A testbed for distributed detection of keypoints and extraction of descriptors for the Speeded-Up-Robust-Features (SURF) algorithm

ANDREAS BRYKT

Master's Degree Project
Stockholm, Sweden December 2013

XR-EE-LCN 2013:024
A testbed for distributed detection of keypoints and extraction of descriptors for the Speeded-Up-Robust-Features (SURF) algorithm

ANDREAS BRYKT

Stockholm December 18, 2013

Laboratory for Communication Networks
School of Electrical Engineering
Kungliga Tekniska Högskolan
Abstract

Detecting keypoints and computing descriptors needed in an image recognition algorithm are tasks that require substantial processing power if they are to be executed in a short time span. If a network of sensor nodes is used to capture the images to be processed, then the sensor nodes could be used to perform the actual processing. The system would distribute the computing tasks to the available nodes in the network, so that the computing load can be divided among the nodes. By this, the computing time could possibly still be kept low, despite the large difference in available computing power between a rack-server and a sensor node.

This report describes the implementation of a testbed for the evaluation of distributed processing of visual features. The testbed is implemented in C++ using creditcard sized computers and Zigbee USB units. Communication between nodes utilizes ASN.1 defined types. The detection and extraction stage use an implementation of the SURF algorithm from OpenCV. Results are sent for matching to a server using a TCP-socket in the sink node. The system is evaluated in terms of data transmission protocol efficiency, and time spent on transmitting data vs. computation.
Contents

1 Introduction ......................................................... 4
  1.1 Methodology .................................................. 4
  1.2 Report structure ............................................ 5

2 Background ........................................................ 5
  2.1 Computer vision .............................................. 5
  2.2 Use cases .................................................... 5
  2.3 Keypoints, descriptors and the SURF algorithm ............ 6
    2.3.1 Integral images ........................................ 6
    2.3.2 Keypoints .............................................. 6
    2.3.3 Descriptors ............................................ 9
  2.4 OpenCV ....................................................... 11
  2.5 Zigbee ........................................................ 11
    2.5.1 libxbee ............................................... 14
    2.5.2 Automatic Repeat-reQuest .............................. 14
  2.6 ASN.1 ........................................................ 15
    2.6.1 Basic encoding rules (BER) ............................ 15
    2.6.2 Distinguished encoding rules ......................... 16
    2.6.3 Asn1.c compiler ....................................... 16
  2.7 Hardware Components ........................................ 16
    2.7.1 Beaglebone black ...................................... 16
    2.7.2 XStick ................................................ 17

3 Design space .................................................... 18
  3.1 Key functionality ........................................... 18
    3.1.1 Area-split and scale-split ................................ 18
    3.1.2 Processing in camera node ................................ 19
    3.1.3 Separation of control and data ......................... 20
    3.1.4 Dynamic configuration of the off-loading scheme from the server node ........................................... 20

4 System Design ................................................... 21
  4.1 Testbed workflow ............................................ 21
  4.2 Class layout ................................................. 22
    4.2.1 NodeManager .......................................... 23
    4.2.2 Connection ............................................ 23
    4.2.3 Receiver .............................................. 24
    4.2.4 ReceiveItem ........................................... 27
    4.2.5 DfeImage .............................................. 28
    4.2.6 ImgTask .............................................. 28
    4.2.7 ImageManager ......................................... 28
    4.2.8 TaskCreator .......................................... 30
    4.2.9 TaskManager .......................................... 30
    4.2.10 TaskExecutive ........................................ 33
    4.2.11 CompletedTasks ....................................... 35
    4.2.12 Encoder .............................................. 35
    4.2.13 Sender ............................................... 36
    4.2.14 Decoder .............................................. 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.15</td>
<td>TcpClient, TcpSender, TcpReceiver</td>
<td>36</td>
</tr>
<tr>
<td>4.2.16</td>
<td>DfeSettings</td>
<td>38</td>
</tr>
<tr>
<td>4.2.17</td>
<td>ThreadedQueue</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>ASN.1-types</td>
<td>40</td>
</tr>
<tr>
<td>4.4</td>
<td>Overhead in packets</td>
<td>40</td>
</tr>
<tr>
<td>4.5</td>
<td>Maximum supported data size</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>Experimental results</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>802.15.4 Data rate measurement</td>
<td>42</td>
</tr>
<tr>
<td>5.2</td>
<td>Distributed keypoint detection and descriptor extraction performance</td>
<td>43</td>
</tr>
<tr>
<td>5.3</td>
<td>Unit testing</td>
<td>45</td>
</tr>
<tr>
<td>A</td>
<td>ASN.1 types</td>
<td>49</td>
</tr>
<tr>
<td>B</td>
<td>XStick configuration</td>
<td>52</td>
</tr>
</tbody>
</table>
1 Introduction

Computer vision is used today in various situations. When an area needs to be monitored for undefined events that cannot be measured with electro-mechanical sensors, computer vision is a technology that can detect a wide range of incidents. By extending the objects included in the model, the detection range can easily be extended. For example, when an animal community is to be monitored for different events, and it is uncertain in what place this event is going to happen, then computer vision can be used to detect the event. If the area to be monitored is large, a network of cameras can be used to capture images or video from multiple places [1]. Another example is when an area is to be monitored for trespassing, and conventional detection methods cannot be used due to the fact that certain creatures are allowed to pass. Examples of usage can also be found in a home or industrial environment, such as smart meeting rooms or automated production [2] [3] [4].

A sensor node is a node that is able to perform some processing, with low complexity and energy consumption. A system of such sensor nodes, that captures and process images from multiple locations, is often termed a Visual Sensor Network (VSN). Such a network has recently been subject of extensive research. The device that captures images or video in the network is often a low-cost device termed a sensor node. A node can be configured with a wireless transmitter to communicate with other sensor nodes, in some cases over a distance of many kilometres [5]. Sensor nodes have a power consumption in the range of 10 mA, and can in sleep mode stay powered for a year [5]. The sensor nodes has the ability to perform some processing, the typical processing power is in the range of 10 MHz [6] [7]. By using the network of nodes as a distributed computing system, one could eliminate the need for a centralized server, and possibly, with decreased processing time.

The goal of this thesis is to develop a testbed for such a system. The computing tasks that are to be distributed are common operations used in computer vision. The system shall use existing system components and software. The system will be evaluated in terms of transmission and computing efficiency.

1.1 Methodology

Prior to implementing the testbed, a literature study was performed. The literature study included network topics, the Zigbee protocol and different Xbee-hardware communication libraries. Some operating systems were considered and tested. A study of the ASN.1-syntax and encoding rules was carried out, together with a design of the different types needed to be defined with this syntax. Software development topics like multi-threading and cross compiling were also considered, together with topics like build systems, testing frameworks and applications for memory allocation analysis. After the literature study a specification of included features was agreed upon. Then a first design of the testbed was made, followed by a first implementation. After that followed some iterations of redesign of selected parts, implementation of the new design and debugging. At last an evaluation of the testbed was performed. The evaluation consisted of measuring how much time is spent in the different parts of the testbed when an image is processed.
1.2 Report structure

The rest of the report is laid out in the following way. In the next chapter a background to the problem is given. Computer vision is described together with some use cases for a visual sensor network with distributed computing capacities. Different concepts like descriptors and keypoints are described, together with the algorithm that use these concepts. The implementation of the algorithm to be distributed in the testbed is described. In the same chapter the Zigbee protocol is outlined together with the key concepts of the ASN.1-syntax and decoding method. The last part of the next chapter describes the different hardware components used in the testbed. Chapter 3 then describes the functionality that is included in the testbed. Areas like computing task creation and different processing schemes are discussed. Chapter 4 describes the actual software implementation. Here, the different classes are described. This chapter also includes a description and analysis of the application layer protocol defined with the ASN.1 syntax. The last chapter includes result from the evaluation of the testbed.

2 Background

2.1 Computer vision

Visual features are often used in computer vision applications for image retrieval and object recognition. Visual feature extraction is the process of identifying important areas in an image and then describing those areas with mathematical tools. The identified objects in the image are described with a mathematical model, and the models are then compared to models in a library of existing models, available in a database of the recognition system. The comparison is done with a matching algorithm that tries to find the objects in the database that match the object found in the image. Computer vision is used in many areas, from robot navigation to gesture recognition in digital cameras. If computer vision is performed in consecutive frames from a video, it can be used to track the movement of objects in the sequence of images.

2.2 Use cases

A distributed computing system for computer vision has many applications. The general case is when the computing power of each node is insufficient for the application, or when the computing power could be increased by utilizing available nodes in a network. Another case is when one would like to have a system that is not dependent on a central server. The cost is kept down if different components can be excluded. And, the complexity added with central server can be avoided if the nodes can perform the processing.

