MARVELO – A Framework for Signal Processing in Wireless Acoustic Sensor Networks

Haitam Afifi[1], *Joerg Schmalenstroeer*[2], *Joerg Ullmann*[2], *Reinhold Haeb-Umbach*[2], *Holger Karl*[1]

Computer Networks Group[1], Department of Communications Engineering[2], Paderborn University
Email: {haitam.afifi,karl}@uni-paderborn.de, {schmalen,haeb,ullmann}@nt.uni-paderborn.de

Abstract

Signal processing in Wireless Acoustic Sensor Networks (WASN) is based on a software framework for hosting the algorithms as well as on a set of wireless connected devices representing the hardware. Each of the nodes contributes memory, processing power, communication bandwidth and some sensor information for the tasks to be solved on the network.

In this paper we present our MARVELO framework for distributed signal processing. It is intended for transforming existing centralized implementations into distributed versions. To this end, the software only needs a block-oriented implementation, which MARVELO picks-up and distributes on the network. Additionally, our sensor node hardware and the audio interfaces responsible for multi-channel recordings are presented.

1 Introduction

Distributed signal processing in sensor networks has attracted a lot of attention in the past years [1, 2]. During the early phase of this research topic, authors usually selected exemplary algorithms, e.g., a beamformer, and investigated how to distribute the computational task on a network of nodes. Investigated algorithms include distributed Kalman filters [3, 4], distributed Delay-and-Sum Beamformer (DSB) [5], distributed Minimum Variance Distortionless Response (MVDR) beamformer [6], acoustic source localization [7] or generalized sidelobe canceller [8].

In later publications, more general approaches were presented which might be less task-specific and thus easier to adapt to other problem statements. Bertrand et al. presented the class of Distributed Adaptive Node-specific Signal Estimation (DANSE) algorithm for Minimum Mean Square Error (MMSE) estimators [9, 10]. A distributed Expectation-Maximization (EM) algorithm was discussed in [11] and distributed compressive sensing is the topic of [12], just to name a few signal processing techniques that could be useful in a variety of scenarios and tasks.

However, these strategies all target the same idea: First they estimate parts of the problem locally and, subsequently, try to find a global optimum by averaging across the local results. For this task, decentralized algorithms like gossipping [13] can be applied. Another interesting idea for distributing an estimation problem was presented by Vlachos et al. in [14] where a correlation matrix was locally estimated in parts and the missing parts where reconstructed.

All publications mentioned above try to optimize their approaches to work on a specific hardware at low data rate, limited memory and computational power. Some consider data rate-preserving strategies, e.g., [9], or transform a bandwidth-consuming task (signal distribution) into a computation power-consuming one (matrix reconstruction), e.g., [14].

The ongoing progress in hardware development for embedded hardware reduces the need for algorithm optimization, since every generation offers more performance at reduced costs. Power consumption is out of our scope, because these devices are normally power plugged.

Recent examples are neural network-empowering chips, like the Tensor Processing Units (TPUs) from Google or the Bionic SoCs from Apple. They will enable the local processing of large neural networks on embedded devices.

But in a large network, we still have the question of which node will be running which part of the distributed algorithm.

In this paper, we hypothesize that the algorithm is implemented in a block-oriented fashion (e.g., [15]), as is common practice for many online acoustic signal processing algorithms. The blocks may perform only small parts of the algorithm with standard operations, e.g., a Fast Fourier Transform (FFT), or even bigger algorithm parts like a gradient descent filter update. Hence, we look now at distributing the blocks on the networking nodes.

Our framework "Multicast Aware Routing for Virtual network Embedding with Loops in Overlays (MARVELO)" does not split an algorithm into blocks, so that the degree of freedom is defined by the given blocks. Therefore, all blocks should be kept as small as possible.

After distributing the algorithms, another question rises up: which route should be taken to forward the data between the nodes. In modern network architectures, Software Defined Network (SDN) changes the network behavior when it comes to taking routing decisions. In its simplest definition, SDN relies on a controller (server), which receives all the networking updates (e.g., bandwidth utilization) from the networking nodes. Then, the server builds a forwarding table and sends it to the nodes, so that they know to where the data needs to be forwarded.

SDN has been introduced to Wireless Sensor Networks (WSN) in Sensor OpenFlow [16], Software Defined Wireless Networks (SDWN) [17] and TinySDN [18]. However, they were all limited to the routing functionalities and did not support algorithm distribution.

