Building low-cost wireless image sensor networks: from single camera to multi-camera system

Congduc Pham
University of Pau, LIUPPA Laboratory
congduc.pham@univ-pau.fr

Vincent Lecuire
University of Lorraine, CRAN UMR 7039, CNRS
vincent.lecuire@univ-lorraine.fr

ABSTRACT

Wireless Image Sensor Networks (WISN) where sensor nodes are equipped with miniaturized visual CMOS cameras to provide visual information is a promising technology for situation awareness, search&rescue or intrusion detection applications. In this paper, we present an off-the-shelf image sensor based on Arduino boards with a CMOS uCamII camera. The image sensor works with raw 128x128 image, implements an image change detection mechanism based on simple-differencing technique and integrates a packet loss-tolerant image compression technique that can run on very limited memory platforms. We detail the performance and energy consumption measures of the various image platforms and highlight how both medium-end and low-end platforms can be supported. From the single-camera system, we describe the extension to a multi-camera system which provides omnidirectional sensing at a very lost cost for large-scale deployment.

CCS Concepts

•Hardware → Wireless integrated network sensors;

Keywords

Wireless image sensor networks; multi-camera nodes; image processing; resource-constrained design

1. INTRODUCTION

Wireless Image Sensor Networks (WISN) where sensor nodes are equipped with miniaturized visual CMOS cameras to provide visual information is a promising technology for situation-awareness, search&rescue or intrusion detection applications. There are a number of image sensor boards available or proposed by the very active research community on image and visual sensors: Cyclops [1], MeshEyes [2], Citric [3], WiCa [4], SeedEyes [5], Eye-RIS [6], Panoptes [7], CMUcam3&FireFly [8, 9], CMUcam4 and CMUcam5/PIXY [9], iMote2/IMB400 [10], ArduCam [11],... All these platforms and/or products are very good but are mostly based on ad-hoc development of the visual part (i.e. development of a camera board with dedicated micro-controller to perform a number of processing tasks) or are based on very powerful micro-controller/Linux-based platforms or do not have an efficient image encoding and compression scheme adapted to wireless sensor networks. Our motivations in building our own image sensor platform for research on image sensor surveillance applications are:

1. to have an off-the-shelf solution so that anybody can reproduce the hardware and software components: we use an Arduino-based solution for maximum flexibility in programming and design; we use a simple, affordable external camera to get raw image data. We can also easily extend to a multi-camera system,
2. develop and experiment an efficient image compression scheme running on host micro-controller (no additional nor dedicated micro-controller) which addresses the problem of resource limitations of sensor nodes, as memory size, processor speed, battery capacity and low-bandwidth radio, and which produces a packet stream tolerant to packet losses.

The image sensor that we propose works with raw 128x128 8-bbp gray scale image which can be compressed with various quality factors for reducing the bandwidth usage and end-to-end delay of image communication over the multi-hop network path. An image change detection mechanism based on simple-differencing technique shows very good results with negligible processing time. The total time to capture an image, detect changes, encode the image and transmit it can be less than 2.4s at medium quality level on our image sensor platform. One original feature of the present work with respect to previous works about image sensors is the generalization from a single-camera system to a multiple-camera system.

The rest of the article is organized as follows. Section 2 describes the generic image sensor components. Section 3 presents the image processing tasks: the image change detection mechanism and the image encoding technique for transmission on low bandwidth radios. We will describe how these stages have been adapted to run on limited memory platforms. Section 4 presents the performance and energy consumption measures of the image sensor platforms. In Section 5, we present how a multi-camera system can be built from the generic design to provide omnidirectional sensing at a very low cost. We conclude in Section 6.

