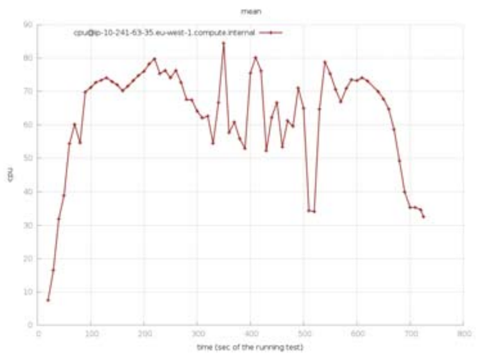


Figure 18. CCPU utilization during the fourth test.

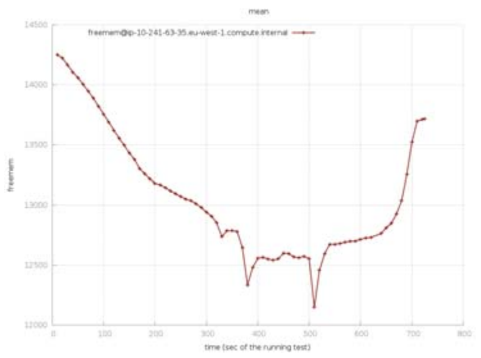


Figure 19. Memory consumption in KB during the fourth test.

3.4. Platform performance tests summary

Comparing these four test results, we can see that generating views is a CPU-intensive task (first and second test), and this can introduce occasional performance drops, albeit at acceptable levels.

CouchDB show great potential when it comes to storing binary files attached to the documents reported by the sensors. During the third test, network connectivity was the bottleneck. The CPU and the memory were almost not used.

The very high load in the fourth test did not crash the database completely. The server recovered from errors which occurred when more than 12,000 sensors submitted data to the server.

4. Car plates recognition algorithm

The task of the designed system is to find a license plate on a given image, recognize its characters, and compare the results with each entry from the list of stolen cars. The best possible recognition system is not necessary; all that is needed is to find a match of a given plate number on an image. The algorithm is expected to return information on the fitness level. The last phase of plate matching will always involve a human who will ultimately decide if the information is consistent with the visual.

The system was designed with real-time camera image processing in mind. It could be used in a car (police patrol, public transport vehicle) or mounted above a street. Weather conditions and the operating environment can affect the image quality. For an optimal operation, a frame rate of 3 per second is defined as sufficient. The algorithm's efficiency and detection rate is highly dependent on camera setup, lighting conditions, and weather. The hardware setup should be chosen according to these requirements. Lighting can greatly vary in consecutive frames so it is important that the frame rate is high to compensate for difficulties induced by environmental factors.

The proposed solution is combining an existing approach based on morphological operations with custom character classifier. The algorithm consists of several consecutive stages,described below.

1. **Preprocessing**
   Before the connected component labeling algorithm can be used, a number of preprocessing operations must be applied. Firstly, the image is converted into gray scale. (Fig. 20-left). Then, a white top hat morphological operation is used to brighten up the plate area (Fig. 20-center). A threshold operation is used to binarize the image (Fig. 20-right). There are only some areas of interest left – some of them represent a car plate.

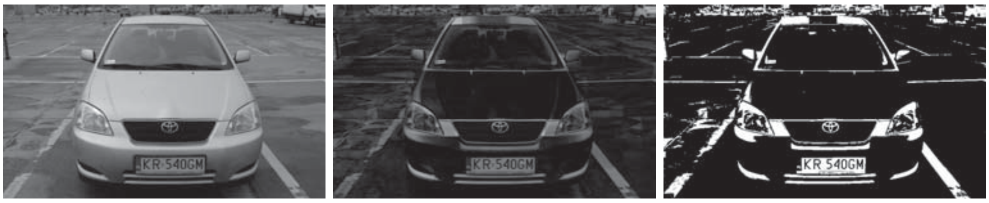


Figure 20. Initial preprocessing of the algorithm: left: conversion to gray scale, center: after a white top hat operation, right: black and white thresholded image.

2. **Plate localization**
   The next step is to divide the white regions and pick the ones that resemble car plate in most ways. Connected components labeling algorithm separate certain parts of the picture (bigger or smaller than a given size). All of the red squares in Figure 21-left are possible plate candidates. These are filtered based on their size, aspect ratio, and bounding rectangle angle. After that, there are just a few plates left for further analysis, as seen in Figure 21-right.

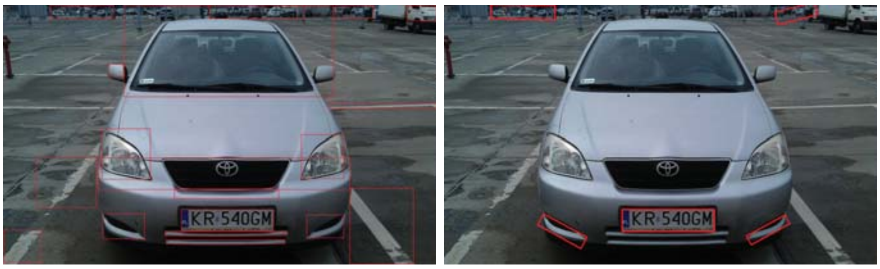


Figure 21. Finding the plate candidates: left: all possible candidates, right: filtered possibilities.