SDN-WISE [19] is an operating system framework that combines the SDN and algorithm distribution concepts. It focuses, however, on networking applications (e.g., firewalls or proxies) and considers routing only for auto-configuration purposes.

Our MARVELO framework is a middleware that combines SDN and algorithm distribution, so that it jointly considers the allocation of functions and the routing decisions. Moreover, it supports various types of applications. We focus here on WASN applications.

The paper is organized as follows: The MARVELO framework is described in Sec. 2, where the focus is set on the integral parts Monitoring (2.1), Management (2.2), Communication (2.3) and preparing an existing algorithm to work with MARVELO (2.4). Our hardware platforms for WASNs are presented and discussed in Sec. 3 and we conclude the paper with some experiments in Sec. 4 and a brief conclusion in Sec. 5.

2 Software Framework

There are two main roles in the MARVELO framework: controller and client (see Fig. 1). A controller is hosted either on one of the nodes or on a special device connected to the network, (e.g., a gateway to an external network or a large processor). The controller's responsibilities include monitoring and management of the network.

Within the network, the controller distributes the algorithm's code (i.e., blocks) and the accompanying data (e.g., models, parameter files, etc.) required for executing the code. Additionally, it collects log files for debugging purposes and controls the process life cycle (e.g., initialization, start, stop). Moreover, the routing decisions are taken by the controller (actual traffic forwarding of course takes place at the clients).

Each client hosts a single "Daemon" process, whose main task is to process the algorithm blocks and forward their outputs

to the next blocks in the processing queue, as defined by the controller. Hence, they are responsible for performing the algorithmic work on the nodes.

Algorithm blocks are encapsulated in independent processes, enabling parallel processing of multiple blocks controlled by the Linux kernel scheduler.

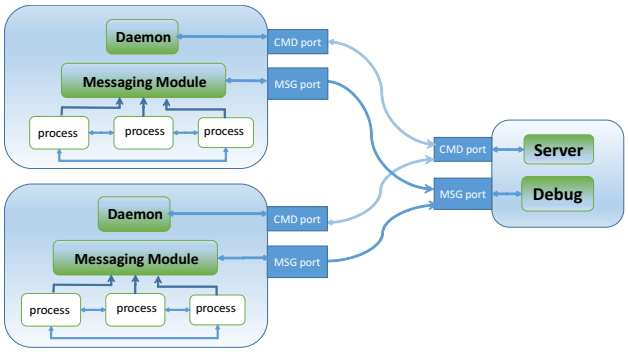


Figure 1: Software architecture for the monitoring function: Example with server and two clients.

2.1 Monitoring

Communication between controller and clients (Fig. 1) takes place through two different ports. The *CMD* ports are exclusively used for internal communication between controller and clients. For example, commands are sent through these ports to start/stop a process, and to check reachability of other clients in the network. Additionally, it can be used to collect the system status of the clients. The *MSG* ports are used to monitor the progress of each processing algorithm and visualize their terminal outputs, if this is required for debugging or supervision.

Messages from all processes running on a given node are collected by a local *Messaging* module. It forwards the messages through its *MSG* port to the controller, which can display all debugging logs using a special *Debug* module.

2.2 Management and Configuration

Process management and algorithm configuration are based upon a single Extensible Markup Language (XML) file (compare example in Fig. 2). It is a machine and human-readable representation describing the distribution of the blocks. The XML file specifies the individual nodes on which the processing blocks of the algorithm are hosted, defines the routes between the blocks (i.e., inputs and outputs) and states parameters of the blocks (e.g., number of channels).

The root element of the XML document is `network`. It contains multiple children elements of type `node`; each representing a single hardware device defined by the attribute `pi_id`. This attribute can either be an IP address or a hostname.

Each `node` has one or multiple children elements of type `algorithm`. They define and parametrize the *processes* for encapsulating single processing blocks, including the information from where the block receives the input, and to where it will send its output. Furthermore, the attributes `path` and `executable` are set to define the binary to be executed and where to find it.

The `path` attribute refers to a local folder on the controller containing all data required for processing the block, including the binary (attribute `executable`) and all additional data. Note that MARVELO automatically sends all files and folders including all sub-folders to the respective nodes, so a user does not have to take care about how to push the data to the nodes for synchronization.