ICDSC '15, September 08 - 11, 2015, Seville, Spain
© 2015 ACM. ISBN 978-1-4503-3681-9/15/09...$15.00
DOI: http://dx.doi.org/10.1145/2789116.2789118

2. A LOW COST IMAGE SENSOR

We use Arduino boards with the CMOS uCamII camera from 4D systems [12]. The uCamII is shipped with a 56^o angle of view lens but we also use 76^o and 116^o lenses. The uCam is connected to the Arduino board through an UART interface at 115200 bauds. The uCamII is capable of providing both raw and JPEG bit streams but we are not using this last feature as it is impossible from the delivered JPEG bit stream to build a packet stream tolerant to packet losses. As a result, we retrieve raw 128x128 8-bpp grey scale images from the uCamII then we operate image compression on the Arduino board. For comparison purposes, we made two versions of our image sensor: one is based on the Arduino Due board and the other on the Arduino MEGA2560. The Arduino Due is a micro-controller board based on the Atmel SAM3X8E ARM Cortex-M3 running at 84MHz with 96KB of SRAM memory. The MEGA2560 features an ATmega2560 at 16Mhz and has 8KB of SRAM memory. The Arduino Due would represent the medium-end platform while the MEGA2560 is the low-end platform. A short-range IEEE 802.15.4 radio is provided by a Digi XBee S1 module. The XBee module is also connected to the Arduino through an UART line but at 125000 bauds. It is also possible to use a long-range radio technology such as LoRa$^{\mathrm{TM}}$ from Semtech but this issue is out of the scope of this paper. Fig. 1 shows the image sensor.

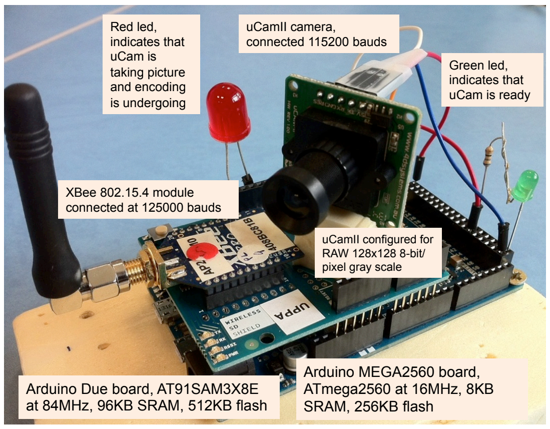
Red led, indicates that uCam is taking picture and encoding is undergoing
uCamII camera, connected 115200 bauds
Green led, indicates that uCam is ready
XBee 802.15.4 module connected at 125000 bauds
uCamII configured for RAW 128x128 8-bit/ pixel gray scale
Arduino Due board, AT91SAM3X8E at 84MHz, 96KB SRAM, 512KB flash
Arduino MEGA2560 board, ATmega2560 at 16MHz, 8KB SRAM, 256KB flash

Figure 1: Image sensor built with Arduino (Due or MEGA) and uCam camera

The Due has enough memory to perform image change detection and image compression tasks. This is not the case on the MEGA2560 where very few memory remains after loading all the libraries required for managing the uCam and the radio communications: only 2000 bytes remain while the raw image already requires 16384 bytes. Therefore, we use an SD card attached to the MEGA SPI bus: when we read the raw data from the serial port, we also write to the SD card in blocks of 1024 bytes. We will show later on the performance measures of the various platforms.

3. IMAGE PROCESSING TASKS

We describe the two image processing tasks implemented on the image sensor. First, the image change detection task, and second, the image compression task for size reduction and significant tolerance to packet losses and drops.

3.1 Image change detection

We implemented an image change detection mechanism based on "simple-differencing" of pixel: each pixel of the image from the uCam is compared to the corresponding pixel of a reference image, taken previously at startup of the image sensor and stored in memory (for the Due) or in a file on the SD card (for the MEGA2560). When the difference between two pixels, in absolute value, is greater than *pixThres* we increase the number of different pixels, *nDiff*. When all the pixels have been compared, if *nDiff* is greater than *nbPixThres* we can assume an image change. However, in order to take into account slight modifications in luminosity due to the camera, when *nDiff* is greater than *nbPixThres* we additionally compute the mean luminosity difference between the captured image and the reference image, noted *lumDiff*. Then we re-compute *nDiff* but using *pixThres*+*lumDiff* as the new threshold. If *nDiff* is still greater than *nbPixThres* we finally conclude for a major image change and trigger the transmission of the image. If no change occurs during 5 minutes, the image sensor takes a new reference image to take into account light condition changes. The image change detection can be used for intrusion detection as tested in figure 2: we set *pixThres* to 35 and *nbPixThres* to 300. In doing so, we were able to systematically detect a single person intrusion at 25m without any false alert. Note that traditional infrared presence sensors (PIR) can not provide detection at that distance. In addition, the image change detection mechanism can be used and tuned to detect changes in close-up views for various surveillance purposes.

