XDense: A Mesh Grid Sensor Network for Extreme Dense Sensing

João Loureiro

Supervisor: Eduardo Médicis Tovar

Co-Supervisor: Raghuraman Rangarajan

Programa Doutoral em Engenharia Electrotécnica e de Computadores

January, 2019

Faculdade de Engenharia da Universidade do Porto

XDense: A Mesh Grid Sensor Network for Extreme Dense Sensing

João Loureiro

Dissertation submitted to Faculdade de Engenharia da Universidade do Porto
to obtain the degree of

Doctor Philosophiae in Electrical and Computer Engineering

President: José Silva Matos
External referee: Leandro Indrusiak
External referee: Jean-luc Scharbarg
Referee: Paulo Portugal
Referee: João Cardoso
Supervisor: Eduardo Tovar

January, 2019

Abstract

We introduce XDense, a novel sensor network system for application scenarios that could benefit from densely physically deployed sensors. More specifically, XDense was conceived for cyber-physical systems (CPS) that require real-time dense sensing, for example, involving hundreds of sensors per square meter in real-time.

We motivate our work by presenting CPS application scenarios that could potentially benefit of dense sensor networks, which are currently limited by available technology. Out of the different application fields we discuss, we give special focus to aviation. More specifically, we focus on active flow control (AFC) on aircraft wing surfaces. We aim at providing means of sampling data with a high spatial and temporal granularity about the air flowing through an aircraft wing, so that active control over the aerodynamics of the wing is feasible.

The XDense architecture consists of a wired 2D-mesh grid network that provides it distributed processing capabilities, that are used to enable real-time complex environmental data extraction in a distributed fashion. It resembles Networks-on-Chip (NoC) architecture, principles of operation and temporal behavior. The similarities and differences are discussed.

We detail XDense's node and network architecture, protocols, and principles of operation. We evaluate the performance of XDense in fluid dynamic application scenarios with extensive experiments on sensing and feature detection capabilities.

We also tackle the issue of time predictability of XDense. We present a methodology that uses traffic shaping heuristics to guarantee bounded communication delays while fulfilling memory constraints. We evaluate the model for multiple network configurations and workloads, and present a comparative performance analysis in terms of link utilization, queue size and execution time. With the proposed traffic shaping heuristics, we endow XDense with the capabilities required for real-time applications.

We also discuss the practical issues involved in implementing XDense and the steps for its experimental validation. A prototype node and a test-bed was implemented to validate our assumptions and to assess the performance capabilities.

Keywords: Real-time Embedded Systems, Real-time Communication, Traffic Shaping, Feature Detection, Distributed Computing, Dense Sensor Networks, Active Flow Control.

Resumo

Apresentamos o XDense, um novo sistema de rede de sensores para cenários que poderiam se beneficiar de instalações densas de sensores. Mais especificamente, o XDense foi concebido para sistemas ciber-físicos (CPS) que requerem operação em tempo real, por exemplo, para amostrar milhares de sensores por metro quadrado em tempo real.

Motivamos nosso trabalho apresentando cenários de aplicação dos CPS que poderiam beneficiar de redes densas de sensores, que atualmente são limitadas pela tecnologia disponível. Dos diferentes campos de aplicação que discutimos, damos especial atenção à aviônica. Mais especificamente, nos concentramos no controle de fluxo ativo (AFC) nas superfícies das asas das aeronaves. Nosso objetivo é fornecer meios de amostragem de dados com alta granularidade espacial e temporal sobre o ar que flui através de uma asa de aeronave, para que o controle ativo sobre a aerodinâmica da asa seja viável.

A arquitetura XDense consiste em uma rede de malha 2D, com fio, que fornece recursos de processamento distribuído, que são usados para permitir a extração de dados ambientais complexos em tempo real de maneira distribuída. Esta arquitectura se assemelha à arquitetura das Network-on-Chip (NoCs), aos seus princípios de operação e comportamento temporal. As semelhanças e diferenças são discutidas.

Nós detalhamos o projeto do XDense e a arquitetura de rede, protocolos e princípios de operação. Avaliamos o desempenho do XDense em cenários de aplicação de dinâmica de fluidos com experimentação extensa em monitorização e estração de detalhes de sinais físicos.

Também abordamos a questão da previsibilidade temporal do XDense. Apresentamos uma metodologia que utiliza heurísticas de modelação de tráfego de comunicação para garantir atrasos de comunicação limitados e o cumprimento dos requisitos de memória. Avaliamos o modelo para diferentes configurações de rede e carga de trabalho e apresentamos uma análise de desempenho comparativa em termos de utilização de "link", tamanho de fila e tempo de execução. Com a heurística de modelagem de tráfego proposta, nós possibilitamos aplicações em tempo real no XDense.

Também discutimos as questões práticas envolvidas na implementação do XDense e as etapas para sua validação experimental. Um nó protótipo e um banco de testes são implementados para validar nossas suposições e para a medição de desempenho.

Keywords: Sistemas Embarcados em Tempo Real, Comunicação em Tempo Real, Modelagem de Tráfego, Detecção de Recursos, Computação Distribuída, Redes de Sen-

sores Densos, Controle de Fluxo Ativo.

Acknowledgments

First and foremost, I would like to thank my supervisor, Eduardo Tovar for his support during my PhD. Despite his busy schedule as the director of CISTER, he was always available to advise. I am thankful for the opportunity to conduct my PhD in such prestigious laboratory.

I thank Raghuraman Rangarajan for his guidance during my PhD. He was very helpful on the development, maturing and consolidation of ideas. His support in writing was also fundamental. It helped me greatly to develop a critical sense of scientific writing.

I express my gratitude to Borislav Nicolic, Shashank Gaur, Vikram Gupta and Vicent Nélis, who were available to discuss and develop ideas. Pedro Santos also helped greatly with the programming tasks related to the development and experimentation with the hardware prototypes. It was a pleasant experience to co-work with them. Their knowledge and advises were clearly beneficial to my learning and overall PhD experience. Several contributions in this Thesis resulted from collaborations with other researchers in CISTER, to which I am grateful.

I thank to Inês and Sandra who contributed in administrative and everyday bureaucracy.

I thank my family for their support, and for understanding my absence in important moments due to our physical distance. Finally, and most importantly, I would like to thank my beloved wife Sandra Montes, for her support, encouragement, patience and unwavering love. I dedicate this work to my son Gabriel, who was born meanwhile, and brought the most joyful moments of my life.

João Loureiro

[1]This work was supported by CNPq (Brazilian National Council for Scientific and Technological Development) under the PhD grant 201176/2012-2.