A child element `algorithm` knows three optional children elements `input`, `output` and `parameter`. The latter can be used to specify any input parameter to be passed during the invocation of the binary (e.g., string or int), whereas the chronological

```
<network>
 <node pi_id="10.0.1.1">
  <algorithm path="record_path"
        executable="./recording">
     <output target_pi_id="10.0.1.2"
          pipe_id="1"/>
     <parameter param="--channels 2"/>
  </algorithm>
 </node>
 <node pi_id="10.0.1.2">
  <algorithm path="beamformer_path"
        executable="./beamforming">
   <input source_pi_id="10.0.1.1"
        pipe_id="1"/>
   <parameter param=" -l res.log "/>
  </algorithm>
 </node>
</network>
```

Figure 2: Configuration file example for a two node network, where each nodes hosts a single process.

order is retained in case of several `parameter` elements. The `input` element has an attribute `source_pi_id`, which specifies the node from where the input is received. The attribute has the same node's `pi_id` if the input is received from a process running on the same node. Additionally, the attribute `pipe_id` is used to define the pipe between data sending and data receiving processes. A corresponding function has the attribute `output`, where `target_pi_id` specifies the destination's `pi_id` and `pipe_id` needs to be consistent with the `input` elements attribute `pipe_id` of the process running at the destination.

```
input_pipe = ParseArgument()
output_pipe = ParseArgument()
Param1 = ParseArgument()

while (Condition == True)
  inputData = read(input_pipe)
  result = runFunction(inputData,Param1)
  write(output_pipe, result)
end
```

Figure 3: Pseudocode of a process: Intialization method for parsing comand line arguments and block oriented data processing method.

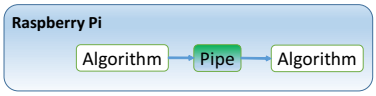
2.3 Communication

A *process* wraps an algorithm with networking parameters for data forwarding, while the *Daemon* module is responsible for the connections between *processes*.

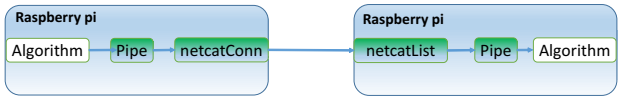
In Sec. 2.2 we show how to manage and configure a *process*. Accordingly, when writing the code for a *process* (Fig. 3), input and output pipes need to be declared and opened for enabling the data transmission between the *processes*. A *process* is created by the local *Daemon* by executing the `executable` followed by the `parameter` attributes and the pipe file descriptors. So each block has to implement a parser for getting the input and output file descriptors that corresponds to the *process* related pipes. Examples on how to parse arguments can be found in [20] (C language) and [21] (Python) and follow standard best practices. Then, we read the input data from the input pipes, pass it to the *process*'s internal function executing the actual algorithm (i.e., "runFunction" in Fig. 3), and write the results to the output pipe.

A communication can either take place between *processes* on

the same node (Fig. 4a) or between *processes* on different nodes (Fig. 4b). Node's internal communication is efficiently realized by pipes, whereas inter-node links require networking techniques. In the later case the pipe's output is forwarded via TCP/UDP connections to communicate with *processes* on other nodes. The specialized modules `netcatConn` (send data) and `netcatList` (listen for data) employ the Linux `netcat` utility to realize these functionalities. The decision whether a `netcat` connection is needed or not is taken by the *Daemon* module in the background, without the need of a user intervention.



(a) Algorithms running on the same Raspberry Pi



(b) Algorithms running on different Raspberry Pis

Figure 4: Communication between different Algorithms.

2.4 Executing Algorithms with MARVELO

In this section, the previous mock up example is used to explain how MARVELO can be controlled by the user via the server module. Lets consider the XML example in Fig. 2 which states a WASN consisting of two nodes at the IP addresses "10.0.1.1" and "10.0.1.2". At first, the desired XML configuration (example.xml) has to be defined and an initial connection to the nodes should be established. This is accomplished by invoking the first two of the following commands in the server shell.

```
1 setxml example.xml
2 connect
3 transferdata
4 start
```

Once the nodes are connected, we proceed with distributing the files. On the controller, there exist two folders `record_path` and `beamformer_path`, where the executables `recording` and `beamforming` are respectively placed. Both processes have additional parameters. `Recording` is parametrized with the parameter `channels` that is set to 2 in this case, while the processs `beamforming` uses the parameter "-l" to save its output in a log file called `res.log`.