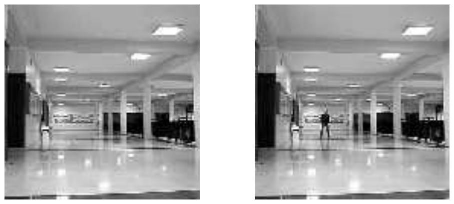

Figure 2: Left: reference image; Right: intruder detected

The "simple-differencing" method is very light-weight because it requires for each pixel of the image only 1 addition (to compute the pixel difference) and 1 comparison (to compare with *pixThres*) and 1 variable incrementation in case the difference is greater than *pixThres*. This process is done on-the-fly while reading data from the uCam. Note that the "simple-differencing" process is performed on the raw image. Once the intrusion is detected, a lower quality version of the image can be transmitted using a lossy compression scheme. Such scheme is useful for achieving high compression ratio of image data at the expense of an eventually significant but generally acceptable degradation of the visual image quality.

3.2 Image compression method

Image compression aims to remove the spatial and spectral redundancies in image data for enabling faster transmission on limited bandwidth radio technologies such as IEEE 802.15.4. Additionally, even if MAC layer retransmission provides quite efficient and low-overhead reliability, packet loss recovery is at the expense of more end-to-end delay and energy consumption. For this reason, the encoded bit stream needs to be tolerant to packet losses.

Our method is based on the JPEG baseline algorithm [13] which is a very popular standard in image compression for either full-color or gray-scale images of natural, real-world scenes. It lies on the transformation–quantization–codeword assignment conventional structure where the transformation stage is based on the 2-D 8-point DCT, i.e., the image is divided into blocks of 8x8 pixels and each block is processed independently. We applied the Cordic Loeffler DCT [14], which is certainly the most efficient fast multiplierless algorithm operating in the 1-D DCT domain. It requires only 38 additions and 16 shifts for an 8-point DCT. All operations are on 16-bit integer. The 8x8-point DCT is obtained by applying first the algorithm over the 8 rows then over the resulting 8 columns, with a cost of 608 additions and 256 shifts. Furthermore, we consider the quantization table and the Golomb and MQ encoding procedure given in the Annexes of the JPEG standard [13]. The quantization stage requires 64 floating or fixed-point multiplications and 64 round-offs operations. The MQ coder is an approximate implementation of arithmetic coding tailored for binary data. The Golomb coding is a lossless compression scheme which encodes the quantized coefficients into a form meaningful for the MQ coder. Using the Golomb and MQ encoding instead of Huffman encoding is significantly profitable from the standpoints of computational complexity and memory requirement. Fig. 3 shows the original raw 128x128 image taken with the image sensor and encoded with various quality factors: Q=90 (high quality), Q=50 (medium quality) and Q=10 (low quality).

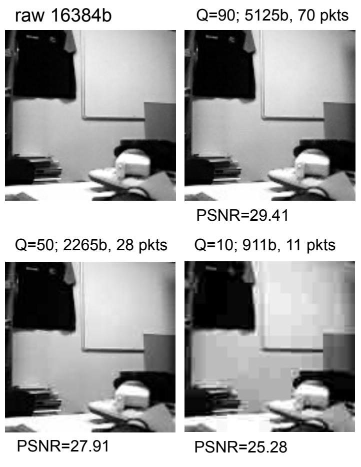

Figure 3: 128x128 image taken by the image sensor, encoded with various quality factor.

The total size of the compressed image, the number of generated packets and the PSNR compared to the original image are shown. We set the maximum image payload per packet to 90 bytes (this is the maximum image payload, in practice, the produced packet size will vary according to the packetization process) because 6 bytes need to be reserved in the 802.15.4 payload for framing bytes (2B), quality factor (1B), real packet size (1B) and offset of the first block of image data in the packet (2B).