One example of when such a system is favourable is a monitored traffic intersection. Each entry to the section could be monitored by one node connected to a camera. The cameras should capture vehicles in the intersection, and the system should then detect if the vehicle is a public transport bus. If a public transport bus is approaching, then the traffic lights should turn green so that the bus can pass without stopping. When an object is detected in one entrance
to the intersection, the node will capture an image from the camera. The different computing tasks for the image are then distributed to other nodes available in the network, hence the speed of the recognition process can be increased. The results from the different nodes are then gathered in one of the nodes. The computed model is then used to match the vehicle against known models of different vehicles. If the model matches a public transport bus, then the traffic lights are switched.

2.3 Keypoints, descriptors and the SURF algorithm

A **keypoint** is a point in an image that defines a point detected because of the pixel(s) value, and the pixel values compared to the surrounding pixel values. For each keypoint a **descriptor** is computed. The descriptor describes the area surrounding the keypoint. The algorithm used in this testbed is the Speeded Up Robust Features (SURF) algorithm. The SURF algorithm operates on greyscale images [9]. There exists other algorithms to detect keypoints and to extract feature descriptors, such as Binary Robust Invariant Scalable Keypoints (BRISK) and Fast Retina Keypoint (FREAK).

2.3.1 Integral images

The filters used for identifying SURF interest points typically rely on the sum of pixel values in rectangular areas. To compute the sum of the pixels, one could simply add all the pixel values in the region. If many regions should be summed, one could first compute the integral image, and then use the integral image to compute the sums. An integral image is, similar to the one-dimensional case, the sum of all pixels from the origin to each index. If the integral image is denoted $I(x, y)$, and the pixel value is denoted $F(x, y)$, then the integral image is

$$I(x, y) = \sum_{a=0}^{x} \sum_{b=0}^{y} F(a, b)$$

With the integral image available, any sum over a region can be computed as a sum of just four values.

2.3.2 Keypoints

To detect keypoints, a 2-dimensional filter is used. The filters used are various second derivatives of a gaussian $g(\sigma)$ [9, 10]. Three filters are used; $\frac{\partial^2 g(\sigma)}{\partial x^2}$ is shown in figure 1, $\frac{\partial^2 g(\sigma)}{\partial y^2}$ is shown in figure 2, and $\frac{\partial^2 g(\sigma)}{\partial x \partial y}$ is shown in figure 3.

The filters shown in figure 1, figure 2, figure 3 are then approximated by box-filters, where the filter that approximates the second order derivative of the gaussian curve only can take on two values; 1 and -1. By this, the computation of the multiplication and sum over the image and the box-filter can be carried out as a sum of all the pixels that have a 1 in the corresponding place in the box filter, and subtracting by the pixels that have a -1 in the corresponding place. The main advantage of this is that the sum of all pixels within a rectangular area can be computed as a sum of only four values, if the integral of the image has been computed beforehand, see section 2.3.1. A Hessian matrix is then
Figure 1: The second derivative of a gaussian curve multiplied with an image tries to detect differences along the x-axis.

Figure 2: The second derivative of a Gaussian curve multiplied with an image tries to detect differences along the y-axis.
formed with the result of the inner product of the image and the second order derivatives. The result of the detection is then computed as the determinant of the Hessian. This is termed the response value. A pixel is identified as a keypoint if the determinant is above some threshold value. The threshold value is termed Hessian threshold.

The above method is carried out with different values of $\sigma$, which results in different filter sizes. The filter sizes used are determined by two parameters; the number of octaves, and the number of octave layers. The number of octaves sets the number of levels in the Gaussian pyramid of filters, and the number of octave layers sets the number of filters within each layer. Keypoints found with a certain filter size are said to be found at a certain scale. A 9 x 9 filter is the filter with lowest scale, which corresponds to $\sigma = 1.2$ [9].

An object in the image can give rise to many keypoints with a response above the threshold. In order to not clutter the set of keypoints detected with keypoints that arise from the same object, a non-maximum suppression is performed [9]. A non-maximum suppression is basically carried out by comparing the response values of adjacent filters in the same layer and response values of filters in adjacent octaves. This is described thoroughly in [11]. To perform the non-maximum suppression, two additional filter sizes are used in each octave. One in the upper limit in the octave range, and one in the lower limit. These filters are used only for comparison, no keypoints can be found at their scale.

With the number of octaves defined as $N_O$ and the number of octave layers defined as $N_L$, the filter size in layer $N_p$ is computed as:

$$dG_2 \frac{\partial^2}{\partial x \partial y}$$

Figure 3: The second derivative of a Gaussian curve multiplied with an image tries to detect differences along the diagonal-axis.
\[ N_p = 9 \cdot 2^{N_o - 1} + 6 \cdot 2^{N_o - 1} \cdot (N_L - 1) \] (2)

With the number of octaves set to 4, and the number of octave layers set to 2, the filter sizes used are shown in table 1.

<table>
<thead>
<tr>
<th>Layer 1</th>
<th>Layer 2</th>
<th>Layer 3</th>
<th>Layer 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Octave 1</td>
<td>9</td>
<td>15</td>
<td>21</td>
</tr>
<tr>
<td>Octave 2</td>
<td>18</td>
<td>30</td>
<td>42</td>
</tr>
<tr>
<td>Octave 3</td>
<td>36</td>
<td>60</td>
<td>84</td>
</tr>
<tr>
<td>Octave 4</td>
<td>72</td>
<td>120</td>
<td>168</td>
</tr>
</tbody>
</table>

For more details, see [9].

The image is also up-scaled and interpolation between adjacent filters is carried out to increase the number of available filter results.

Each detected keypoint has the following parameters:

- **x**: The x coordinate of the keypoint.
- **y**: The y coordinate of the keypoint.
- **scale**: The size of the interest area around the keypoint.
- **octave**: The octave where the keypoint was found.
- **response**: A measure of the difference between the pixels in the keypoint area. In other words; how distinct the keypoint is. See section 2.3.2 for a definition.
- **angle**: The orientation of the keypoint. This functionality increases computation time significantly, and can therefore be disabled.

A visualization of the keypoints found for an image is shown in figure 4.

Each keypoint is marked by a circle. The diameter of the circle is proportional to the scale of the keypoint. Each keypoint also has a line from the center to a point on the circle. The angle of this line is proportional to the angle of the orientation of the keypoint. One can for example see that the last letter ‘I’ in the text is aligned with the angle of the keypoint found in it’s vicinity.

### 2.3.3 Descriptors

For each keypoint a 64 element long vector, termed descriptor, is computed. The descriptor describes the area surrounding the keypoint. The computation of the descriptors is done in a quadratic area aligned with the angle of the keypoint. The size of the area is larger for keypoints detected at a larger scale.

The area around a keypoint used to compute descriptors is dependent on the scale of the keypoint. The width of the area \( d \) is

\[ d = \sqrt{2} \cdot 20 \cdot \sigma \] (3)

where \( \sigma \) is the scale of the keypoint. This area is then divided into 16 sub-areas, laid out in a 4 x 4 grid.
Figure 4: Keypoints with various sizes and angles. The Hessian threshold is set to 1000 to keep the number of keypoints found low.

Each sub-area is then divided in a 5 x 5 grid. For each rectangle in the grid, the response of the Haar-wavelet in the x-direction $dx$, and the response of the Haar-wavelet in the y-direction $dy$, is computed. The responses are then weighted with a gaussian centered around the keypoint.

For each sub-area, the following is computed:

- The sum of the weighted Haar-responses in the x-direction.
  \[
  \sum dx \tag{4}
  \]

- The sum of the weighted Haar-responses in the y-direction
  \[
  \sum dy \tag{5}
  \]

- The sum of the absolute values of the weighted Haar-responses in the x-direction
  \[
  \sum |dx| \tag{6}
  \]
• The sum of the absolute values of the weighted Haar-responses in the y-direction

\[ \sum |dy| \]  

(7)

The values computed by 4, 5, 6 and 7 form a four element long vector for each of the 16 sub-areas. This results in a 64 element long descriptor for the keypoint. Each element contains a 32 bit floating point number.

Shown in figure 4 is a visualization of the values of the matrix constructed from the descriptors for the set of keypoints detected. Each row contains 64 pixels and there are 106 rows, one for each of the 106 keypoints found (shown in figure 4). A vector with 128 32 bit floating point values can be computed in an extended mode of the descriptor computation.