Publications related with the Thesis

E. Tovar, N. Pereira, I. Bate, L. Indrusiak, S. Penna, J. Negrão, J. C. Viana, F. Philipp, D. Mayer, J. Heras et al., "Networked embedded systems for active flow control in aircraft," in Proceedings of the 11th International Workshop on Real-Time Networks (RTN 2012). 10, Jul, 2012. Pisa, Italy.

J. Loureiro, N. Pereira, P. Santos, and E. Tovar, "A sensing platform for high visibility of the datacenter." in Proceedings of the 4th International Workshop on Networks of Cooperating Objects for Smart Cities 2013 (CONET/UBICITEC 2013). 8, Apr, 2013. Philadelphia, PA, U.S.A.

J. Loureiro, V. Gupta, N. Pereira, E. Tovar, and R. Rangarajan, "XDense: A sensor network for extreme dense sensing," Proceedings of the Work-In-Progress Session at the 2013 IEEE Real- Time Systems Symposium - RTSS, pp. 19–20, 2013.

J. Loureiro, N. Pereira, P. Santos, and E. Tovar, "Experiments with a sensing platform for high visibility of the data center," in Internet of Things Based on Smart Objects, ser. Internet of Things, G. Fortino and P. Trunfio, Eds. Springer International Publishing, 2014, pp. 181–198.

N. Pereira, S. Tennina, J. Loureiro, R. Severino, B. Saraiva, M. Santos, F. Pacheco, and E. Tovar, "A microscope for the data center," International Journal of Sensor Networks, 2015.

J. Loureiro, R. Rangarajan, and E. Tovar, "Demo abstract: Towards the development of XDense, a sensor network for dense sensing," Poster presented in 12th European Conference on Wireless Sensor Networks (EWSN 2015). 9 to 11, Feb, 2015, pp 23-24. Porto, Portugal.

J. Loureiro, R. Rangarajan, and E. Tovar, "XDense: A dense grid sensor network for distributed feature extraction," 33o Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2015) - Workshop de Comunicação em Sistemas Embarcados Críticos (WoCCES), 2015.

J. Loureiro, M. Albano, T. Cerqueira, R. Rangarajan and E. Tovar, "A module for the XDense architecture in ns-3," Demo in Workshop on ns-3 (WNS3 2015). 13 to 14, May,

2015. Castelldefels, Spain.

J. Loureiro, R. Rangarajan, and E. Tovar, “Distributed sensing of fluid dynamic phenomena with the XDense sensor grid network,”Proceedings of the IEEE International Conference on Cyber Physical Systems, Networks and Applications (CPSNA’15). 19 to 21, Aug, 2015. Hong Kong, China.

J. Loureiro, R. Rangarajan, B. Nikolic, L. Indrusiak, and E. Tovar. 2017. “Real-time dense wired sensor network based on traffic shaping”. In proceedings of the 23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). 16 to 18, Aug, 2017. Hsinchu, Taiwan. 1–10. https://doi.org/10.1109/RTCSA.2017.8046307

J. Loureiro, P. J. Santos, R. Rangarajan, E. Tovar, “Simulation Module and Tools for XDense Sensor Network”. In proceedings of the Workshop on NS-3 (WNS3’17). 13 to 14, Jun, 2017, pp 110-117. Porto, Portugal.

R. Robles, J. C. Viana, J. Loureiro, J. Cintra, A. Rocha, E. Tovar, “Active Flow Control using Dense Wireless Sensor and Actuator Networks”. In proceedings of Microprocessors and Microsystems: Embedded Hardware Design (MICPRO), Elsevier. 2018, Volume 61, pp 279-295.

Loureiro J, Rangarajan R, Nikolic B, Soares Indrusiak L, Tovar E. “Extensive Analysis of a Real-Time Dense Wired Sensor Network Based on Traffic Shaping”. ACM Transactions on Cyber-Physical Systems. https://doi.org/10.1145/3230872.

Contents

List of Figures

List of Tables

List of Abbreviations

FEUP	Faculdade de Engenharia da Universidade do Porto
AFC	Active Flow Control
SJA	Synthetic Jet Actuator
NS-3	Network-Simulator-3
MSE	Mean square error
CDF	Cumulative density function
TTS	Transmission time slots
DOR	Dimensional ordered routing
COTS	Commercial off-the-shelf
FIFO	First-in-first-out
ADC	Analog-to-digital-converter
SN	Sensor network
WSN	Wireless sensor network
COTS	Commercial-of-the-shelf
IP	Intellectual property
FIFO	First-in-firs-out
MEMS	Microelectromechanical systems
GPU	Graphic processing units
CPU	Central processing unit
LIDAR	Light detection and ranging
MAC	Media access control
BE	Best-effort
GS	Guaranteed service
MR	Multicast radius
MT	Multicast alternative
MA	Multicast area
BC	Broadcast
BC	Guaranteed service
MSE	Mean square error

FPGA	Field programmable gate array
μC	Microcontroller
RP	Routing protocol
CP	Communication protocol
AP	Application protocol

Chapter 1

Introduction

As a consequence of Moore's law, single embedded computers equipped with increasing processing, communication and sensing capabilities are tending to be minimally priced. This makes it economically feasible to deploy dense sensor networks with very large quantities of computing capable sensor nodes. Accordingly, it is possible to take a very large number of sensor readings from the physical world, perform computation on sensed quantities and take decisions from the results.

Very dense sensor networks can offer information about the physical world with greater spatial resolution. When associated with high temporal capabilities, such sensor networks can offer better opportunities in detecting the occurrence of dynamic events with high resolution on an observed natural phenomenon. This is of paramount importance for a number of applications with high spatial and temporal sensing resolution requirements. Some examples are: (i) airflow monitoring on aircraft wings [8]; (ii) fluid dynamic tests on under-water vehicles [9]; (iii) fine-grained structural monitoring [10]; (iv) biomedical devices for electroencephalogram [11]; (v) robotic e-skins [12]; (vi) health-monitoring wearable sensor networks [13] and others.

Such densely instrumented systems face serious scalability issues in many key aspects, such as: communication and processing time, predictability of time, interconnectivity, power requirements, reliability and cost [14]. Moreover, the need for high spatial and temporal resolutions are related, but are concurring requirements, which are difficult to address simultaneously.

Today's dense sensor deployments usually have each sensing element connected to a central digital processor, with multiplexed analogue inputs that allow reading each sensor individually [15]. This usually involves impractical complex wiring setups, in witch sensors are read at predefined time slots [16, 17].

Furthermore, collecting and processing data from a dense sensor network is costly

in terms of communication and computation. Feature extraction is challenging and very difficult to achieve in real-time, hence prohibiting real-time dense sensing and its use in real-time applications.