The packetization stage consists in filling each packet with an integer number of encoded blocks. This ensures that, at the receiver side, a block is either entirely received or entirely lost. Last but not least, encoded blocks are packetized in a pseudo-random order by applying a spatial interleaver based on the famous Arnold's cat map [15]. In this way, neighbor blocks are put into distinct packets to make easier loss concealment at the receiver side. Fig. 4 shows the impact of packet losses (20%, 40% and 60%) on the image quality.

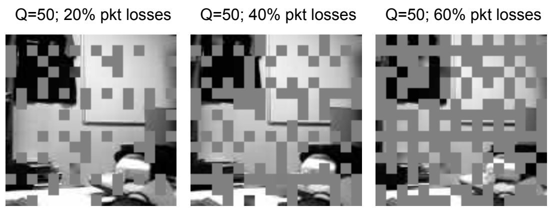

Figure 4: Impact of packet losses on image quality

3.3 Image compression on low-end platforms

The implementation of image compression on limited memory platforms can be challenging. A typical example is illustrated by our MEGA2560 platform which has only about 2KB of SRAM available at run-time to perform image processing tasks and to build data packets for transmission. As a result, a full-frame processing is to be avoided. We use an external Flash (SD card) for storing the raw image data. As standard JPEG encoding involves partitioning of the image into independent blocks, it is possible to encode and packetize on-the-fly the data with low memory usage. Only 4 buffers have to be allocated in SRAM. The first is used to operate the DCT and quantization stages on the current block, 64 integer values stored in Flash in 8-bit format, to produce 16-bit integers. The second is used to store the standard quantization matrix (64 integer values in 8-bit format). The third is used to store the array defining all the possible states for the context of the MQ coder (94x4 values in 16-bit format). The last buffer is used to fill the current packet with data produced by the MQ coder (96 bytes including header and payload fields). Finally, the memory usage required at run-time by this implementation is lower than 1 KB. Compared to the case where the raw image can be entirely stored in SRAM, the processing time is however increased due to the read access time of flash memory.

4. PERFORMANCE OF THE IMAGE SENSOR PLATFORM

4.1 Image processing on Arduino Due

Fig. 5 shows the encoded image size with the compression ratio and the number of produced packets for various quality factors. Column A shows the image encode time which is quite constant. Column B shows the "encode+pkt time" which is the overhead of the image encoding process including the encoding itself and the packetization stage, but without transmission. The time to read the raw image data from the uCam is also shown in column R (1512ms) and it actually does not depend much on the uCam-Arduino connection baud rate (here 115200 bauds) because the limitation is mainly due to memory read operations from the Arduino UART ring buffer. R+B represents the latency between the snapshot taken by the camera and the time all the packets of the encoded image are produced (once again without transmission).

		N	R	A	B	C = D - B	D	E=R+D	F
Quality Factor Q	size in bytes (compression ratio)	number of packets (with MSS=90)	reading time from ucam	encode time	encode + pkt time	transmis-sion time (deduced)	encode + pkt + transmis-sion time	cycle time, with transmis-sion	rcv time at the sink
90	5125 (3.2)	70	1512	512	782	539	1321	2833	799
80	3729 (4.4)	48	1512	511	704	384	1088	2600	599
70	2957 (5.5)	37	1512	519	686	304	990	2502	447
60	2552 (6.4)	32	1512	509	662	263	925	2437	390
50	2265 (7.2)	28	1512	500	646	233	879	2391	349
40	2024 (8.1)	25	1512	516	657	207	864	2376	317
30	1735 (9.5)	21	1512	516	649	177	826	2338	278
20	1366 (12)	17	1512	518	638	140	778	2290	231
10	911 (18)	11	1512	516	628	93	721	2233	177

Figure 5: Cycle time measured on the Due-based platform as a function of the image compression ratio. All times are in ms.