2.4 OpenCv

OpenCV is an open source computer vision library, most parts of it are free for both academic and commercial use. It has a C++ interface and supports Windows, Linux, Mac OS, iOS and Android. There is a large community of users, with more than 47 thousand members and the estimated number of downloads exceeds 6 million [12]. There is a large number of books available about OpenCv. OpenCv has its own implementation of the SURF algorithm but it is not part of the free algorithms, and hence, cannot be used for commercial applications without permission. This work is however purely academic and can therefore use the OpenCv implementation without restrictions.

The OpenCv implementation for SURF provides the class SURF, which has methods for detecting keypoints and computing descriptors. The keypoints are stored in the OpenCv class KeyPoint, and the collection of keypoints is passed to the different methods in the STL class vector. The descriptors are stored in the OpenCV class Mat, which basically is a matrix class that implements common matrix operations.

2.5 Zigbee

Zigbee is a protocol that operates in the network layer and in the application framework layer. It is suitable for communication in a wireless personal area network (WPAN), where the devices have low power consumption, low duty cycles and low data rate requirements. It can be used to connect various devices
in a home environment, such as light switches, alarm devices, locks and cameras. It is also suitable for automation in an industrial environment.

One of the key concepts in a WPAN is the ability for devices to connect and join the network temporarily, and then leave the network at any point. Zigbee supports this and devices can join a network in 30 ms. Zigbee communicates at 2.4 GHz and requires a 3 MHz interference free bandwidth at that frequency. Zigbee supports addressing with 16 or 64 bits [13]. The channel bands operate in 5 MHz wide bands from 2.405 GHz to 2.480 GHz. Data rate is supported up to 250 Kbps and the protocol can operate on 16 different channels [14][15]. Zigbee incorporates security built on the 128-bit AES algorithm [15].

Zigbee is built on the IEEE 802.15.4 protocol, and an overview of the dif-

Figure 6: Descriptors for the keypoints in figure 4. The values are scaled so that the largest value is completely white. The image is 64 x 105 pixels. Each row represents the descriptor for one keypoint.
The IEEE 802.15.4 protocol is a packet based radio protocol [14]. The physical layer is responsible for modulation of the carrier wave that makes it possible for the information to be transmitted through the air. The 802.15.4 protocol uses offset quadrature phase shift keying (O-QPSK) as modulation technique. The physical layer is also responsible for enabling/disabling the radio transceiver, link quality indication (LQI) for received packets, energy detection (ED) on the current channel and clear channel assessment (CCA) [14].

The MAC layer is responsible for correct operation over a shared medium. The MAC layer tries to avoid collision when the medium is occupied by multiple devices transferring at the same time. The protocol used in 802.15.4 is Carrier sense multiple access with collision avoidance (CSMA-CA). The MAC layer also guarantees time slots for a device to transfer. [14]. Additionally, the MAC layer is responsible for sending acknowledgement packets. The functionality to send acknowledgement packets can be switched off.

The network layer provides mechanisms for nodes to join a network. It also has functionality to discover new nodes that intend to join the network, together with functionality to discover a certain node that the network is expecting to join. The network layer also provides mechanisms to transfer data in a secure way. The network layer provides tables that let one node communicate with another node even if the node is not within reach. This functionality is termed message forwarding. The communication via message forwarding can occur by keeping tables of how a node can be reached via other nodes.

The application layer provides the possibility to send messages to groups of nodes. Furthermore, the application layer can map between different addressing methods [14][15]. Additionally, the application framework layer provides mapping between different manufacturers services. The interested reader can refer to [14] for more information.

The 802.15.4 protocol has support for unicast and broadcast messages. The 802.15.4 protocol does not have any functionality for sending acknowledgement.
packets for broadcast messages.

2.5.1 libxbee

Libxbee is an open source C/C++ library to aid the use of Digi's XBee radios running in API mode. Libxbee supports various Zigbee radios from Digi. Below, the main classes used from the library are described.

- **libxbee::XBee(std::string mode, std::string device, int baudrate)**
  This class opens the hardware unit. The `XBee` constructor takes parameters to specify the mode, the device and the baudrate to be used. The device is specified as the serial port which the node is connected to. The `XBee` instance can be created in different modes. The mode of the XStick is "xbee1". Digi offers other Zigbee devices which also could be used with libxbee and some of them use different modes.

- **libxbee::ConCallback(XBee &parent, std::string type, struct xbee_conAddress *address)**
  This class is a super class, which takes a reference to the `XBee` instance used for communication, the type of the connection and the address of the remote xbee unit to create the connection to. The address is specified with a pointer to a `xbee_conAddress` struct. The different types of connections used in this project are "AT" and "64-bit data". The "AT" mode is created to the local device, and this connection can be used to send commands to the xbee unit. Examples include commands for retrieving the address of the local unit, or commands to get/set different settings in the unit. The "64-bit data" type connection is used with remote xbee units, and this type is used for sending the data. This class has a virtual method `xbee_conCallback(libxbee::Pkt** pkt)` which the inheriting class needs to implement. The method `xbee_conCallback(libxbee::Pkt** pkt)` is called for each packet that is received by the connection.

- **libxbee::Pkt**
  This is libxbee's representation of a packet. This holds a pointer to the data contained in the packet together with the length of the data. There is a limit of 100 bytes for each packet.

See [https://code.google.com/p/libxbee/](https://code.google.com/p/libxbee/) for more info.

2.5.2 Automatic Repeat-reQuest

Both the MAC layer of the 802.15.4 protocol, and the libxbee library implements a stop-and-wait protocol. The stop-and-wait protocol implements a scheme where the transmitter waits for an acknowledgement message from the receiver [16]. The stop and wait protocol is favourable because of its simple design, but the time spent waiting for the acknowledgement package decreases the throughput compared to other schemes.
2.6 ASN.1

"Abstract Syntax Notation number One is a standard that defines a formalism for the specification of abstract data types" [17]. ASN.1 is standardized in ISO 8824 and ISO 8825 [17]. The definitions written with the ASN.1 syntax are encoded using encoding rules, which are a part of the ASN.1 standard. The following text presents a brief description of the encoding rules.

2.6.1 Basic encoding rules (BER)

The basic encoding rules use three different types to encode the data.

- **Primitive type.**
  Examples includes INT (ASN.1 type: INTEGER) and double (ASN.1 type: REAL).

- **Constructed, definite-length encoding**
  The constructed type is a type constructed from other primitive types. An example could be a structure to hold a record of a measurement, and the date when the measurement was recorded. The length of the encoded data from this type is always known. The definition for such a type is shown below.

  ```
  Measurement ::= SEQUENCE {
      year INTEGER,
      month INTEGER,
      day INTEGER,
      value REAL
  }
  ```

- **Constructed, indefinite-length encoding**
  The constructed indefinite length type is similar to the type above, but it also has one member where the length is not explicitly set in a field. The end of the type is instead set by a pre-defined end octet.

  The above types are all encoded in series of octets. The different kinds of octets encoded are described below.

- **Identifier octets.**
  These octets sets the class of the ASN.1 value. It also tells the decoder if the type is primitive or constructed by setting bit six to 0 or 1, respectively.

- **Length octets.**
  For the definite-length methods, this set of octets sets the length of the encoded data. The constructed, indefinite-length method does not have a specified length, and this is signalled using a pre-defined value of the length octet. The length octet(s) can be in either short or long form. If the short form is used, then there is only one length octet, with bit number 8 set to 0 and bit 0 to 7 setting the actual length. If bit 8 is set to 1, then the long form is used. Bit 0 to 7 then sets the length of the length octets. The length octets can therefore be encoded with two up till 127 octets using the long form. The following octets are the actual length octets. [18]
• Contents octets.
  When the method to encode a type is the primitive, definite-length method, these octets represent the value of the type encoded. For the constructed methods, these octets contain the encoded types.

• End-of-contents octet.
  When the length is not set explicitly (constructed, indefinite-length method), then this octet signals that the end of the encoded data is reached. The octet used for this is 00. This octet is not used if the method is one where the length is explicitly encoded.

2.6.2 Distinguished encoding rules
The distinguished encoding rules (DER) is a subset of the basic encoding rules. With DER, only one way exists to encode a certain type, which is not the case with BER, where a type can be encoded in different ways. DER is for example used with digital signatures. The DER is compatible with the BER, and therefore a DER encoded stream can be decoded with a decoder using BER.

2.6.3 Asn1.c compiler
The ASN.1 compiler from [http://lionet.info/asn1c/](http://lionet.info/asn1c/) is a compiler that lets the user compile ASN1 definitions into a set of .c and .h files. The .c and .h files contains equivalent structures of those defined. These structures are then used to set the data to be encoded. The compiler will also produce some native types that the structures use, along with C-functions to encode/decode the data that are sent over the link.

2.7 Hardware Components
The testbed consists of Beaglebone Black single board computers equipped with Zigbee USB dongles. In the following chapter a description of these hardware components is given.