Despite the various applications of dense sensor networks mentioned, in this thesis we focus on fluid dynamic applications and its related challenges. More specifically, we focus on airflow sensing on aircraft wings for active flow control (AFC); a scenario that benefits from such dense deployments of cyber-physical systems.

In the next section, we give an overview of the importance of airflow monitoring for AFC in aviation and its spatial and temporal requirements. We also discuss the involved challenges on implementing a sensing system for AFC.

1.1 Motivation and Challenges

In order to understand the importance of AFC, we first have to understand the benefits it can have in terms of flight costs reduction to the growing aerospatial industry.[1]

Cost reduction is often associated with the reduction of fuel consumption, which is important because of both environmental effects and cost efficiency. It is known from the Breguet range equation [19] that improvements in aerodynamics, engines, and structure have major importance in reducing weight and drag of the aircraft. In fact aerodynamic drag due to skin friction is known to be one of the main factors contributing to increased fuel consumption. It constitutes approximately one half of the total drag for a typical long range aircraft at cruise conditions [20].

A significant part of the skin friction is due to turbulent airflow over the wing [21]. Turbulent airflow is composed of coherent structures of chaotic temporal evolution, such as vortices. This chaotic behavior causes an increase in interaction between the air and the wing (and the fuselage, in general), and consequently an increase in total skin friction [22]. Turbulent airflow is usually undesirable on aircraft as it increases drag and noise, and consequently fuel expended [23].

The main cause of the occurrence of turbulent airflow on an aircraft, is the flow separation from the wing's surface. The airflow tends to separate from the solid surface due to its high speed and high inertial forces, that exceed the viscous-elastic forces that keep it attached to the surface. This excess causes its detachment from the surface of the wing, and creates a low pressure zone under the airflow. This difference is pressure leads the

[1] According to the Airbus 2016-2035 Global Market Forecast, it is expected that until 2035 the world passenger traffic will nearly triple, and airlines will more than double their fleets of passenger aircraft (with over 100 seats) [18].

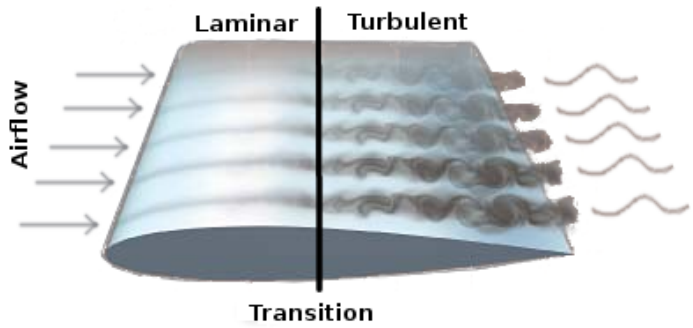


Figure 1.1: Example fluid dynamic application scenario: Airflow over a wing surface exhibiting transition from laminar to turbulent flow.

airflow into turbulent chaotic behavior as the air tends to spread and mix while flowing to the low pressure zone [24].

The region in which the separation initiates is called transition region. It is where the airflow transits from the laminar regime (with a more homogeneous speed profile distribution) to turbulent regime (with coherent structures of chaotic temporal evolution) [22].

In an aircraft, a high speed airflow over a wing can present both laminar and turbulent characteristics at the same time at different regions. Figure 1.1 shows an airflow over a wing surface, illustrating the transition region.

In order to minimize turbulent airflow, it is crucial to understand, distinguish, characterize and quantify the physical mechanisms taking place in the transition region. This is where the separation takes place and originate the turbulent airflow [25]. More importantly, in some applications, detecting where the transition takes place may be enough to enable efficient AFC.

Many studies have been conducted towards flow characterization and features extraction. A direct approach consists of developing sensor deployments to study the airflow by measuring its properties on-site, such as pressure, speed and temperature. However, in order to depict such phenomena with enough granularity, deployments of sensors with inter-space smaller than that of the spatial granularity of the observed phenomena may be required. That is for example, of 100 μm or less, with sampling rates in excess of 100 kHz. Figure 1.2 shows the spatial and temporal scales of coherent structures from different application scenarios.

Along these lines, in 2011, NASA proposed a design concept called “fly-by-feel” that also tries to address the challenge of dense sensing [26]. Their objective was to equip the next generation aircraft with a self-sensing capability via an integration of a dense network of sensors within the wing structure to achieve high spatial and temporal sensing resolutions.

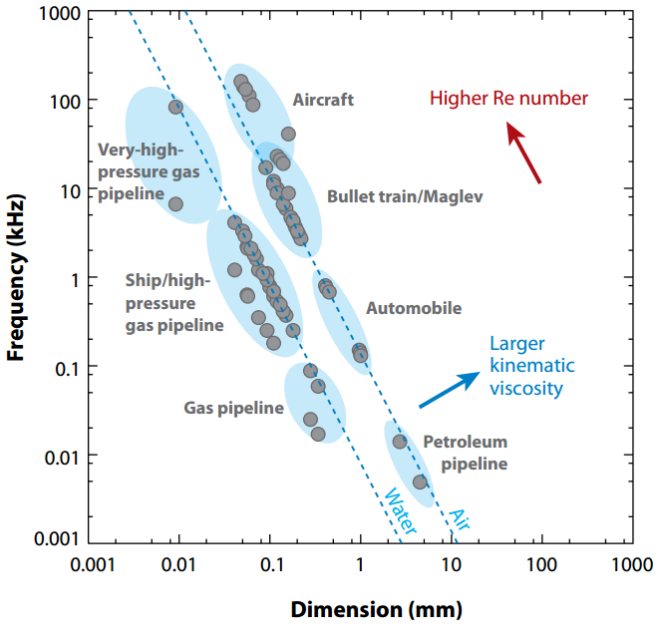


Figure 1.2: Spatial and temporal scales of coherent structures in various applications. Closed circles correspond to a 30 wall-unit length and 0.01 wall-unit frequency in different applications. Ovals represent different application groups. The figure is taken from [1].

A "fly-by-feel" system could measure the aerodynamic forces directly on the aircraft surfaces and use that information in a physics-based adaptive flight controls system to increase maneuverability, safety and fuel efficiency. In this way, the wing would be able to "feel", "think" and "react" in real-time for increased performance through autonomous control. Other advantages are also foreseen in [3], such as:

- structural complexity reduction by saving the efforts of large sensors installation;
- structural health monitoring through the embedded multi-functional sensor network;
- extreme conditions investigation and prevention for flight safety improvement;
- autonomous flight control for real-time optimum decision-making;
- maintenance cost reduction and structural design optimization.