The command *transferdata* initiates the data transfer to the nodes, and *start* triggers the *Daemons* to start the signal processing by invoking the respective executables. Automatically a pop-up window will appear to show the output of the processing algorithms. A simple example for testing the framework can be found on our website [22].

MARVELO can also auto-generate the XML file, if the available resources (nodes & bandwidths) and resource requirements by the algorithm blocks are defined. To this end it solves a system of linear equations to jointly allocate the function and forwarding decisions (see [23] for details). In the current expansion state of MARVELO the requirements are manually configured, but we are in the progress of updating the framework so that these parameters are auto-collected.

3 Hardware for Sensor Nodes

The hardware design process for WASN has to incorporate a wide range of properties and capabilities, including aspects like power consumption, form factor, costs, computational power, memory and sensor equipment [24]. These aspects are stated either by the intended application scenario or the algorithms to be running on the hardware. While some characteristics state stringent limitations, e.g., the bandwidth in underwater scenarios [25], others may be relaxed by software solutions, e.g., reducing the overall power consumption by putting devices periodically asleep [26] or optimizing protocols for Medium Access Control (MAC) [27] or routing [28].

One central issue of WASNs remains often unconsidered in hardware design: Time synchronization [29]. Due to the fact of a missing centralized clock signal all devices of a WASN sample data with different rates and start offsets. If a hardware integrated solution (see [30] for example) is not possible data streams have to be resampled in software for synchronization [31] or algorithm selection becomes restricted to Sampling Rate Offset (SRO) independent ones.

In the following subsections we present two compact sensor nodes on Raspberry Pi^3 Model B+ basis. One uses an existing soundcard interface and the other is a completely new development targeting the SRO issue.

3.1 Raspberry Pi – Audio Frontend

The Raspberry Pi^3 Model B+ platform offers computational power (ARM A53 Core, 1.4 GHz) at a compact form factor and reasonable costs. Its widespread application in smart home environments has forced the development of multi-channel soundcards, e.g., the Octo soundcard from Audio Injector [32].

We developed a pluggable analog frontend (see Fig. 5, blue Printed Circuit Board (PCB)) hosting a circular microphone array with 6 synchronously sampled audio channels. The Octo soundcard is accessible via the ALSA interface and the required drivers are already integrated in many Linux distributions.

The Raspberry Pi^3 Model B+ (and also the Pi^2 and Pi^3) employs a BCM2837 chip for offering a I^{2}S interface. This interface is limited on chip-side to stereo audio [33]. To offer more than two channels the developers of [32] utilizes a Field Programmable Gate Array (FPGA). The 8 channel Time-division Multiplexing (TDM) audio signal from the Analog-to-Digital Converter (ADC) is rearranged by the FPGA to a pseudo dual channel audio signal at four times higher sampling rate. After receiving the data in the Kernel space of the Raspberry Pi the signal is converted back to an 8 channel audio signal.

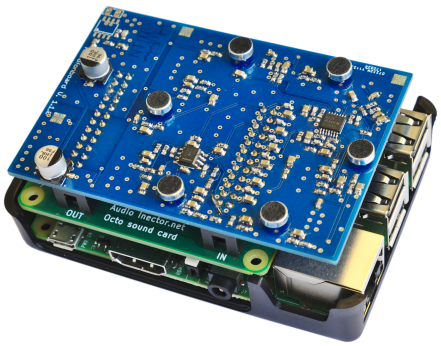


Figure 5: Raspberry Pi^3 with mounted Audio Injector soundcard and analog frontend (blue PCB).

3.2 Raspberry Pi – Quad node

The analog frontend from Sec. 3.1 is a 6 channel soundcard with a convenient audio quality. However, the SRO problem remains unsolved on this platform, as the crystal oscillator is neither configurable at ppm precision nor is the sampling process observable in detail.

Hence, we designed a simple pluggable card, called the Quad node (see Fig. 6), with a configurable Any Frequency Oscillators (AFO) (Silicon Labs Si514) driving the sampling process. Its frequency resolution is configurable with a precision of 0.026 ppb. The ADC chip from Texas Instruments (ADS 1274) samples 4

channels synchronously at 24 Bit and up to 144 kHz. On board we placed four omni directional condenser microphones at 4 cm distance. Audio data is exchanged by a Serial Peripheral Interface (SPI) interface, while the Si514 is controlled via the I^2C interface.