2.7.1 Beaglebone black
The Beaglebone Black is a low-cost, credit-card sized computer from Beagleboard. The board comes with an ethernet connector, and this is used to connect the board to a switch. The board is then accessed via SSH. The Beaglebone Black can also be connected to a monitor via an HDMI connector. The board can act as both USB client and host. When using it as a host, the USB hardware must be connected when the board is powered up. The client mode can be used as a power supply, and it is also possible to connect it to a desktop computer and gain access to the board via SSH via a USB to ethernet adapter that is pre-installed in the board. The board comes with an operating system called Angstrom. Angstrom is a Linux distribution made specially for embedded systems. The board also comes with a microSD card reader, and other operating systems can be used by booting from a card with the OS image. The board measures 86.36mm x 53.34mm and the cost is approximately 45 USD. [19]

Other properties:
• AM335x 1GHz ARM Cortex-A8
• 512MB DDR3 RAM
• 2GB 8-bit eMMC on-board flash storage
• NEON floating-point accelerator
• 2x PRU 32-bit micro-controllers
• 2 x 46 pin headers

More information can be found at [http://beagleboard.org](http://beagleboard.org).

Figure 8: Beaglebone Black.

### 2.7.2 XStick

The Zigbee unit used for transmission is the XStick 802.15.4 from Digi International.

Properties [20]:

- Operation range is 15 m
- Transmits at 1 mW (0dBm)
- Transmit and receive current is 64 mA
- Serial interface data rate is supported up to 115.2 Kbps.
The XStick provides retries on top of the retries in the MAC layer. The number of retries can be set to a value between zero and six. For each XStick retry, there are three MAC layer retries.

The XStick presents itself as a serial port in `/dev/tty/USBx` where `x` is a number starting from 0. The XStick can be configured via the X-CTU software available from Digi’s webpage[^1]. The XStick can also be opened in AT-Mode (Attention), and different settings can then be changed by sending strings with commands to the port connected to the XStick. Examples of settings that can be changed are datarate and addressing.

See Appendix B for the configuration settings used for the XStick.

### 3 Design space

In figure 10, a drawing of the known topology of the different nodes in the testbed is shown. To the far left is the camera node. The camera node is connected to a camera. The camera is then able to communicate via Zigbee with the other nodes. The processing nodes sends the result to a sink node, which has a TCP-connection to a server. [10]

### 3.1 Key functionality

The following chapter describes the schemes implemented in the testbed following the descriptions in [10], and how the testbed can be extended to include other schemes.

#### 3.1.1 Area-split and scale-split

- Area-split

[^1]: http://www.digi.com/
Figure 10: Topology of the testbed nodes. The camera node is connected to a camera. The camera node and the sink node also have a mutual xbee connection (not shown in the picture). The sink node is connected to a server via a TCP connection. The server application is not part of the testbed.

The testbed should support the area split scheme when creating tasks. The area split scheme divides the image in equally large areas, called regions of interest (roi). The roi is the area for which each node should detect keypoints and compute descriptors. The image data for the roi’s are transmitted to each node, or, the whole image is transmitted and a parameter to set the roi is transmitted.

In the case of an area split, where each processing node is assigned an area to detect keypoints in, it is important to distribute not only the sub image that the processing node is assigned. The camera node also needs to distribute a portion of the area surrounding each image, because the filters are overlapping in adjacent areas. If this is not done, the detection/extraction process will zero-pad the adjacent area, and then the results will not be the same as if keypoints and descriptors had been computed locally on the whole image.

The following section discusses other schemes to create tasks (not implemented).

- **Scale split** The computing tasks can also be created by separating detection at various scales between the nodes. In this method, one node could for example only detect keypoints in the lowest octave, the second node would only detect keypoints in the second octave, and so on. The majority of keypoints are found in the first octave [10]. To distribute the computing load evenly, the detection of keypoints and extraction of descriptors in the first octave could be distributed to multiple processing nodes, while detection/extraction in higher octaves is performed by a smaller set of nodes.

### 3.1.2 Processing in camera node

The keypoints and features described in [2.3.2] can be computed in various nodes, as described in [10]. The following scheme is supported in the testbed.

- **No detection / extraction in camera**

  With this scheme, the camera node is only responsible for capturing the image and then creating computing tasks for the image. The image and the tasks are then distributed to the different processing nodes. The camera
does not have any information about the load that the task will put on the node, except for the size of the sub image that the node should detect keypoints in. This is the method that is implemented in the testbed.

The following text discusses other extraction schemes. They could possibly be implemented in a future version of the testbed.

- **Detection at the camera node**
  If the image is divided in separate areas, and one area is assigned to each processing node, it is possible that only a few keypoints are found in the part of the image that is assigned to one processing node. Another processing node might detect a large number of keypoints, making the computing load uneven among the different nodes. One way to distribute the tasks so that the computing load is more balanced, is to perform the detection in the camera. This way the number of keypoints, and where in the image the keypoints are located, would be known before the tasks are sent to the different processing nodes. Each processing node would then be assigned the same number of keypoints to compute descriptors for. This way, the camera node would distribute keypoints to compute descriptors for more evenly among the processing nodes.

- **Detection and partial extraction at the camera node**
  The camera could also compute the descriptors for the interest points found in certain layers, and then assign other layers to other nodes. If only the data for the part of the image where the nodes region of interest is located is transmitted to each node, then this could be an advantage. Then, the camera node would detect keypoints in the layers that need to use larger filter sizes, and leave the smaller filter sizes to the processing nodes. In this case the overlapping area around each region of interest is decreased, hence, a smaller image can be transmitted to each node.

### 3.1.3 Separation of control and data

Efficient communication between nodes is important for a fast and extendible testbed. Separation between data messages, and control messages facilitates distribution of tasks and results. The testbed should be implemented with separated messages for data and control messages. An example of a data message is an image and an example of a control message is a command to detect keypoints for an image. If these two message are kept separate, additional control messages can be sent, and those control messages can refer to already received data. If data and control were sent jointly, an image would be sent with each message to detect keypoints or extract descriptors, thus, dramatically increasing the amount of data that is sent from the camera node.

### 3.1.4 Dynamic configuration of the off-loading scheme from the server node

The objects in an image are not necessarily located such that each processing node is assigned a task so that the computing load is distributed evenly. One use case could be when a series of images is to be processed. If the series of images is taken within a relatively short interval, then the location of the object is
probably correlated between the images. If one object produces a large amount of keypoints, and if the node that is assigned the region where the object is located also should compute descriptors for the keypoints, then that processing nodes computation load will be large for consecutive images. If a new picture is taken within a small time interval the system could assign that region to another node, thus distributing the high computational tasks to different nodes.

To facilitate this, timing info is added to a completed task before it is sent to the server. The server could then compare the timing info in each task for an image, and with this information it is possible to approximate the computing load in each node.

4 System Design

A node can have the role as a camera, as a processing nodes, or as a sink node. Each node is identified by a variable \texttt{nodeid}. When the application is started it is given a list of the different nodes in the system which also sets the role of each node. Zigbee connections are created to all of the other nodes in the list, and if the node is acting as sink node, the application also tries to connect to an open TCP socket in the server.

4.1 Testbed workflow

The following text describes the work flow in each node, from the command to capture an image and process it, to sending the results to the server for object matching.

![Sequence diagram for the initial commands to take a picture.](image)

Figure 11: Sequence diagram for the initial commands to take a picture. Bold arrows represent a synchronous call and an outlined arrow represents an asynchronous call (an outlined arrow is an arrow which is not "filled"). A dashed line represents communication to another node.

1. The server sends a command to the sink node to capture an image. See Figure [11]

2. The sink node sends the message onwards to the camera node. See Figure [11]

3. The camera node receives the message, and captures the image.

4. The image is then broadcast from the camera node to all other nodes (including the sink node)
5. The camera node creates tasks for the image according to the setting for creating tasks.

6. The tasks are added to a queue, and one by one the tasks are checked if they have been executed (they have not, but the queue is also used for received tasks). If they haven’t been executed, the identifier that tells the application which node that is the executing node is compared to the node identifier of the node where the application is run. If the tasks should be executed at the camera, then the node adds it to the queue for execution. If the node to execute the task is another node, it is sent to that node. If a task is completely executed then the task is sent to the sink node.

7. The processing nodes receive a task and the task is added to the same queue described in the previous step.

8. The sink node receives a task and the task is also in this case added to the same queue described in the previous step. Because this is the sink node it is not sent to the sink node, it is instead added to the structure for saving completed tasks for a particular image, along with information about what time the completed task is received. When a new task is received, the node performs a check to see if all the tasks for that image are received. If they are, the result is sent to the server via the TCP connection.