Microelectromechanical systems (MEMS) are a technology with the potential to enable dense micro sensing, and are widely found in literature. More specifically to fluid dynamics, micro flow sensor arrays for AFC that try to meet such requirements are surveyed in [1].

However, as mentioned previously, such dense sensor deployments are challenged by the dynamics and complex nature of the data. Due to that, validations are usually done by individually connecting each of the many sensors to multiplexed channels of an analog-to-digital-converter (ADC), and data acquisition and analysis are limited to laboratory controlled conditions, with off-line data processing [27, 28, 29].

In order to achieve AFC, actuation technology is also required. A promising actuator technology that aims at doing that is the Synthetic Jet Actuator (SJA). SJAs run at key positions on the wing and continuously energize the low pressure zone by blowing jets of air into it. This action displaces the transition region towards the back of the wing, and consequently reduces the overall friction [30]. The weakness of SJAs is to *not* use sensors to detect and trace the turbulent flows and hence offer only open loop actuation. This compromises the efficiency of AFC, leading to waste of energy resources when there is no turbulent flow or when the turbulence lies outside the actuators' control field.

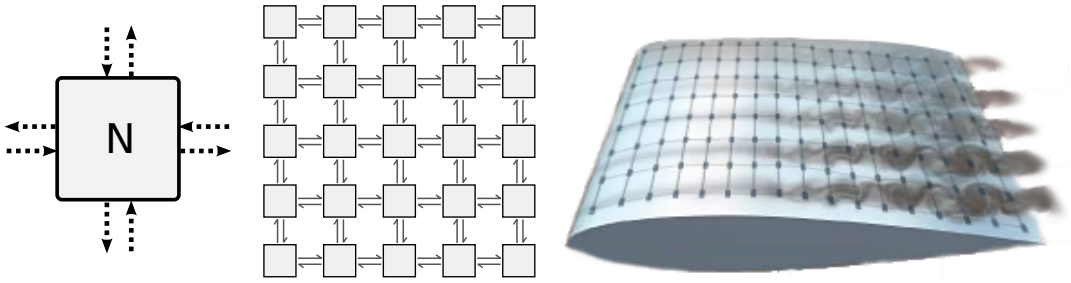
Morphing wings are another potential actuating technology to enable AFC, as it allows wings to actively change their shape in order to achieve efficient flight conditions. However, none of the proposed solutions perform closed-loop actuation through dense sensing [31]. Cattafesta and others survey these and other actuators for AFC [32].

The importance and challenges related to airflow sensing discussed, and the availability and low cost of computationally powerful COTS microcontrollers, have led us to believe in the potential of a custom design network architecture. Following, we elaborate the objectives of this research work and the proposed approach.

1.2 Thesis Statement

Considering the aforementioned requirements of AFC applications, we state our thesis as following:

By associating low latency mesh grid communication networks, with computationally capable sensor nodes, it should be possible to sense a phenomena and perform in-network distributed complex feature extraction in real-time. Such sensor networks should enable dense deployments that fulfill the spatial and temporal granularity required by AFC applications, while keeping low spatial and temporal complexity, minimally influenced by the large number of nodes on the network.



(a) (b) (c)

Figure 1.3: Proposed XDense network: (a) A single node with four ports in the four directions; (b) Example of a 5×5 mesh grid network; (c) Deployment envisioned for XDense.

1.3 System Requirements and Proposed Approach

Having introduced the application requirements and challenges, next we state the requirements of the sensor network targeted, which were considered while deciding on the system architecture:

(a) Efficient data extraction: The network infrastructure should allow efficient extraction of complex information about the phenomena. This should be done without the need of centralized data collection or processing. The network should provide mechanisms to allow distributed data exchange and processing for efficient feature extraction;

(b) Scalable infrastructure: The network infrastructure should allow scalability in terms of nodes count and density of deployments. An increase in sensor count should have minimal impact on the complexity of the architecture (spatial and temporal);

(c) Real-time behavior: Along with efficiency, the network infrastructure should also be able to respond with timeliness, so that actions can be taken in real-time based on the extracted data. Metrics such as end-to-end delay, response time, network load need to be considered in the design;

(d) Plug-n-Play: The network should accommodate different kinds of sensors. The system should also allow communication, data processing and high level data extraction algorithms to be co-designed with the application's requirements. Robustness and cost should also be taken into account.

In order to fulfill such system requirements, a custom designed network seems to be the most suitable solution. Therefore, we propose XDense: a sensor network tailored to address the challenges of e**X**tremely **Dense** sensor deployments for real-time applications.

The network uses a 2-D mesh grid architecture where each node is connected with four neighbors. Figure 1.3 shows (a) the XDense node and (b) network, along with the (c) envisioned deployment for airflow monitoring.

In order to enable efficient data extraction of the observed phenomena, XDense takes advantages of the mesh topology. That is, it exploits the available throughput and computational power (that scales with the number of nodes on the network) to extract data of interest, in real-time, without the need of collecting the data from each individual node centrally. Furthermore, because the network is based on regular structures, it allows dense and scalable sensor deployments.

The data of interest here are the airflow characteristics. More specifically, the feature we want to detect is the transition region, which is (as stated before), the region of the wing on which the airflow transits from laminar to turbulent regime. For this detection process, we use distributed feature detection and extraction algorithms to localize the transition region, by detecting the presence/absence of vortices that characterize turbulent/laminar airflow.

To detect the transition region, XDense performs local data sharing and processing operations in clusters of nodes before communicating pre-processed results to a central node, which is connected to a gateway link. By reducing the number of transmissions to a central node on the network (gateway), we achieve better network load distribution, decreased congestion and improved response time.

XDense differs from other sensor network (SN) architectures that also use nodes clustering with similar purposes. The main difference is that XDense was conceptualized and designed specifically to maximize opportunities on distributed processing for demanding real-time applications, keeping scalability and low system complexity as an important requirement. We state the following main differences from traditional wired/wireless sensor network approaches:

(a) Communication link: XDense uses point-to-point (P2P) links, which are not subject to concurrency for shared media, and are very little susceptible to noise issues when compared to shared buses and wireless star or multi-hop networks;

(b) Density: we consider a far more denser deployment scenario than traditional SNs (up to hundreds of nodes per square meter);

(c) Power constraints: XDense nodes may share power supply and the impact of communication on power is negligible, specially when compared to battery powered radio links on wireless networks;

(d) Communication rate: assuming wired links, rates are expected to be higher when compared with wireless links.