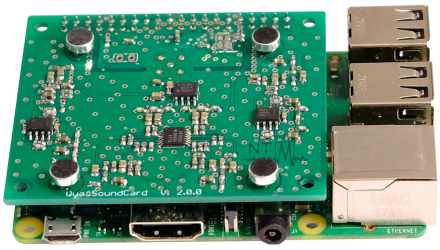


Figure 6: Raspberry Pi with mounted quad node.

3.3 Open Framework & Hardware

All schematics, PCB layouts and Gerber files for the Raspberry Pi based sensor nodes are available from our website [34]. Feel free to modify, enhance or change our hardware to your custom needs.

The MARVELO framework and an installation guide on how to setup a WASN consisting of Raspberry Pi^3 Model B+ is available on our project websites [22].

4 Experiments

We evaluate our framework on a wireless 4 nodes (***A***, ***B***, ***C***, ***D***) Raspberry Pi network using a Python implementation of the sampling rate offset estimator [15], which is depicted in Fig. 7. The algorithm's implementation does not reach real time processing performance on the Raspberry Pi platform, but it is a compact example to experimentally study MARVELO.

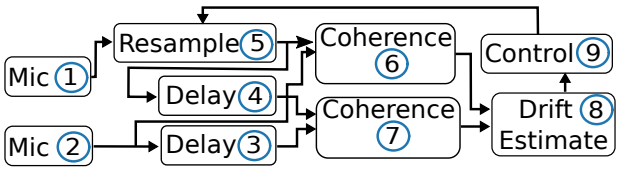


Figure 7: Block diagram example from [15].

It is assumed that only nodes ***A*** and ***B*** are close to the desired source, hence, they are selected for recording in all scenarios. Furthermore, node ***D*** requested the SRO information for other tasks and has to be incorporated. Since the wireless signal between ***D*** and the recording nodes is weak (i.e., low wireless links capacities), it utilizes node ***C*** for forwarding high data rates. In all tasks, audio recordings of 25 s are processed.

We place the processing blocks as described by Tab. 1. Exemplarily, the third scenario is depicted in Fig. 8. The first three scenarios utilize the same processing blocks in different placements so that the blocks are connected via pipes, while the fourth runs a single bulk script on node ***D***. Hence, *Centralized-4* has to process the tasks sequentially while the others can process them in parallel.

In Fig. 9, we show the network and the end to end delays for each scenario. The former is defined as the time required to receive the first input at ***D*** from the senders, while the latter is the time required to process the whole recording.

Scenario	Node A	Node B	Node C	Node D
Centralized-1	1	2	-	3,4,5,6,7,8,9
Distributed-2	1,6,8	2,7	3,4,5	9
Distributed-3	1,5,4	2,3	6,7,8	9
Centralized-4	1	2	-	[15]

Table 1: Comparison of scenarios for distributed and centralized realizations.

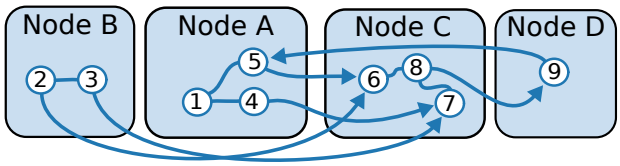


Figure 8: Visualization of scenario "Distributed-3".

We observe that the centralized scenarios (*Centralized-1* and *Centralized-4*) are subject to higher network delay, compared to distributed scenarios (*Distributed-2* and *Distributed-3*), as they need to forward the raw data in the network. Although both centralized scenarios execute the processing tasks on only one node, *Centralized-1* has a higher processing delay compared to *Centralized-4*. This is due to the higher interprocess communication overhead in scenario *Centralized-1* which, in our small example, outweighs the possible advantages from executing these blocks in parallel.

Regarding our distributed scenarios, they have a similar network delay, which is in all cases lower than that of the centralized scenarios. This is due to sending only the processed data (e.g., 1 Byte by block 8) compared to raw audio chunks (1 kB). On the other hand, they have different end to end delays. Scenario-2 shows a higher end to end delay (210 s), which is almost equal to that of *Centralized-4* (200 s). However, *Distributed-3* shows the best distribution in terms of network and end to end delays.