4.2 Class layout

The following section describes the different classes and their relation to each other.

![Class Diagram]

Figure 12: Overview of classes and their relation to each other. An arrow from one class to another means that the first class has a pointer to an instance of the second class. Two classes that have a pointer to a class does not necessarily share the same instance of the class.
Figure 13 shows an overview of the NodeManager class. The NodeManager is the central class of the application. An instance of the DfeSettings class is input to the constructor of the NodeManager. The NodeManager retrieves the nodid from the xbee unit. The NodeManager identifies the role of the node in the testbed depending on which identifier is set to each role in the settings. The NodeManager then creates the different classes needed for the role in question. The nodid is also used during the various operations to make different choices depending on the role of the node. For example; a connection to a server is only created if the nodid matches the nodid in the settings that is specified as the sink node. The NodeManager creates the libxbee instance, and will then input a pointer to this instance when each instance of the Connection class is created.

Figure 13: UML class diagram of the NodeManager class.

4.2.2 Connection

Figure 14 shows an overview of the Connection class. The Connection class inherits from the libxbee superclass ConCallback. The ConCallback is a C++ interface of libxbee's c-functions. The constructor takes a pointer to the instance of Xbee (which is connected to the hardware), and the adress of the remote xbee unit to create a connection to. The Connection that inherits from
the ConCallback superclass has to implement a method `xbec::conCallback(pkt)`. This method will be called for each packet received.

The ConCallback can also be created with an address that specifies a broadcast transmit. Packets received from a broadcast transmit will also cause the callback method to be called for each packet.

If a second packet is received, the callback for the second packet is called when the callback triggered from the first packet has returned. When the callback is called for a newly received packet, the Connection class method `ConCallback` adds a pointer to the packet to a queue available in the Connection class own instance of the Receiver class. The callback can then return and a new packet can be processed. This way, consecutive packets can be processed directly without being held up, waiting for a callback that was triggered from a previous packet to return.

### 4.2.3 Receiver

<table>
<thead>
<tr>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>saved_bytes ::uint8_t</td>
</tr>
<tr>
<td>num_saved_bytes ::int</td>
</tr>
<tr>
<td>~Receiver()</td>
</tr>
<tr>
<td>Receiver(inlmg_man : source::ImageManager, inout task_man : source::TaskManager, inout node_m...</td>
</tr>
<tr>
<td>DoStuffToObject(inlmg_lxbee_pkt : libxbee::Pkt) ::int</td>
</tr>
<tr>
<td>RoutePacket(inlmg_pkt : source::Packet) ::TypeHeaderAsn_PR</td>
</tr>
<tr>
<td>CheckNewPacket(in pkt_h : PktHeaderAsn_fly ::int</td>
</tr>
<tr>
<td>TakeAction(in : TypeHeaderAsn_fly ::TypeHeaderAsn_PR</td>
</tr>
<tr>
<td>DecodeAndTakeAction(in bufl : uint8_t, in ms::size ::int) ::TypeHeaderAsn_PR</td>
</tr>
</tbody>
</table>

Figure 15: UML class diagram of the Receiver class.

Figure 15 shows an overview of the Receiver class. One instance of the Receiver is created for each connection. The receiver also creates an instance of the Decoder class.

The Receiver fetches pointers to packets from its internal queue. The packet header in each packet is decoded, and the packet number and the number of total packets that this packet belongs to is read. For each batch of packets, i.e packets that have the same number of total packets, the Receiver creates an instance of the ReceiveItem class, and then adds the packet that belongs to this batch to the ReceiveItem instance. As long as a received packet have the same number of total packets, it is added to the same instance. If the last packet is received, the receiver calls a method in the ReceiveItem class to check if all packets have been received. If so, the ReceiveItem is used to concatenate the data in the packets, and then the Decoder is used to decode the data divided among the multiple packets. When the information in the data is decoded, the Receiver will processes the data accordingly.
Figure 16: Sequence diagram for the Receiver::DoStuffToObject. The method names that do not have a namespace preceding the name belong to the Receiver. Bold arrows represent a synchronous call and an outlined arrow represents an asynchronous call.
As specified in the ASN.1-definition file, shown in section 4.3, the different messages that can be decoded by the receiver are listed below, along with the actions that are performed upon receiving them.

- An image.

Upon receiving an image the receiver will call the `ImageManager` to create the image. The `ImageManager` will also add it to its "library". See Figure 17.

![Sequence diagram for the case when an image is received in the Receiver. The method names that do not have a namespace preceding the name belong to the ImageManager. Bold arrows represent a synchronous call and an outlined arrow represent an asynchronous call.](image)

- A task to find keypoints and/or compute descriptors for the keypoints.

The `Receiver` will call the `ImageManager` to get a pointer to the image that should be available at the receiver before the task for the image is received. If the image is available, then the task manager is called to create the task and take further action. If the image is not available, then no task is created. See figure 18. *Functionality to take actions when an image cannot be found can be added here. For example, a task that could not be paired with its image could be saved to the TaskManager for processing in the future when the missing image is finally received.*

The task received can be an already finished task, and if that is the case the node where the application is running should be the sink node. The task is still added to the `TaskManager`, and the `TaskManager` is responsible to process it accordingly.

- A message to take a new picture.

The receiver will call a method in the `NodeManager` to take a picture. The `NodeManager` will then call necessary methods to capture and process the image.
Figure 18: Sequence diagram for the case when a task is received in the Receiver. The method names that do not have a namespace preceding the name belong to the TaskManager. Bold arrows represent a synchronous call and an outlined arrow represents an asynchronous call.

- A message about a finished task.
  The receiver will call the TaskManager method to delete the task from its bookkeeping.

4.2.4 ReceiveItem

Figure 19: UML class diagram of the ReceiveItem class.

The stop and wait mechanism should ensure that a packet is acknowledged.
when it is received. Also, a new packet should not be sent before the previous packet is received, or if all the retries time out. So, if all packets were successfully sent then the packets should have arrived in the correct order. However, the same functionality is not possible when data is broadcast, since there are no acknowledgement packets sent. The ReceiveItem class is used to store packets that are received. The class contains methods to check if packets are lost or if a packet has already been received. These methods are used before decoding to make sure that all the data is present. If a packet is lost, and the data of the received packets are concatenated as the packets arrive, then it will not be possible to decode the data. The same problem arises when a packet is received multiple times and no checking is done if the data have already been received.

The Receiver creates an instance of the ReceiveItem class for each batch of packets. An overview is shown in figure 19. When a new packet is received, the packet is added to the instance of the ReceiveItem class. If the packet already has been received, it is discarded. When the last packet in the class is received the ReceiveItem class can be called to check if all the packets have been received. This class can be used to implement various automatic repeat request schemes. For example; if all frames were not received, the class can be used to detect which packets that were not received, and then send a request to the sender to re-send those packets.

4.2.5 DfeImage

The DfeImage class is mainly just a wrapper around the OpenCV Mat class. The DfeImage class adds an identifier to the image. There are also methods for serializing the Mat data.

4.2.6 ImgTask

The class ImgTask is the class that creates instances of classes from the OpenCv library used to detect the keypoints and to compute the descriptors. An overview of the ImgTask is shown in figure 20. The results from the SURF operations are stored in internal variables in the class. The class also implements functionality to fill a struct needed for serializing the class with the ASN.1 encoding scheme. Other methods in the class include functionality to compute the number of pixels that a region of interest needs to be extended with. This is to make sure that there is no unnecessary zero padding because of missing values when filtering the image with the box filters used in the keypoint detection and descriptor computation.

The ImgTask class inherits from an interface, and this interface is used as input/output type for a task. This will facilitate for other types of tasks to be added in the future. OpenCv includes other feature extraction algorithms like SIFT, which could be added to extend the family of supported algorithms by creating a new class and inherit from the interface.

4.2.7 ImageManager

The ImageManager class is used to store images received or created in the node. An overview is shown in figure 21. The camera node will send the image in a separate message when distributing tasks to the nodes. The image that is
processed is broadcast to all the nodes, and then the task to be carried out by each node is sent separately. Therefore, the image that the task is to be carried out on is received first. When the Receiver receives an image, it is sent to the ImageManager. The ImageManager adds it to its internal structure, identifying it with its id. When a task is received, the task identifies the image to operate on by its id. This id is input to the ImageManager which will return a pointer to the image that the tasks should be carried out on. This way, an image only need to be transferred once to all nodes, instead it being transferred together with the task.

The ImageManager also has methods to capture an image from the camera, or from an already existing file.
4.2.8 TaskCreator

The class TaskCreator is responsible for creating tasks for an image and a set of nodes. An overview is shown in figure 22. The TaskCreator class will divide the image in regions of interest, and create tasks with the supplied detection / extraction parameters.