XDense builds a SN that behaves as if it is a distributed many-core computation platform, but programmed with specific sensing applications in mind. In a sense, it resembles Network-on-Chip (NoC) architectures. This is especially true regarding aspects like network topology (based on regular structures that compose mesh grids), routing schemes, timing properties and distributed computing capabilities [33]. On the other hand, it also differs from NoCs. The key differentiating aspects:

(a) The network is not on a single chip, but built on a larger surface that is physically attached to the phenomena of interest;

(b) Node count is greater than that for typical NoC applications;

(c) Input data is locally generated at each node by its sensor (which imposes different restrictions and opportunities);

(d) Distributed algorithms will have to be designed taking into account nodes location on the network;

(e) NoCs are likely to have communication links with higher throughput (as they tend to be parallel links), because of the lower cost of interconnecting nodes at such scales.

Moreover, the XDense topology is not limited to static layouts, protocols or applications. Quite the contrary, our design goal is to allow plug-n-play nodes and applications for a diversity of deployment scenarios.

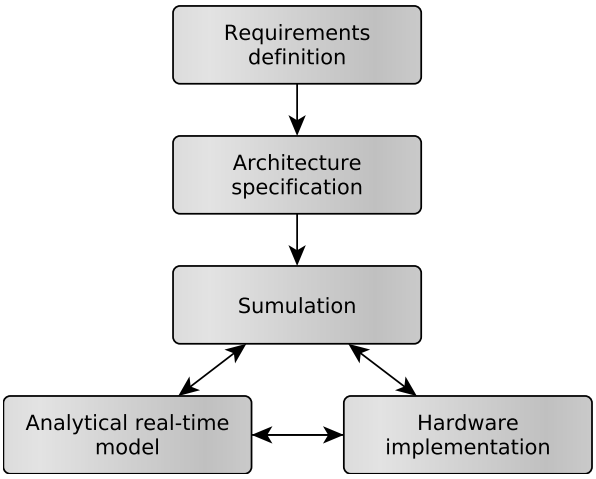


Figure 1.4: Tasks to be executed iteratively for the completeness of this research.

1.4 Methodology

To realize XDense, we elaborate on the methodology of this work by identifying a set of sub-tasks that should be executed in sequence in order to achieve our overall goal. Figure 1.4 outlines that.

In summary, we define the application requirements, specify the network architecture, protocols and its principles of operation. Having that defined, we enter into an iterative co-design process in order to develop and refine the simulation, analytical and implementation models.

Each of these tasks are summarized individually in the following paragraphs, that serves as an introduction to the next chapters in the thesis in which we address each of these issues in details.

Requirements Definition

An in depth understanding of the airflow phenomena is required to determine the sensing requirements in terms of magnitude, rate of change and granularity.

We do this with a detailed literature survey. We also survey the available network architectures which are suitable to dense deployments of nodes, and the protocols utilized. A special attention is given to 2-D mesh grid networks. Finally, we survey currently available dense sensing solutions, airflow sensing, visualization and feature detection methods.

Network Design

Taking into account the application requirements definition, we specify the network architecture. Its feasibility, performance and impact of different communication protocols, and data processing techniques are studied. We ensure that the specified network architecture and protocols are suitable to the hardware implementation.

Simulation Model

Having specified the network architecture and protocols, we develop a simulation model to test and debug the protocols, measure the performance and identify the limitations of the proposed network architecture. Our simulator also allows investigating distributed processing algorithms for feature detection applied on fluid dynamics data.

The simulation model also serves as a validation tool for any feature conceived during the development of the analytical model or hardware implementation.

Analytical Model

In carrying out the analytical model of XDense, we provide time guarantees and bounds on the resource utilization for real-time applications like AFC. The bounds provided are crucial to correctly dimension the hardware implementation.

The resulting model is validated using the simulation model, and its feasibility is demonstrated.

Hardware Implementation

We then implement and benchmark XDense nodes using different hardware in order to identify the potential performance bottlenecks associated with each implementation.

To close the design loop, the benchmark data is feed into the simulation model, in order to approximate the performance results from the simulation to the ones measured in the hardware.

1.5 Thesis Structure

The reminder of this thesis is structured as follows: In Chapter 2 we discuss the different technologies that inspired XDense, followed by a literature survey of dense sensor networks and their limitations. In Chapter 3, we review the theoretical background and state-of-the-art on techniques utilized for airflow features detection for AFC, as well as for real-time communication networks. Chapters 2 and 3 correspond to Part I (Background) of this Thesis. In Chapter 4, we detail the design of XDense by presenting its architecture, protocols and principles of operation.

Following, in Chapter 5, we present the proposed simulation model along with results from experiments with airflow feature extraction algorithms. In Chapter 6, we present our analytical model and traffic shaping heuristics, that enables real-time communication and provides memory bounds for XDense. We simulate various scenarios to evaluate the performance resulting from the heuristics proposed. In Chapter 7 we present the hardware implementation of XDense using commercial-of-the-shelf (COTS) hardware, along with benchmark results for various implementation approaches. Chapters 4-7 correspond to Part II (Proposed Novel Design: XDense) of this Thesis. Finally, in Chapter 8, we conclude this Thesis by providing a summary of the present research contributions and the future research plan.

Part I

Background

Chapter 2

Applications and Technology Enablers

2.1 Introduction

XDense is a sensor network inspired by modern networking and sensing technologies, which have demonstrated feasibility, benefits and performance. In this chapter we review these technologies, which, in different contexts, partially deal with the challenges that XDense tries to address.

We review 2-D mesh networks used on many-core processors and discuss their network topology, routing protocols, communication protocols, how they relate to XDense and how XDense can potentially benefit from some techniques used on such networks. We then review sensor specifically designed for airflow sensing, the networks that try to interconnect dense deployments of such sensors and their limitations. We then review some general purpose dense sensor networks that try to address the challenges related to dense sensing.

2.2 Related Network Architectures

2.2.1 Computation Arrays

Array Processors ware probably the first attempt to have a dense interconnect of computing elements. They were proposed in early 80's [34] and are widely found in literature.

In order to interconnect a large amount of computing nodes, different interconnection architectures have been utilized, either application tailored or for general purpose computing. In general, simple processing elements are set in an array, and more complex processing operations are done using pipelined computation; that is, input data enters through one extreme of the array from a host interface bus, and the result of the compu-

tation is delivered at the opposite extreme of the array. At each node traversed, simple computations are done by each node, until the final result is computed at the end of the array, where the result is collected.

The XDense network architecture resembles array processors regarding the large number of processing elements (nodes), and the distributed processing capabilities that we provide. This kind of architecture has the advantage of its modularity given by its construction based on regular structures. This leads to high performance, low cost and scalable systems.

A practical example of array processors arrays is provided in [35]. The authors propose grids of processors that use very large system integration (VLSI) technology to perform relational database operations directly in hardware. Each processing element was suited to compare tuples of data, then combined to perform complex database operations.