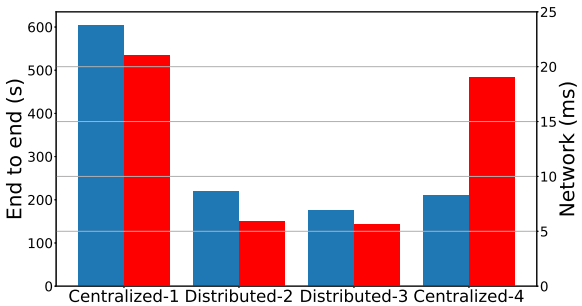


Figure 9: End to end (blue) and network (red) delays.

5 Conclusions & Outlook

The MARVELO software framework for signal processing is a flexible environment for hosting and distributing block-oriented implementations of algorithms on WASNs. Its simple approach reduces the needs for elaborated software modifications when an existing, block-oriented implementation is adapted. Furthermore, the presented open hardware can be customized to individual scenarios. Utilizing an AFO circuit enables handling synchronization issues at a high precision in hardware.

Next, we will enhance MARVELOs abilities to automatically profile and decide on distribution scenarios and we will add new features for network monitoring and online decision taking.

Acknowledgment

This work was supported by *Deutsche ForschungsGemeinschaft* (DFG) under contract no. KA2325/4-1 and SCHM 3301/1-1 within the framework of the Research Unit FOR2457 "Acoustic Sensor Networks".

References

[1] Z.-q. Luo, M. Gastpar, J. Liu, and A. Swami, "Distributed Signal Processing in Sensor Networks," *IEEE Signal Processing Magazine*, vol. 23, no. 4, pp. 14–15, 2006.

[2] A. Bertrand, "Applications and trends in wireless acoustic sensor networks: A signal processing perspective," in *2011 18th IEEE Symposium on Communications and Vehicular Technology in the Benelux (SCVT)*, pp. 1–6, Nov 2011.

[3] R. Olfati-Saber, "Distributed Kalman filtering for sensor networks," in *Proceedings of the IEEE Conference on Decision and Control*, pp. 5492–5498, 2007.

[4] F. Kirsch and M. Vossiek, "Distributed kalman filter for precise and robust clock synchronization in wireless networks," *RWS 2009 IEEE Radio and Wireless Symposium, Proceedings*, pp. 482–485, 2009.

[5] Z. Yuan and R. C. Hendriks, "Distributed delay and sum beamformer for speech enhancement in wireless sensor networks via randomized gossip," *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, vol. 22, no. 1, pp. 4037–4040, 2012.

[6] R. Heusdens, G. Zhang, R. C. Hendriks, Y. Zeng, and W. B. Kleijn, "Distributed MVDR Beamforming for (Wireless) Microphone Networks Using Message Passing," *Acoustic Signal Enhancement; Proceedings of IWAENC 2012; International Workshop on*, no. September, pp. 1–4, 2012.

[7] Y. Liu, Y. H. Hu, and Q. Pan, "Distributed, robust acoustic source localization in a wireless sensor network," *IEEE Transactions on Signal Processing*, vol. 60, no. 8, pp. 4350–4359, 2012.

[8] S. Markovich-Golan, S. Gannot, and I. Cohen, "Distributed multiple constraints generalized sidelobe canceler for fully connected wireless acoustic sensor networks," *IEEE Transactions on Audio, Speech and Language Processing*, vol. 21, no. 2, pp. 343–356, 2013.

[9] A. Bertrand and M. Moonen, "Distributed Adaptive Node-Specific Signal Estimation in Fully Connected Sensor Network;Part I: Sequential Node Updating," *IEEE Transactions on Signal Processing*, vol. 58, no. 10, pp. 5277–5291, 2010.

[10] A. Bertrand and M. Moonen, "Distributed Adaptive Node-Specific Signal Estimation in Fully Connected Sensor Networks – Part II: Simultaneous and Asynchronous Node Updating," *IEEE Transactions on Signal Processing*, vol. 58, no. 10, pp. 5277–5291, 2010.

[11] S. S. Pereira, R. Lopez-Valcarce, and A. Pages-Zamora, "A Diffusion-Based EM Algorithm for Distributed Estimation in Unreliable Sensor Networks," *IEEE Signal Processing Letters*, vol. 20, no. 6, pp. 595–598, 2013.

[12] J. Liang and C. Mao, "Distributed compressive sensing in heterogeneous sensor network," *Signal Processing*, pp. 1–7, 2015.

[13] M. Franceschelli, A. Giua, and C. Seatzu, "Distributed averaging in sensor networks based on broadcast gossip algorithms," *IEEE Sensors Journal*, vol. 11, no. 3, pp. 808–817, 2011.