4.2.9 TaskManager

Received or created tasks are added to the TaskManager, which is a subclass to the ThreadedQueue. An overview of the TaskManager class is shown in figure 23. The task manager is responsible for deletion of all tasks that are still in memory when the application terminates. The TaskManager adds all tasks to its own queue, and a separate thread fetches tasks from the queue and processes
them in the method `DoStuffToObject`, which is implemented by all subclasses to the `ThreadedQueue`. See figure 24. This way, the method that added the task does not have to wait until the task is processed. The `TaskManager` responsibility is not to actually carry out the detection / extraction for the tasks. The `TaskManager` checks if the tasks are to be carried out at this node or at another node. If a task is to be carried out at this node then it is added to the `TaskExecuter`. If the task is supposed to be carried out at another node, then the task is sent to that node. If the task is completed, then the task is added to an instance of the `CompletedTasks` class that corresponds to the image id of the task. If all tasks for the image are completed, then the `TaskManager` will call the appropriate method of the `TcpSender` to send the results to the server.
Figure 24: Sequence diagram for TaskManager::DoStuffToObject. The method names that do not have a namespace preceding the name belong to the TaskManager. Bold arrows represent a synchronous call and an outlined arrow represents an asynchronous call. A dashed line represents communication to another node.
4.2.10 TaskExecturer

As the name indicates, this class is responsible for execution of tasks. The TaskExecturer is a sub-class to the ThreadedQueue, and an overview is shown in figure 25. Tasks that should be executed at the node are added to the queue in the TaskExecturer. This way, the TaskManager can add tasks to the queue and then return to process the next task. The next task could be a task that is to be executed at another node. If the TaskExecturer did not have a queue and a thread working on the tasks in the queue, then the TaskManager would be kept waiting while the task was executed. Other tasks waiting to be processed by the TaskManager would then be held up, and not sent until the first task had been executed. When the thread that executes tasks from the queue has executed a task, and the task is completely executed, then the task is sent to the sink node. If descriptors should be computed in another node, then the task is sent to that node. See figure 26. It should be very simple to add the possibility to execute tasks at the sink node. This could be advantageous since the sink node is mostly idle when the processing nodes are doing work. The TaskExecturer would then pass the executed task back to the TaskManager. The TaskManager will detect that it is executed, and add it to the correct instance of the CompletedTasks class. See section 4.2.11.
Figure 26: Sequence diagram for TaskExecuter::Execute. The method names that do not have a namespace preceding the name belong to the TaskExecuter. Bold arrows represent a synchronous call and an outlined arrow represent an asynchronous call. A dashed line represents communication to another node.
4.2.11 CompletedTasks

The CompletedTasks class has functionality to store pointers to executed tasks, an overview of the methods for this class is shown in figure 27. Only tasks that originate from the same image can be added to the same instance of this class. A task that is pointing to an image with another id compared to tasks added prior cannot be added. When a completed task is added, a timestamp is added to the completed task. This can be used to estimate the processing load in the different nodes. The CompletedTasks class also implements a method to check if all tasks for the image have been added. The method computes the total area of the regions of interests of the completed tasks. If the size of this area is the same size as the area of the image, then all tasks must have been completed. The class also implements a method to fill an ASN.1-structure needed to serialize the completed tasks along with the timing info.

4.2.12 Encoder

To encode, the Encoder implements one method for each of the different
types that need to be serialized. An overview of the class is shown in figure 28. The Encoder is just a wrapper around the c-functions generated by the Asn1c compiler. If another type of encoder should be used, only this class needs to be changed. The Encoder adds a small header to the encoded array that sets which type that is encoded. This way the type can be identified when the array is to be decoded. The objects that is input to the Encoder is hereby encoded as the same type, so when the array is decoded, it can be decoded as the same type, and then a series of octets tells the decoder what type that follows.

4.2.13 Sender

![Image of UML class diagram of the Sender class.]

The Sender class implements a template method to send objects. An overview of the class is shown in figure 29. The Sender class creates its own instance of the Encoder class, and this class is used to encode the different objects. By having the send method taking a template as input parameter, it is easy to extend the Sender’s capability to send other types of objects. When the Sender has encoded the object into an array of bytes, the Sender will create an instance of the SendItem class. This class takes an array of data and divides it into packets. The Sender class is a sub class to the ThreadedQueue, and the ThreadedQueue protects its internal queue with mutexes, so the Sender’s send method can be called from multiple places without the risk of a deadlock. Also, by having the Sender fetch data to be sent from a queue, the method that is sending an object can return directly after the SendItem is created and added to the queue. See figure 30 and figure 31. Hence, the calling method does not have to wait until the data are sent.

4.2.14 Decoder

The Decoder takes a pointer to data and the length of the data. An overview is shown in figure 32. The decoder returns a struct, with one field pointing to the decoded object, and one field that says what type of object that has been decoded.

4.2.15 TcpClient, TcpSender, TcpReceiver

The TcpSender and the TcpReceiver is used in the sink node to connect to the server through a TCP connection. The TcpClient initializes and connects to the socket. The TcpClient then creates an instance of the TcpSender and an
instance of the TcpReceiver. A pointer to the socket created by the TcpClient is passed to the TcpReceiver and the TcpSender when they are created. The TcpClient is only created if the node is the sink node. An overview of the three classes is shown in figure [33].

The TcpSender is a subclass to the ThreadedQueue. When the method to send an object is called, the TcpSender encodes the object using its own instance.
of the Encoder. The TcpSender creates a struct with a pointer to the encoded data and the length of the encoded data. This struct is then added to the queue implemented in the ThreadedQueue class. The TcpSender then implements a method that is called for each object in the queue, and this method sends the data pointed to by each struct.

The TcpReceiver creates a thread and runs a method that reads from the socket that is input to the TcpReceiver when it is created. The TcpReceiver has access to a pointer to the Sender class created by the NodeManager, and can with this pointer send messages to nodes that are received by the sink node. For example, when a message to capture an image is received, the sink node sends this message to the camera node via the Sender.

### 4.2.16 DfeSettings

To facilitate different parameters, a settings class is available. The class implements methods used to parse a settings file. An overview is shown in figure 34. A similar class for settings related to the SURF algorithm is available. This class should come to use if the application is extended with other functionalities, where the user would like to change between different settings, and to automate test related to the functionality. An example of a settings file is

```plaintext
capture_from_camera=0
```
4.2.17 ThreadedQueue

Threads are used in multiple classes for the application. An overview of the superclass ThreadedQueue is shown in figure 35 which forms the base for all threaded queues in the application. The class presents a method to add tasks to the queue, so if one would like to change the inner representation of the queue, then it is possible without breaking other parts in the application. The TaskExecuter inherits from this class so that a thread continuously fetches tasks from a queue where tasks to be executed have been added. The Sender also inherits from the ThreadedQueue class. The objects to be sent are added to the queue and the thread then fetches the objects in the queue and calls the method implemented in the Sender to send them to the connection specified.
4.3 ASN.1-types

The type used for sending image data is the MatAsn type.

\[
\text{MatAsn ::= SEQUENCE} \\
\text{  numrows INTEGER,} \\
\text{  numcols INTEGER,} \\
\text{  type INTEGER,} \\
\text{  data BIT STRING}
\]

It includes fields for setting the number of rows, the number of columns, the type of the data stored at each index in the matrix and a pointer to the data. To recreate the OpenCv core type Mat, the Mat class presents a constructor that takes these four parameters. The length of the data stored at the pointer position can then be computed with

\[ l = \text{bitdepth}/\text{bitperbyte} \times \text{cols} \times \text{rows} \times \text{channels}. \]  \(8\)

For example, consider the OpenCV type CV_16SC3, which defines an image where each element has:

- \text{bitdepth} = 16
- \text{channels} = 3

With \text{bitperbyte} = 8 and an image with \text{cols} = 10 and \text{rows} = 20, the length of the data is 1200 bytes.

The full definition of the various ASN.1 types can be viewed in appendix A.

4.4 Overhead in packets

The type used for sending packets is the PktHeaderAsn type. The ASN.1 definition of this type is
Libxbee has limited the packet size to 100 bytes, so the SEQUENCE type, the field to set the packet number, the number of total packets, and the data in the BIT STRING type, cannot exceed 100 bytes. In the following subsection an analysis of each component's contribution to the size of the packet header is made.

The SEQUENCE type includes one identifier to set the type and this identifier is one octet long. The following octet is the length octet. The total length of the packet cannot exceed 100 so the short form is sufficient for the length, see section 2.6.1. The total amount of octets for the SEQUENCE is therefore two.