Another example of a computation array is the Geometric-Arithmetic Parallel Processor (GAPP). It is a processor with over 10,000 processing elements [36], in which each element can only perform simple processing such as one bit sum. They are interconnected to their nearest orthogonal neighbors to form a grid array such that it can perform matrix operations for image processing. It was used in military applications for tracking moving targets in real-time. It has proven performance thanks to pipelined computation.

The drawback of computational arrays is usually associated with the specificity of its architecture that requires custom solutions to address each specific problem. Therefore, computational arrays are not a good solution for general purpose computing, and were rapidly substituted by more modern and general purpose oriented architectures.

2.2.2 Many-core Systems and Networks on Chip

Since computation arrays were first proposed, the individual processing elements evolved and became highly complex, to become what is now called multi-core and many-core computing systems. The sharp increase in the number of cores, with much greater computing power within a single chip, lead to various new opportunities and challenges. One of the major challenges is related to the interconnection of cores, since their input/output capacity increased proportionally to their computing capacity.

Interconnecting multiple cores using shared buses is a common approach utilized to connect multiple cores with each other and with peripherals inside a chip. This is a suitable approach specially in cases where the data traffic is still low. For example, the AMBA bus by ARM [37] uses time division multiplexed (TDM) buses, and define a multilevel bus with a system bus and a lower-level peripheral bus for exchanging data with different purposes with different priorities. It was designed for custom silicon, to provide standard

bus protocols for connecting on-chip intellectual property (IP) components and custom logic. These bus protocols are independent of the processor and are generalized for systems on chip.

Even though shared buses offer good performance and timing predictability for multi-core systems, throughput is limited and do not scale with the increase of the number of nodes (cores and/or peripherals). Having multiple cores communicating though a shared bus can result in substantial increases in access time by each core due to concurrency and contention issues [38]. This simple fact imposes a reduced network performance as the nodes count increases. This is a limiting factor of shared buses, as it decreases opportunities on distributed data processing on systems with many nodes [38].

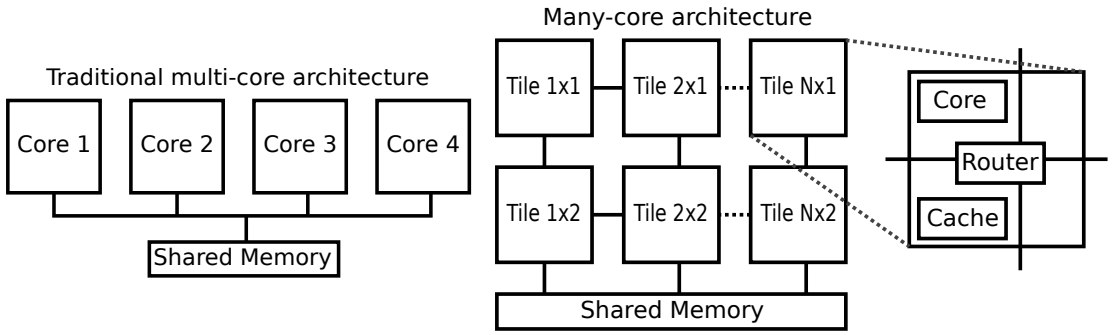


Figure 2.1: (a) Multi-core vs. (b) many-core systems.

Due to the limitations imposed by shared buses, such design paradigms had to be shifted towards more scalable interconnection schemes, which gave rise to NoCs [39], that aim at interconnecting even more numerous cores inside a chip and enable many-core processor architectures. Figure 2.1 compares both interconnections using (a) a shared bus and (b) a mesh network to interconnect many-cores.

For scalability issues and to keep low cost, many-core processors tend to use many of the same processing nodes, which form regular structures interconnected by the NoC. Each node is composed, for example, of a processor core, a private cache subsystem and a network switch. Each node is connected to its four neighboring nodes located in the four cardinal directions, thereby forming a 2D-mesh. The NoC provides a communication channel among the cores and other off-chip subsystems, for example a shared memory, in a multi-hop fashion. Figure 2.1(b) shows a typical NoC architecture.

Such many-core systems offer enhanced computational capabilities as compared to traditional multi-core platforms, as discussed in [38]. To list some: distributed processing capabilities; dynamic load balancing for reduced bottlenecks; fault tolerance; low power consumption.

Many-core processors are now becoming commercially available, as for example the Epiphany processor [40, 41]. It features up to 64 processors on a single chip connected through a high-bandwidth on-chip 2-D mesh NoC. Its NoC was designed to interconnect up to 4096 cores in a multi-chip arrangement. Each Epiphany core includes a small high performance floating-point RISC processor, a high bandwidth local memory system, and a set of builtin hardware features for networking. A distributed shared memory space allows the multiple cores to access their own, or neighbors data, trough load/store instructions using DMA. The Tile64 from Tilera, which is a 64 core processor [42], and the 48-core Single-Chip-Cloud computer [43] are just a couple of other examples of such many-core architectures.

XDense presents similarities with NoCs regarding the architecture and distributed processing capabilities, except that it is not confined into a single chip. This leads to different challenges regarding node's interconnection. That is because the physical distance between XDense's nodes are much higher when compared to the many-cores, preventing sensors nodes to be connected by wide, high bandwidth parallel links (as in NoCs). Instead, serial links tend to be more appropriate to interconnect XDense nodes (as commonly used in mid to long-range links). These issues lead us to further investigate possible alternatives.

Routing Protocols

Due to the limitations imposed by the serial links used by XDense, it is important to use the limited communication resources efficiently. This makes it important to correctly define routing and communication protocols to achieve low latency with low resource utilization. The choice of the algorithms can impact in memory requirements, power consumption, traffic utilization, performance, scalability, among others aspects. To choose the right algorithms, we analyze the trade-off between different available implementations approaches considering different aspects.

Routing algorithms can be classified as either static or dynamic. Static routing uses fixed paths for transferring data between two nodes in the NoC. Routers do not take into account the load on the network, neither avoid congested paths while taking routing decisions. Static routing algorithms are easy to implement and demand less hardware resources. Some of the most common static routing algorithms available are: XY routing, including X-first, Y-first and other variants [44]; pseudo-adaptive XY [44]; surrounding XY [45]; turn model (west-first, north-last, negative-first) [46]; ALOAS [47]; topology adaptive [48]. In [49] the authors survey many of the available static routing algorithms.