3. **Character separation**
   This stage is where the plate candidate (Fig. 22-left) is split up into individual letters. Firstly, the plate candidate is blurred and normalized (Fig. 22-center). A binary threshold operation is then used to create a black and white image (Fig. 22-right). It is important to clear the image from all the noise that is left before continuing.

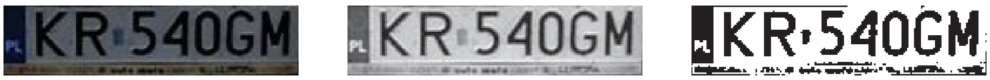


Figure 22. Character segmentation stages: left: original image, center: normalized, right: binarized.

Horizontal and vertical stripes-of-interest are chosen based on statistical analysis. There are only the letters and the hologram sticker left on the picture as seen in Figure 23-left and center. It is now possible to use region separation algorithm to extract the letters (Fig. 23-right). Each character is then moved into its own 2D binary matrix and passed to the classifier.

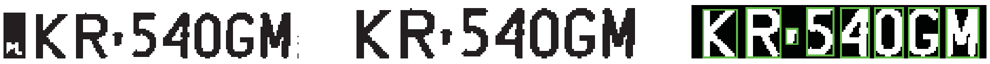


Figure 23. Character segmentation stages: left: horizontal region of interest selected, center: vertical region of interest selected, right: separated characters.

4. **Character recognition**
   A simple kNN algorithm is used. Input gray scale image is resized down to the size of pattern images (5x7 in this example – Figure 24-bottom) and converted into a vector (35 elements in this case). The calculated vector is then compared with each feature vector generated by the external tool (one or more for every character). The calculated fitness value is between 0.5 (worst possible case) up to 1.0 (the vectors are identical). Feature vectors were generated from car plate font samples.

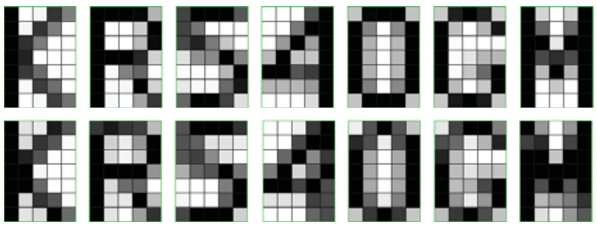


Figure 24. Top: feature vector generated from sample font; bottom: recognized characters downsized to the 5x7 resolution.

5. **Interpreting the results**
   It is possible to calculate a similarity level with plate numbers that are being searched for. A value returned in this manner is a means of letter classification for the best plate – match fitness. The recognized plate number can be shorter or longer than a given match; hence, the need for an algorithm that is capable of comparing them properly, maximizing the result. The proposed solution is rotating one word over another, calculating each time the sum of the fitness levels. The best possible result is then returned, taking into consideration the penalty factor for the characters that were not found.

The implemented plate recognition algorithm provides sufficient quality for the application. Provided that the image contains a clearly-visible car plate, it will find it and build a proper list of character feature vectors. If the plate is on the list of desired plates, it will be located with average match factor between 0.8–0.9.

The performance of the method is also sufficient. Tests on a modern laptop showed that processing of images of about 1 M pixel took 100–250 ms depending on the number of plates in the image. The performance of a mobile device equipped with an ARM Cortex A9 CPU with floating point unit was a little worse; 2–4 frames per second were processed (which is sufficient for the task considered in this paper).

5. Integration and tests

The platform was successfully integrated with ARM-based mobile devices which autonomously executed the car-plates-finding algorithm. Hardware setup consisted of an ARM-based PandaBoard ES 2 device. Images were acquired with an external USB camera. The operating system used was an ArchLinux ARM edition.

The sensor application was built with constant-image processing in mind. There were three main operation threads: heartbeat, task management, and image processing. After startup, the program was controlled only via the platform web page – responded to the user tasks and sent data with appropriate results to the platform database.

The car plate sensor did not use local CouchDB instance. It sent data directly to the main database using remote HTTP requests. The sensor application itself was responsible for holding its own setup data (car plate list). The user could manipulate the data by using suitable tasks in the online platform management system. The list of plates was persistent throughout system's life.