If we take into account the transmission overhead shown in column C, column D shows the "encode+pkt+transmission time". The packetization and the transmission tasks are performed in a row for each packet. Values in column B and column D have been globally measured and can be used to get column C which is the time taken globally for transmitting the produced packets: more packets means higher transmission time. If we use a quality factor of 50, the total time between the snapshot taken by the uCam and the end of the transmission of the image is 1512+879=2391ms. Column E shows the image sensor cycle time with transmission of image packets.

To quantify the cost of the image change detection mechanism, we measured the time to get data from the uCam when the "simple-differencing" method is included and when it is not. We did not observe any difference on the Arduino Due: the time to read data from the uCam and perform the pixel comparison is still 1512ms. Additionally, the image luminosity computations takes 1ms and recomputing *nDiff* taking into account *lumDiff*, in case *nDiff* is greater than *nbPixThres*, takes 17ms. It means that an intrusion detection mechanism can be realized at the maximum rate of one every 1512+18=1530ms.

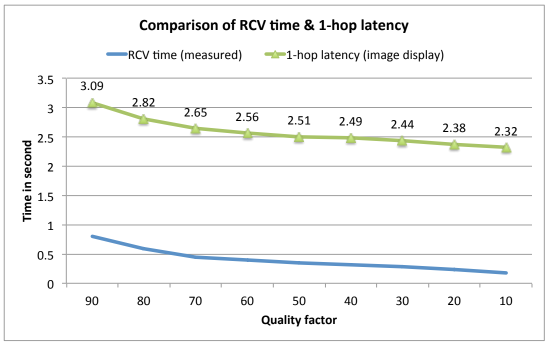

Figure 6: 1-hop image display latency

Column F shows the receive time measured at a sink which will decode and display the image. The receive time represents the elapsed time between the first packet received and the last packet received. Fig. 6 then shows the 1-hop image display latency which is the elapsed time between the snapshot at the source image sensor and the display of the image at the sink 1-hop away (column R+B+F). The smaller the latency, the more responsive is the system. Since the time to transmit a packet is not very large, we can actually see that having Q up to 70 is not very penalizing in term of receive latency.

4.2 Image processing on Arduino MEGA2560

The encoding time and the transmission time may depend on the micro-controller type and speed. Regarding the transmission time, we use a traffic generator to measure the minimum time spent in the send function and the minimum time between 2 sends as the payload size is varied. 100 packets are sent and the mean is computed. We compared the Arduino Due to the Arduino MEGA and the code of the traffic generator, as well as the communication library for the XBee radio module, is exactly the same on both platforms. We found the Arduino MEGA and the Arduino Due have quite close transmission overheads: mean minimum time between 2 sends (for a 100-byte payload) are 9.2ms (Due) and 9.6ms (MEGA); time in send are 9.18ms (Due) and 9.07 (MEGA). Therefore the impact of the slower micro-controller is not high on the performances of the communication stack.

For the encoding time, we run the modified image encoding code on the Arduino MEGA without transmitting packets. Fig. 7 shows the encoding timing when the quality factor is varied. Column A, A1 and A2 are respectively the cumulated encoding time, the cumulated SD card reading time and the cumulated packetization time. Column B shows the global processing time measured globally, included cost of control loops, function calls and other pieces of glue code. This is why B is a bit greater than A+A1+A2 (between 12ms and 28ms).

	N	R	A	A1	A2	B	C = D - B	D	E=R+D
size in bytes	number of packets (with MSS=90)	reading time from ucam	cumulated encode time	cumulated SD time	cumulated pkt time	encode + SD + pkt time	transmis-sion time (deduced)	encode + SD + pkt + transmis-sion time	cycle time, with transmis-sion
5125	70	1515	879	884	1572	3363	542	3905	5420
3729	48	1515	869	871	941	2704	386	3090	4605
2957	37	1515	878	877	713	2493	308	2801	4316
2552	32	1515	868	892	589	2371	266	2637	4152
2265	28	1515	865	883	537	2305	235	2540	4055
2024	25	1515	871	871	467	2226	210	2436	3951
1735	21	1515	871	878	405	2171	179	2350	3865
1366	17	1515	871	881	325	2093	142	2235	3750
911	11	1515	854	886	251	2003	96	2099	3614

Figure 7: Cycle time measured on the MEGA-based platform as a function of the image compression ratio. All times are in ms.