With dynamic routing, routing decisions are taken on-the-fly, according to the actual load on the network and availability of links. Because of that, different paths between two nodes can be taken, depending on the network state. In general, dynamic routing allows a more balanced and better use of the network resources, with the cost of non-determinism and overhead, which is added by the resource monitoring activities. There are many dynamic routing algorithms available, such as: minimal adaptive [50]; fully adaptive [50]; congestion look-ahead [51]; turnaround–turnback [52]; turnback when possible [53]; IVAL [53]; 2TURN [53]; odd–even [54]; and hot potato routing [55]. In [56] the authors survey these and other examples of dynamic routing algorithms available.

However, because of their constrained memory resources, most NoCs (including the ones mentioned in Section 2.2.2) use deterministic routing algorithms. Deterministic routing algorithms are also deadlock and livelock free [33], what make them suitable for real-time systems.

Communication Protocols

Communications protocols are also a significant aspect to consider in order to achieve a network with increased bandwidth, better resource utilization, better load balance, real-time behavior and good overall performance [39]. The trade-off between performance, power consumption and predictability has to be taken into account before the specification of a communication protocol.

Communication protocols have been extensively studied by numerous researchers. In general, implementations are specific to the network architecture, while trying to fulfill common application requirements, such as minimum bandwidth, real-time behavior or any other service guarantees, power consumption, cost, among others. In the following paragraphs, we review communication protocols which were taken into account while designing XDense.

A connection oriented communication protocol for NoCs, named SoCBus, was introduced in [57]. The authors introduce the concept of packet connected circuit, where a packet is routed through the network establishing a circuit as it travels. Their contribution cover the physical, link, network and transport layers mainly. Once the circuit is established, circuit switched networks enable low latency communication, minimally influenced by the distance of the two nodes communicating; latency is mainly dependent on contentions that may occur while setting up the circuit. Data is also guaranteed to arrive in the same order it was sent. However, the authors show that SoCBus is not suitable for general purpose computation platforms with random traffic patterns, because of the high probability of route blocking due to contention caused by already existing circuits. Long

distance circuits limit the maximum number of connections, saturating the network and decreasing the bandwidth maximum usage.

The wormhole protocol [58] is also widely applied on NoC-based many-cores. That is specially due to its good throughput and small buffering requirements. For its operations, a packet is divided into a number of parts called flits (flow control digits) for its transmission. Each flit has the size of the channel width, which transfers one flit at the time, in a single transaction, between two adjacent routers. Consecutive flits are pipelined after the header-flit (which governs the route) to form what is called a virtual channel. If along the way, a channel is found busy, the flow is blocked until the channel becomes available. During this period, all the flits are blocked through a process called back pressure, each at its corresponding channel. If the transmission is completed successfully, channels are released as the tail-flit passes through each channel. It is widely used in NoC-based many-cores due to its low latency, which is relatively insensitive to path length, and due to its reduced buffering (memory) requirements.

In [59] the authors proposed AEtheral. It is a complete architecture and implementation of a NoC, with guaranteed services such as uncorrupted, lossless ordered data delivery with guaranteed throughput and bounded latency. For services guarantees, they use resource reservation for the worst case, but also provide best-effort services to exploit the unused NoC capacity. Guaranteed services serve critical real-time applications traffic, while best-effort serve non-critical applications traffic.

The authors use contention-free routing, or pipelined time-division-multiplexed circuit switching, to provide such guarantees; both requiring the reservation of wires and buffers, in order to accomplish the guaranteed services. Nodes have to be synchronous, and each node uses a slot table to determine how to forward blocks of data arriving at one of the N ports and departing from one of the $N-1$ ports. The latency that a block incurs in every router it passes, is equal to the duration of a slot, and for guaranteeing bandwidth, multiple slots are reserved for multiple blocks transmissions. With this, there is no contention, because there is at most one input per output for each time slot, and the blocks can contain data only, improving the NoC efficiency.

Building the slots table is an optimization problem, which is done offline when designing specific applications. The resulting slot assignments are then programmed at runtime. However, if the connection requirements are only known at runtime, the design should use a distributed algorithm for randomly picking slots, or some other centralized algorithm, assuming limited runtime computational resources. Best-effort services uses conventional wormhole-routing. The packet header contains the path from the source to the destination.

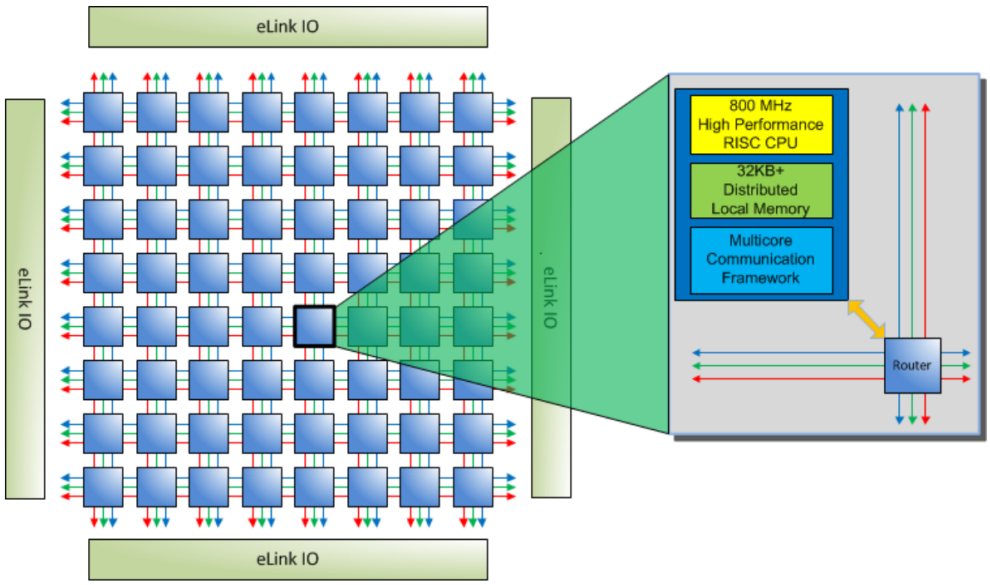


Figure 2.2: Epiphany processor internals. It shows 64 cores interconnected by a mesh grid NoC. Each node on the network consists of a router, CPU, Memory and communication framework.

2.2.3 Study Case: The Epiphany Processor

From the reviewed literature, the Epiphany NoC presents the most similar network architecture and protocols to the ones of XDense. Thus, we review its architecture routing and communication protocols closely, in order to substantiate the design decisions taken for XDense.

The Epiphany NoC consists of three independent 2-D mesh grid networks. Each of the three networks interconnect each node with their four immediate neighbors with a full duplex connection. Each node's router serve packets using a per-direction round robin arbiter, whereas each communication port has a single stage first-in-firs-out (FIFO) queues. Figure 2.2 shows the epiphany processor internals.