The pktnum and the totpkts are both INTEGERS. The identifier for an INTEGER is an octet with value 0x02. The maximum value that can be encoded into the field is limited by the testbed to 65535. The length octet of an INTEGER therefore requires one octet, see section 2.6.1. Encoding the pktnum and the totpkts value will require one octet if the value is below 128, and two octets if the value is 128 or above [21]. Three octets will therefore be used by the pktnum and the totpkts fields at least, and four octets at the most.

The BIT STRING type is identified with the value 0x03, which takes up one octet. Since the maximum size of the whole packet is 100 bytes, the length octet of the BIT STRING can be encoded with the short form, thus taking up one byte, see section 2.6.1. The length of the data in the BIT STRING can also never be above 127 bytes. The length of the data can therefore be encoded with one byte, plus one byte for setting the amount of bits not used in the last byte. This results in two bytes needed to encode the data length field of the BIT STRING. This gives us three bytes total for the BIT STRING.

The BIT STRING type could possibly be changed to an ASN.1 type that does not have the option to set unused bits, since the data that is encoded always takes up an even amount of bytes. This is true not only for the PktHeaderAsn type.

<table>
<thead>
<tr>
<th>Name</th>
<th>Size (min)</th>
<th>Size (max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQUENCE</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>pktnum INTEGER</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>totpkts INTEGER</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>typeheaderasn BIT STRING</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>11</strong></td>
<td><strong>13</strong></td>
</tr>
</tbody>
</table>

Consequently, the packet header takes up 11 bytes at its minimum, and 13 bytes at its maximum. The amount of data that can be sent within each packet is then 89 bytes to 87 bytes, depending on the total number of packets and the packet number. The current encoding method always adds 87 bytes of data to each packet. Logic to determine if one byte is used to encode the INTEGER’s value could be added to pack one extra byte in the data array for each INTEGER type that has a value of 127 or less.
4.5 Maximum supported data size

A maximum packet number of 65535 for the packet number limits the maximum data size for a type to be sent. With 87 bytes per packet, and a maximum of 65535 packets, the largest data array that can be transmitted is 87 times 65535, which is 5701545 bytes.

5 Experimental results

To evaluate the testbed in a situation that resembles a network of sensor nodes, the CPU power is limited in each Beaglebone to 5% of the CPU power. This is done by using the software CPULimit. When CPULimit is started it takes an argument where one specifies the executable that should have limited CPU power. The BeagleBone Black has a CPU at 1 GHz, and limiting it to 5% gives us an effective CPU of 5 MHz, which is in the range of a sensor node.

5.1 802.15.4 Data rate measurement

The XStick operates in the 2.4 GHz band, and at this frequency the maximum over-the-air data rate is 250 kbps. Due to overhead in the Zigbee protocol the actual theoretical maximum data rate is approximately half of that. The evaluation was performed with a baud of 9600 bits per second. The reason for using a lower baud between the Xstick and the Beaglebone is that many packet losses was observed when using a higher rate.

With approximately 100 bytes in each packet, the image of size 223x324 is transferred with 813 packets (the image data takes up $223 \cdot 324 \cdot 8 = 578016$ bits). With $813 \cdot 10^2$ bytes transferred in 155 seconds, we have a data rate of $813 \cdot 10^2 \cdot 8 \text{ bits} / 155 \text{ s} \approx 4.2 \text{ kbits} / \text{s}$. The ratio between the theoretical maximum data rate and the data rate of the testbed is then $4.2 \text{ kbits} / 125 \text{ kbits} \approx 0.034$. One parameter that is decreasing the data rate is the lowered data rate between the Xstick and the Beaglebone. The Xstick supports a data rate up to 115200 bits per second, and when tests where performed on a desktop computer using a baud of 115200 then the 223x324 image could be transferred in 54 seconds. The ratio between the over-the-air data rate and the maximum theoretical over-the-air data rate, using a baud of 115200, is then approximately $0.1$. If a protocol for resending packets that are dropped is implemented, then it is possible to use a higher data rate between the XStick and the Beaglebone, and thereby increasing the available over-the-air data rate.

The data rate was also measured when the processing power of the Beaglebone was not limited. The image in the previous example was then transferred in 153 seconds, so the decreased processing power does not have such a high impact on the over-the-air data rate.

The observed data rate is the same for both broadcasting and direct addressing.

\[^2\text{http://cpulimit.sourceforge.net/}\]

\[^3\text{The author of libxbee has also performed an evaluation of the over-the-air data rate, using a Xbee series 1 module with a baud rate of 57600. The speed was then 12.73 kbs/s}\]
5.2 Distributed keypoint detection and descriptor extraction performance

In this section the results from an evaluation of the system in terms of time consumed at different stages is presented. The different stages that are clocked are:

- Time taken to transfer an image
- Time taken to detect keypoints
- Time taken to compute descriptors
- Time taken to transfer a completed task

The image used for evaluation is the image used to detect keypoints, shown in figure 4. The system is evaluated at the Hessian thresholds 100, 500, 1000, 1500, 2000, 2500 and 3000, for each of the image sizes 223 x 324, 112 x 162 and 55 x 81. One processing node was used. The SURF settings used are:

- extended = false
- nOctaveLayers = 3
- nOctaves = 4
- upright = false

The results shown in figure 36 shows that the time to transfer the image is proportional to the image size.

The number of keypoints found for each image and value of the Hessian threshold is shown in figure 37. For an image size of 223x324 and 111x162, one can see an abrupt decrease in the number of keypoints found between a Hessian threshold of 100 and 500. This change is not so large in 55x81. The reduced amount of filters used in 55x81 decrease the amount of results, so there are not so many keypoints found with a low response value, and this reduces the amount of keypoints in the interval 100 to 500. The amount of keypoints found is larger for larger images, but it is not proportional to the image size. This is also because of the filter sizes that can not be fitted in the smaller image, thus, keypoints at larger scales cannot be found because there is simply not enough pixels to detect them.

The time to detect keypoints is larger for larger image sizes obviously, but the time taken to compute the keypoints is not proportional to the image size. This is because the filter sizes that do not fit in the smaller images are not used, hence decreasing the effective amount of filters. In figure 38 we can also see that the time spent on detecting keypoints is also dependent on the value of the Hessian threshold. From this we can say that there must be some optimizing functionality in OpenCV to speed up the process when the threshold is larger.

The time spent on computing descriptors is shown in figure 39. The time spent on computing descriptors is linearly proportional to the amount of keypoints found within the image. This is expected since one descriptor is computed for every keypoint found. The time it takes to compute descriptors is however dependent on the image size. One can for example see that at approximately
200 keypoints the time taken to compute descriptors is different for the 223x324 image compared to the 111x162 image. This is probably because that there are more keypoints found at a larger scale in the 223x324 image, and computing descriptors for larger areas is more time consuming. This needs to be verified however.

The time spent on transferring keypoints and descriptors is shown in figure 40. Similar to the proportionality in time taken to transfer the image relative to image size, the time taken to transfer the detected keypoints and computed descriptors is proportional to the amount of keypoints found.

Figure 41 shows the total time spent on transmitting image and results data, and the total time spent on processing in each processing node. Time spent on transferring data is larger for every image size, so no speed-up in processing time is gained with a distributed system. However, there is no necessity for a central server, so one could possibly accept the increased processing time in favour of a more simple system. If different compression techniques is applied to the image data and the keypoints / descriptors, the time spent on transferring data could be decreased. Keypoint data are transferred to the sink node, though some matching algorithms only use the descriptor data with no information about where in the image the descriptors was computed [23]. To further decrease the amount of data needed to be transmitted, the keypoint information could be left out when sending results to the sink node. See section 5.1 for a discussion about the data rate of the testbed.
5.3 Unit testing

Unit tests are written with Google’s GTest framework, and extensively make use of mock classes to test the intended functionality. A mock class is used to test the behaviour of another class. The mock class inherits from the same interface as the class it is implemented to mock. The mock class can then be used to check whether a certain method was called, how many times it was called, with what type of input parameters it was called and the values of the input parameters. The mock objects can also be used to check in what order the mock object’s method was called. Basic functionality of each class is tested and this should facilitate future work that builds on this project.

The project is hosted at [https://code.google.com/p/distributed-feature-extraction/](https://code.google.com/p/distributed-feature-extraction/)

References


Figure 38: The time spent detecting keypoints for each image and for different values of the Hessian threshold.


Figure 39: The time spent computing descriptors for each image and for different values of the Hessian threshold.


Figure 40: Time spent transferring keypoints and descriptors to the sink node.


Figure 41: The total time spent on transferring data and the total time spent on processing (detecting keypoints and computing descriptors).