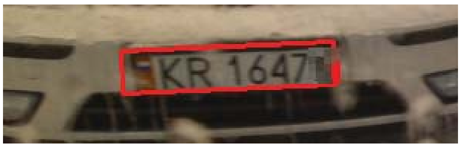
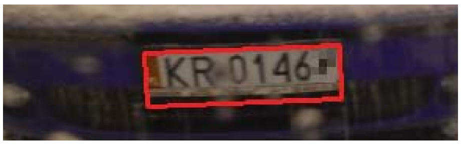
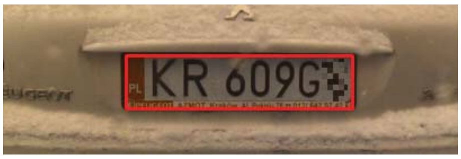
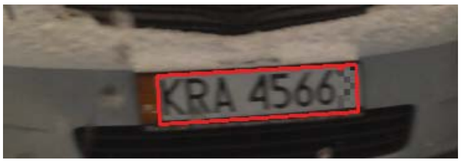
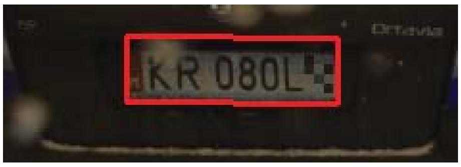
The real-life experiments have been performed twice at the same location – the University parking lot. Both involved a few-minute's drive with a camera installed in a car.

The first time, weather conditions were perfect for the recognition algorithm: full sun and cars that were not dirty. On the second day, conditions were completely different. Heavy snow, dirt on cars, and drops of water on our car windshield caused serious problems with recognition.

In both cases, several cars were chosen in advance – the algorithm was configured to look for them. The experiments showed that, under suitable conditions (the first day), the system performed at a 92% efficiency, while in improper conditions (the second day), only 25–30% of cars were identified. Below, we present several images along with the car plate number matched by the algorithm. The last letter on each plate was removed for privacy reasons.

Table 3

Results of recognition and matching in poor weather conditions.

	KR1647?	81%
	KR0146?	81%
	KR609G?	84%
	2A48	—
	KR380C?	79%

The results in Table 3 show some imperfections of the algorithm. The fourth plate was not recognized at all due to a blurry source image. The last plate was matched incorrectly. Nevertheless, the overall quality of recognition and matching was satisfactory considering the weather conditions.

6. Conclusions

The CouchDB database seems to be a very good basis for large-scale sensor networks using actively-processing mobile devices. The architecture of the Sensor Management

Platform presented in this paper is rather simple; however, the non-functional features of the platform and its performance are surprisingly good.

The recognition algorithm used as a basis for a car-plates-finding sensor is definitely not the best solution in the domain, leaving room for improvement. However, the overall performance of the integrated platform is satisfactory. The image-processing algorithm operates reliably on mobile devices, providing sufficient performance.

Acknowledgements

The research leading to these results has received founding from the AGH grant no. 15.11.120.270.

References

[1] Eurostat. *Crime trends in detail.* `http://epp.eurostat.ec.europa.eu/statistics_explained/index.php/Crime_trends_in_detail`, 2013.

[2] Madden S. R., Franklin M. J., Hellerstein J. M., Hong W.: *TinyDB: an acquisitional query processing system for sensor networks.* Transactions on Database Systems (TODS) – Special Issue: SIGMOD/PODS, 2003.

[3] Mueller R., Alonso G., Kossmann D.: SwissQM: Next Generation Data Processing in Sensor Networks. *Third Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, USA, 2007.

[4] Yick J., Mukherjee B., Ghosal D.: Wireless sensor network survey. *Computer Networks*, 52(12): 2292–2330, 2008.

[5] Leszczuk M., Janowski L., Romaniak P., Głowaczand A., Mirek R.: *Quality Assessment for a Licence Plate Recognition Task Based on a Video Streamed in Limited Networking Conditions.* Multimedia Communications, Services and Security. Communications in Computer and Information Science vol. 149, 2011, pp. 10–18.

[6] Iwanowski M.: Automatic car number plate detection using morphological image processing. *Przeglad Elektrotechniczny.* 81(3): 58–61, 2005.

[7] Ozbay S., Ercelebi E.: Automatic vehicle identification by plate recognition. World Academy of Science, Engineering and Technology, 2005.

[8] Anagnostopoulos C.-N. E., Anagnostopoulos I. E., Psoroulas I. D., Loumos V., Kayafas E.: License Plate Recognition From Still Images and Video Sequences: A Survey. *Intelligent Transportation Systems, IEEE Transactions on.* 9(3): 377–391, 2008.

[9] Niec M., Pikula P., Mamla A., Turek W.: Erlang-Based Sensor Network Management for Heterogeneous Devices. *Computer Science.* 13(3): 139–151, 2012.

[10] Niclausse N., *Tsung Documentation.* `http://tsung.erlang-projects.org/`, 2013.

Affiliations

Łukasz Nocuń
AGH University of Science and Technology, Krakow, Poland

Michał Nieć
Erlang Solutiuons, Krakow, Poland

Paweł Pikuła
AGH University of Science and Technology, Krakow, Poland

Aleksander Mamla
AGH University of Science and Technology, Krakow, Poland

Wojciech Turek
AGH University of Science and Technology, Krakow, Poland, `wojciech.turek@agh.edu.pl`

Received: 05.02.2013
Revised: 11.04.2013
Accepted: 14.06.2013