[14] E. Vlachos and K. Berberidis, "Adaptive completion of the correlation matrix in wireless sensor networks," in *2016 24th European Signal Processing Conference (EUSIPCO)*, pp. 1403–1407, Aug 2016.

[15] J. Schmalenstroeer, J. Heymann, L. Drude, C. Boeddecker, and R. Haeb-Umbach, "Multi-stage coherence drift based sampling rate synchronization for acoustic beamforming," in *19th International Workshop on Multimedia Signal Processing (MMSP)*, IEEE, 2017.

[16] T. Luo, H. P. Tan, and T. Q. S. Quek, "Sensor openflow: Enabling software-defined wireless sensor networks," *IEEE Communications Letters*, vol. 16, pp. 1896–1899, November 2012.

[17] S. Costanzo, L. Galluccio, G. Morabito, and S. Palazzo, "Software defined wireless networks: Unbridling SDNs," in *2012 European Workshop on Software Defined Networking*, pp. 1–6, Oct 2012.

[18] B. T. de Oliveira, C. B. Margi, and L. B. Gabriel, "TinySDN: Enabling multiple controllers for software-defined wireless sensor networks," in *2014 IEEE Latin-America Conference on Communications (LATINCOM)*, pp. 1–6, Nov 2014.

[19] L. Galluccio, S. Milardo, G. Morabito, and S. Palazzo, "SDN-wise: Design, prototyping and experimentation of a stateful SDN solution for wireless sensor networks," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, pp. 513–521, April 2015.

[20] "Parsing Program Arguments, howpublished = `http://www.gnu.org/software/libc/manual/html_node/parsing-program-arguments.html`, note = Accessed: 2018-04-20."

[21] "Parser for command-line options, arguments and sub-commands, howpublished = `https://docs.python.org/3/library/argparse.html`, note = Accessed: 2018-04-20."

[22] DFG FOR 2457, "https://upb.de/asn/software," Homepage MARVELO, 2018.

[23] H. Afifi, S. Auroux, and H. Karl, "MARVELO: wireless virtual network embedding for overlay graphs with loops," in *2018 IEEE Wireless Communications and Networking Conference (WCNC) (IEEE WCNC 2018)*, (Barcelona, Spain), Apr. 2018.

[24] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer Networks*, vol. 38, no. 4, pp. 393–422, 2002.

[25] V. Chandrasekhar, W. K. Seah, Y. S. Choo, and H. V. Ee, "Localization in underwater sensor networks," *Proceedings of the 1st ACM international workshop on Underwater networks - WUWNet '06*, p. 8, 2006.

[26] T. Arici and Y. Altunbasak, "Adaptive sensing for environment monitoring using wireless sensor networks," *2004 IEEE Wireless Communications and Networking Conference (IEEE Cat. No.04TH8733)*, vol. 4, pp. 2347–2352, 2004.

[27] I. Demirkol, C. Ersoy, and F. Alagoz, "MAC protocols for wireless sensor networks: a survey," *IEEE Communications Magazine*, vol. 44, no. 4, pp. 115–121, 2006.

[28] J. Al-Karaki and A. Kamal, "Routing Techniques in Wireless Sensor Networks: A Survey," *IEEE Wireless Communications*, vol. 11, no. 6, pp. 6–28, 2004.

[29] J. Elson and K. Römer, "Wireless Sensor Networks: A New Regime for Time Synchronization," *Hot Topics in Networks*, vol. 33, no. October, pp. 28–29, 2002.

[30] J. Schmalenstroeer, P. Jebramcik, and R. Haeb-Umbach, "A combined hardware-software approach for acoustic sensor network synchronization," *Signal Processing*, vol. 107, pp. 171–184, 2015.

[31] J. Schmalenstroeer and R. Haeb-Umbach, "Efficient sampling rate offset compensation - an Overlap-Save based approach," in *Accepted for 26th European Signal Processing Conference (EUSIPCO) (EUSIPCO 2018)*, (Roma, Italy), Sept. 2018.

[32] Audioinjector, "http://www.audioinjector.net/rpi-octo-hat," Flatmax Studios, 2016.

[33] BCM2835, "https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/bcm2835-arm-peripherals.pdf," ARM Peripherals documentation, 2018.

[34] DFG FOR 2457, "https://upb.de/asn/hardware," Homepage Acoustic Sensor Networks Project, 2018.