Appendices

A ASN.1 types

```asn1
/**
 * Creator: ASNDT (http://www.asnlab.org)
 * Author: andreas
 * Created: Mon Mar 25 20:36:37 CET 2013
 */
DistributedFeatureExtraction DEFINITIONS AUTOMATIC TAGS ::= BEGIN

-- imports and exports

EXPORTS ;
IMPORTS ;

-- type assignments

ImageAsn ::= SEQUENCE {
  matdata MatAsn,
  imageid INTEGER
}
```
MatAsn ::= SEQUENCE {
    numrows   INTEGER,  
    numcols   INTEGER,  
    data      BIT STRING,  
    type      INTEGER  
}

ImgTaskAsn ::= SEQUENCE {
    taskid   INTEGER,  
    imgid    INTEGER,  

    -- task parameters  
    kpexecute BIT STRING,  
    dexecute  BIT STRING,  
    keypoints-detected BOOLEAN,  
    descriptors-extracted BOOLEAN,  
    rect      RectAsn,  
    parameters AlgorithmParametersAsn,  

    -- Result  
    keypoints SEQUENCE OF KeyPointAsn,  
    descriptors MatAsn
}

ParametersAsn ::= CHOICE {
    surfparameters SurfParametersAsn
}

AlgorithmParametersAsn ::= SEQUENCE {
    detectionparameters ParametersAsn,  
    extractionparameters ParametersAsn
}

SurfParametersAsn ::= SEQUENCE {
    extended BOOLEAN,  
    hessianthreshold REAL,  
    noctavelayers INTEGER,  
    noctaves INTEGER,  
    upright BOOLEAN
}

TaskFinishedAsn ::= SEQUENCE {
    taskid   INTEGER
}

ReportNewNodeAsn ::= SEQUENCE {
    adress   BIT STRING
}

DeleteNodeAsn ::= SEQUENCE {
    adress   BIT STRING
}


RectAsn ::= SEQUENCE {
  x INTEGER,
  y INTEGER,
  width INTEGER,
  height INTEGER
}

KeyPointAsn ::= SEQUENCE {
  angle REAL,
  klass INTEGER,
  octave INTEGER,
  x REAL,
  y REAL,
  response REAL,
  size REAL
}

PktHeaderAsn ::= SEQUENCE {
  pktnum INTEGER,
  totpkts INTEGER,
  typeheaderasn BIT STRING
}

TypeHeaderAsn ::= CHOICE {
  imageasn ImageAsn,
  imgtask ImgTaskAsn,
  taskfinishedasn TaskFinishedAsn,
  reportnewnodeasn ReportNewNodeAsn,
  deletenodeasn DeleteNodeAsn,
  takepicture TakePicture
}

TakePicture ::= ENUMERATED {
  takepicture
}

TimingInfoAsn ::= SEQUENCE {
  hour INTEGER,
  minute INTEGER,
  second INTEGER,
  millisecond INTEGER
}

CompletedTasksAsn ::= SEQUENCE {
  imgtasks SEQUENCE OF ImgTaskAsn,
  timinginfos SEQUENCE OF TimingInfoAsn
}

-- value assignments

END
B XStick configuration

This is the configuration used for the XStick. The Destination address high, the Destination address low and the 16 bit source address is important for the communication between nodes. Setting the 16 bit source address to FFFF disables the use of 16 bit addresses which is necessary when using the 64 bit addresses.

Setting the xbee retries to 6 will make the XStick retry to send a packet six times. This will result in 6 times the number of MAC layer retries plus one retry, totalling in 19 retries before the packet is skipped.

The Interface data rate can be set to the users preference, but the value need to be reflected in the value passed to the libxbee::Xbee constructor. This value is set in dfe_typedefs.h with a #define BAUDRATE statement. Setting it to a too high value will speed up transmission, but also result in more packet losses.

The API enabled parameter needs to be enabled (set to 1) for libxbee to function properly.
<table>
<thead>
<tr>
<th>Name</th>
<th>AT command</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>CH</td>
<td>C</td>
</tr>
<tr>
<td>Pan id</td>
<td>ID</td>
<td>3332</td>
</tr>
<tr>
<td>Destination address high</td>
<td>DH</td>
<td>0</td>
</tr>
<tr>
<td>Destination address low</td>
<td>DL</td>
<td>0</td>
</tr>
<tr>
<td>16 bit source address</td>
<td>MY</td>
<td>FFFF</td>
</tr>
<tr>
<td>Mac mode</td>
<td>MM</td>
<td>0</td>
</tr>
<tr>
<td>xbee retries</td>
<td>RR</td>
<td>6</td>
</tr>
<tr>
<td>Random delay slots</td>
<td>RN</td>
<td>0</td>
</tr>
<tr>
<td>Node discovery time</td>
<td>NT</td>
<td>19</td>
</tr>
<tr>
<td>Node discovery options</td>
<td>NO</td>
<td>0</td>
</tr>
<tr>
<td>Coordinator enabled</td>
<td>CE</td>
<td>0</td>
</tr>
<tr>
<td>Scan channels</td>
<td>SC</td>
<td>1FFE</td>
</tr>
<tr>
<td>Scan duration</td>
<td>SD</td>
<td>4</td>
</tr>
<tr>
<td>End device association</td>
<td>A1</td>
<td>0</td>
</tr>
<tr>
<td>Coordination association</td>
<td>A2</td>
<td>0</td>
</tr>
<tr>
<td>AES encryption enabled</td>
<td>EE</td>
<td>0</td>
</tr>
<tr>
<td>Node identifier</td>
<td>NI</td>
<td></td>
</tr>
<tr>
<td>Power level</td>
<td>PL</td>
<td>4</td>
</tr>
<tr>
<td>CCA Threshold</td>
<td>CA</td>
<td>2C</td>
</tr>
<tr>
<td>Sleep mode</td>
<td>SM</td>
<td>0</td>
</tr>
<tr>
<td>Time before sleep</td>
<td>ST</td>
<td>1388</td>
</tr>
<tr>
<td>Cyclic sleep period</td>
<td>SP</td>
<td>0</td>
</tr>
<tr>
<td>Disassociated cyclic sleep period</td>
<td>DP</td>
<td>3E8</td>
</tr>
<tr>
<td>Sleep options</td>
<td>SO</td>
<td>0</td>
</tr>
<tr>
<td>Interface data rate</td>
<td>BD</td>
<td>3</td>
</tr>
<tr>
<td>Parity</td>
<td>NB</td>
<td>0</td>
</tr>
<tr>
<td>Packetization timeout</td>
<td>RO</td>
<td>3</td>
</tr>
<tr>
<td>API enabled</td>
<td>AP</td>
<td>1</td>
</tr>
<tr>
<td>Pin settings</td>
<td>D8</td>
<td>0</td>
</tr>
<tr>
<td>Pin settings</td>
<td>D7</td>
<td>1</td>
</tr>
<tr>
<td>Pin settings</td>
<td>D6</td>
<td>0</td>
</tr>
<tr>
<td>Pin settings</td>
<td>D5</td>
<td>1</td>
</tr>
<tr>
<td>Pin settings</td>
<td>D4 - D0</td>
<td>0</td>
</tr>
<tr>
<td>Pull up resistor enabled</td>
<td>PR</td>
<td>FF</td>
</tr>
<tr>
<td>I/O Enabled</td>
<td>IU</td>
<td>1</td>
</tr>
<tr>
<td>Samples before TX</td>
<td>IT</td>
<td>1</td>
</tr>
<tr>
<td>DIO change detect</td>
<td>IC</td>
<td>0</td>
</tr>
<tr>
<td>Sample rate</td>
<td>IR</td>
<td>0</td>
</tr>
<tr>
<td>PWM0 Configuration</td>
<td>P0</td>
<td>1</td>
</tr>
<tr>
<td>PWM1 Configuration</td>
<td>P1</td>
<td>0</td>
</tr>
<tr>
<td>PWM Output timeout</td>
<td>PT</td>
<td>FF</td>
</tr>
<tr>
<td>RSSI PWM Times</td>
<td>RP</td>
<td>28</td>
</tr>
<tr>
<td>I/O Input adresses</td>
<td>IA</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>T0 - DO to T7 - D7 Output addresses</td>
<td>T0</td>
<td>FF</td>
</tr>
<tr>
<td>Device type identifier</td>
<td>DD</td>
<td>10000</td>
</tr>
<tr>
<td>AT Command Mode Timeout</td>
<td>CT</td>
<td>64</td>
</tr>
<tr>
<td>Guard Time</td>
<td>GT</td>
<td>3E8</td>
</tr>
<tr>
<td>Command Sequence character</td>
<td>CC</td>
<td>2B</td>
</tr>
</tbody>
</table>