We can see that the encoding process (column B) takes a bit more than 3.5 times more on the MEGA than on the Due, e.g. about 2305ms instead of 646ms for Q=50. The time to read data from the uCam is still close to the 1.512s found on the Due. Similarly to the Due case, column D shows the "encode+SD+pkt+transmission time" and column E shows the cycle duration with transmission of image packets. The 1-hop image display latency with the MEGA can be obtained by adding about 1.6s to the Due's 1-hop image display latency shown previously in Fig. 6.

For the image detection mechanism on the Arduino MEGA, the data from the uCamII are compared to the reference image data stored in a file on the SD card: blocks of 512 bytes are read from the file and compared to the corresponding image pixels from the uCam. As for the Due, we did not observe any significant difference when introducing the "simple-differencing" comparison and reading the SD card for the reference image.

4.3 Energy consumption measures

To make the energy consumption measures we inserted additional power consumption by toggling a led to better identify the various phases of the image sensor operations. For all the energy tests, the image transmitted was encoded using a quality factor of 50 and between 45 and 49 packets were produced at the packetization stage. The objective here is not to have a complete energy map with varying quality factors and packet number, but to have an approximate idea of the energy consumption on both platforms. Fig. 8 shows an entire cycle of camera sync, camera config, data read, data encode and packetization with transmission on the Due. We can compute the baseline energy consumption of the Due once the camera has turned to sleep mode (this happen after 15s of being idle. We waited long enough before starting the energy measure process). We measured this consumption at 1.39J/s. Note that we did not implement any advanced power saving mechanisms such as putting the micro-controller in deep sleep mode or lower frequency, or performing ADC reduction, nor powering off the radio module. It is expected that the baseline consumption can be further decreased with more advanced power management policy.

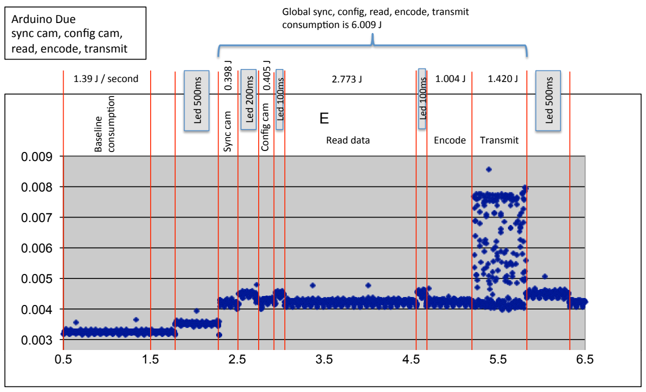

Figure 8: Energy consumption for the Arduino Due

After removing the energy consumed by the led, we found that an entire cycle for image acquisition, encoding and transmission consumes about 6J. The largest consumed energy part on the Due comes from polling the serial line to get the image data from the uCam (through the system serial buffer). The encoding process actually consumes less than half that amount of energy. The cost of periodic image change detection, without encoding and transmission is similar to the "Read data" cost. Therefore, we found that the "Global sync, config, read&compare" consumption is 3.571J.

On the MEGA board, the baseline consumption was found at 1.25J, a bit smaller than on the Due. However, the MEGA board consumes much more than the Due for all operations. This is mainly due to its much slower clock frequency making all the processes to take longer time. The need of an external storage such as an SD card also contributes to higher energy consumption. This energy consumption statement is actually quite surprising for us because we thought that the Due board would consume much more energy than the MEGA. Given the price of the Due compared to the MEGA, building the image sensor with the Due seems to be the best choice both in terms of performances and energy efficiency.

Fig. 9 summarizes and compares the Due and MEGA platforms. In the autonomy category, the uptime is computed with the baseline consumption. Once again, no power saving mechanisms have been implemented yet.

	Consumption			Autonomy		
Platform	Baseline J/s	cycle J	intrusion J	Uptime (hour)	# cycle	#intrusion
Due	1.394	6.009	3.571	7.75	6470	10887
MEGA2560	1.251	10.472	4.418	8.63	3713	8799

Figure 9: Energy consumption summary

♯cycle represents the number of image capture, encoding and transmission cycles that can be performed. Similarly, ♯intrusion represents the number of image change detection (but no encoding nor transmission) that can be performed. These values are obtained by taking the energy amount of a 1200mAh 9V battery, i.e. 38880J.

5. BUILDING MULTI-CAMERA SYSTEMS

From the 1-camera system it is not difficult to have a multiple camera system. Both Arduino Due and MEGA2560 have 4 UART ports. One port is used for connection to the XBee module, so the 3 others are available for 3 uCamII cameras. On initialization, the image sensor board will take and store a reference image for each camera (raw format). On the Arduino Due, there is enough memory to store a reference image for each camera. On the MEGA2650, these reference images are stored in the SD card. Figure 10(left) shows our Arduino Due connected to 3 uCamII cameras. The cameras are set at 120^o from each other and are activated in a round robin manner. The image change detection process is then done on each camera with the corresponding reference image. Note that when images need to be transmitted (upon intrusion for instance), each camera can be configured with a different image quality factor if necessary. 76^o and 116^o lenses can be mounted on the uCamII, in addition to the 56^o lens shipped with the uCamII. Figure 10(right) compares the FoV of the 3 lenses.

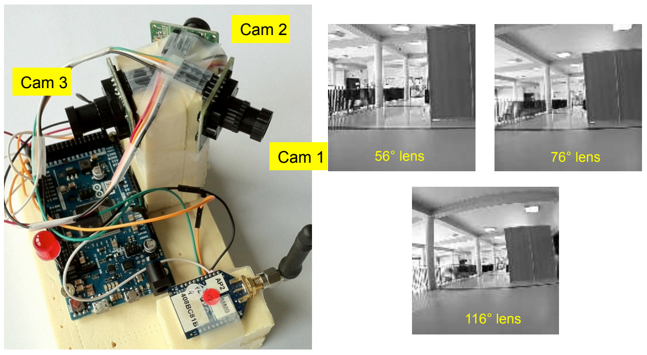

Figure 10: A 3-camera system on the Arduino Due

For large-scale deployment, such as in intrusion detection applications, Figure 11 compares the coverage of a 80 x 1-uCamII system (top-left, 36.3%) to a 80 x 3-uCamII system (top-right, 71.5%) and to a 240 x 1-uCamII system (bottom-left, 71.2%). The image sensors are randomly deployed in an 400mx400m area. The depth of view of the cameras has been set to 35m and lenses are 76^o. The FoV in red is the one of camera 0, for both 1-camera and 3-camera systems.

The blue is for camera 1 and the green for camera 2, in the 3-camera system. We can see that the coverage is greatly improved, at a much lower cost than having 3 times more full sensor boards. With 116^o lenses, using 3 cameras can almost provide omnidirectional vision as shown in Fig. 11(bottom-right) (91.6% of coverage).

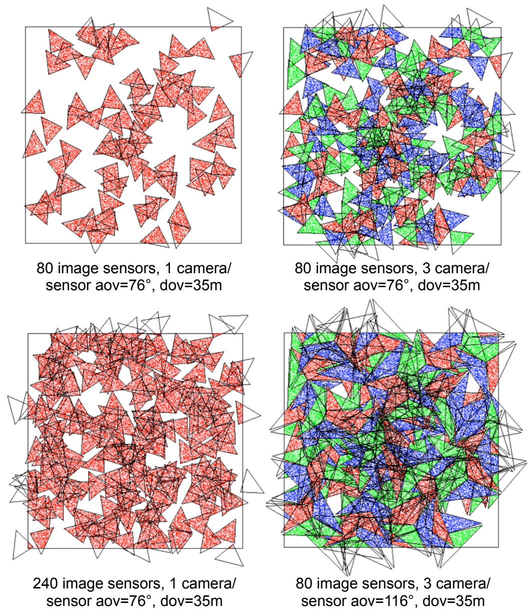
80 image sensors, 1 camera/
sensor aov=76°, dov=35m
80 image sensors, 3 camera/
sensor aov=76°, dov=35m
240 image sensors, 1 camera/
sensor aov=76°, dov=35m
80 image sensors, 3 camera/
sensor aov=116°, dov=35m

Figure 11: Comparison of coverage by various image sensor systems

Once programmed, the image sensor can be completely autonomous: on startup a first image for each camera will be taken to serve as the reference image, then periodic image change detection will send images to the sink. At the sink (a Linux computer), a display program will continuously waits for image packets from the radio interface and will display the received image from different sensors. We deployed one 3-camera system and three 1-camera system in a hall and tested it during several hours without any false alarms. The sink has an internet access and use Dropbox to sync the received image folder with remote devices such as smartphones.

6. CONCLUSIONS

We integrated an image change detection method and a packet loss-tolerant image compression method technique that can run on very limited memory platforms. Encoded images can be transmitted with low-bandwidth radio such as IEEE 802.15.4. The latency between the image snapshot and the end of image transmission can be less than 2.4s at medium image quality factor. For intrusion detection applications, we showed that multi-camera systems are attractive because the number of nodes required for achieving the full coverage of the area under surveillance is largely lower than in the case of single-camera systems. In future works, we will add energy saving techniques for the Arduino board and the radio module for usage in domestic or industrial Internet of Things applications. A promising direction is also the usage of long-range 1-hop radio technology to reduce the complexity of visual surveillance application deployment.

7. REFERENCES

[1] M. Rahimi et al. “Cyclops: In situ image sensing and interpretation in wireless sensor networks,” in *ACM SenSys*, 2005.

[2] S. Hengstler, D. Prashanth, S. Fong, and H. Aghajan, “Mesheye: A hybrid-resolution smart camera mote for applications in distributed intelligent surveillance,” in *IPSN*, April 2007.

[3] P. Chen et al., “Citric: A low-bandwidth wireless camera network platform,” in *ACM/IEEE ICDSC*, Sept 2008.

[4] R. Kleihorst, A. Abbo, B. Schueler, and A. Danilin, “Camera mote with a high-performance parallel processor for real-time frame-based video processing,” in *IEEE AVSS*, Sept 2007.

[5] Evidence Embedding Technology, “Seed-eye board, a multimedia wsn device. http://rtn.sssup.it/index.php/hardware/seed-eye,” accessed 20/12/2013.

[6] Á. Rodríguez-Vázquez et al., “The Eye-RIS cmos vision system,” in *Analog Circuit Design*, Springer Netherlands, 2008.

[7] W.-C. Feng et al., “Panoptes: Scalable low-power video sensor networking technologies,” *ACM TOMCCAP*, vol. 1(2), May 2005.

[8] A. Rowe, D. Goel, and R. Rajkumar, “Firefly mosaic: A vision-enabled wireless sensor networking system,” in *IEEE RTSS*, Dec 2007.

[9] Evidence Embedding Technology, “cmuCam: open source programmable embedded color vision sensors. http://www.cmucam.org/,” accessed 19/12/2014.

[10] S. Paniga, L. Borsani, A. Redondi, M. Tagliasacchi, and M. Cesana, “Experimental evaluation of a video streaming system for wireless multimedia sensor networks,” in *IEEE/IFIP Med-Hoc-Net*, 2011.

[11] ArduCam, “Arducam. http://www.arducam.com,” accessed 19/12/2014.

[12] 4D Systems, “uCamII. http://www.4dsystems.com.au/product/ucam_ii/,” accessed 19/12/2014.

[13] International Standard Organization, “ITU-T recommendation T.81. http://www.jpeg.org/jpeg/,” accessed 6/2/2015.

[14] B. Heyne, C. C. Sun, J. Goetze and S. J. Ruan, “A computationally efficient high-quality Cordic based DCT,” in *EUSIPCO*, Sep 2006.

[15] C. Duran-Faundez and V. Lecuire, “Error resilient image communication with chaotic pixel interleaving for wireless camera sensors,” in *ACM Workshop on Real-World Wireless Sensor Networks*, 2008.