The network is packet switched, and packets are transmitted or forwarded in a single clock cycle, meaning that a single cycle latency is added per node as the packet travels. The network links supports up to 64 bits of data and 32 bits of address, per transmission and per cycle.

Each of the three meshes have different purposes: (i) Requests for data; (ii) carry data on-chip and (iii) carry data off-chip.

This architecture favors writes over reads, since reading a foreign address consists of sending a read request and waiting for the answer to arrive with the data requested. Writes, on the other hand, allow the node to send the data and continue processing while the data moves towards its destination (without busy waiting for any reply).

The two separate networks used for writing data are meant to decouple off-chip and on-chip communication. The intention of the designers was to make it possible to write on-chip applications that have deterministic execution times regardless of the types of applications running on neighboring nodes. From the application angle, the off-chip traffic is indistinguishable from on-chip one; apart from its lower bandwidth and higher latency.

The Epiphany NoC uses X-first static routing based on an absolute addressing system, that specifies an static global address to each memory address at each node. It is a deadlock free routing algorithm that works as follows. At every hop, the router compares its own address with the destination address. If the row addresses are not equal, the packet is routed to the east or west; once it reaches the correct row, but if the column addresses are still not equal, the packet gets routed to the north or south. When both column and row addresses match, the packet gets routed into the node. When multicasting, a different routing algorithm is used. In this case, the data is sent radially outwards from the transmitting node. Receiving nodes compare the destination addresses with their own, and in case it matches, the packet gets routed into the node. This feature allows writing to multiple nodes using a single transmission.

The off-chip write mesh network allows the NoC to expand among multiple chips through what they call eLink IO interface (see Figure 2.2), allowing it to scale to very large arrays; limited only by the address space. A 32-bit Epiphany architecture can scale up to 4,096 processors while a 64-bit architecture could scale up to 18 billion processing elements within a unified shared memory system.

To design the Epiphany NoC, the authors considered a number trade-offs while taking design decisions. We list the main ones regarding the topology, the switching mechanism, the routing algorithms and the flow control techniques:

- The 2-D mesh network topology was selected because of its simplicity, implementable on low complexity planar standard CMOS fabrication processes. Other topologies were considered by the authors, such as Torus wraparound, Butterfly and CLOS [60], but ruled out because of their complex circuitry and big footprint as they would have required twice as many wires per cross-section;

- Because 2-D mesh is a well studied topology, many numerical methods and signal processing algorithms have already been ported to (and can benefit from) processors based on this kind of NoCs;

- Packet switching with a global addressing scheme was chosen instead of a circuit switching, in order to provide implementation flexibility. For it to work, the complete destination address has to be transmitted at every transaction. This decision

may sound counter-intuitive, but in this way, the hardware implementation resulted simpler, what justifies the cost of the extra address bits;[1]

- Flow control uses extra pins to perform back-pressure to signal congestion on the route, to ensure that there are no packet drops. Protocols for QoS are not provided by the authors, and with the current architecture it is not possible to provide guarantees for real-time applications;

XDense has a lot in common with the Epiphany NoC, including the aforementioned design considerations. Specially concerning the 2-D mesh network topology and the packet switched communication protocols. The actual design differences result mainly as consequence of the difference in magnitude of the nodes and the resulting cost of interconnection. These differences are mainly the following:

- We use a single mesh, while compared to the three meshes used by Epiphany;
- We use serial full-duplex serial links to interconnect nodes (instead of wide parallel ones);
- Our nodes are connected to sensors that produce data locally;
- We provide more options of routing algorithms, which can be selected at run time;
- Our nodes are based on microcontrollers with much more per-node resources, including memory and processing power;
- Because we have reduced pin-count per node's microcontrollers, we do not use extra pins for flow control (what would imply an unacceptable added costs).

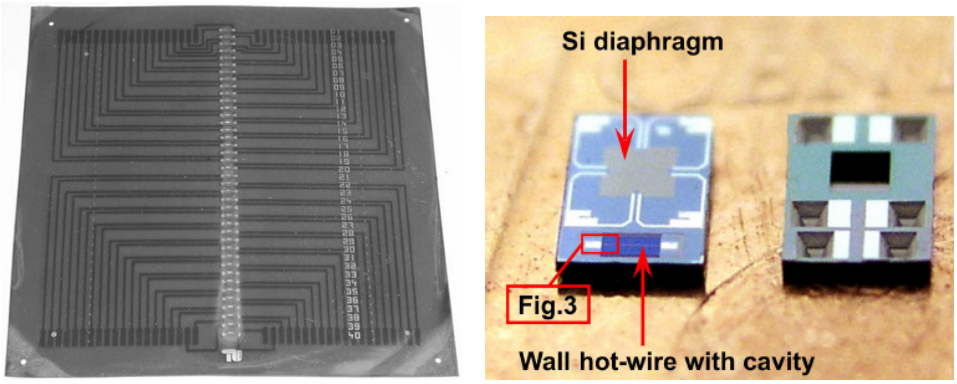
2.3 Dense Deployments of Sensors

2.3.1 Airflow Sensing

Passive Airflow Sensors

As previously discussed in the Introduction, a common goal in fluid dynamics is to perform active flow control on aircraft's fuselage in order to reduce the turbulent airflow and, consequently, drag forces, noise and energy waste [27, 23]. That is, an increase in

[1] That is because, at the on-chip scale, the relative cost of wires, drivers and registers are orders of magnitude lower as compared to larger scale systems (on board, off board and Ethernet connections for example).



(a) (b)

Figure 2.3: (a) A linear array of forty of the same MEMS sensors mounted on a polymer foil; (b) pressure sensor diaphragm and a hot-wire MEMS sensors on top of an Euro cent coin; (Figures taken from [2]).

the flight efficiency is desirable, and different opportunities to minimize turbulence with MEMS devices have been proposed [61]. Sensor [1] and actuators [32] have been actively researched, and had promising advances towards the realization of active flow control; although we focus mainly on the sensing part.

Among other requirements, sensors need to be easy to install, be non-invasive with respect to the observed phenomena, and mainly, be able to detect and estimate the flow state near the surface it is installed. To measure the flow state, sensors have to be able to directly or indirectly measure the flow properties, such as its pressure and shear stress distribution.

The main methods to measure shear stress are categorized as thermal, mechanical and optical. Thermal micro hot-wire shear stress sensor has been studied from the early stages of MEMS technologies [2]. In simple terms, it consists of indirectly measuring the shear stress by measuring the heat loss on a hot wire exposed to the airflow. For that, a control circuit keeps either the temperature, current or voltage across the hot wire at a constant value, and by means of measuring the variation on the current, voltage or temperature (the non-fixed variable) of the hot-wire, it is possible to indirectly measure the shear stress.

Good results were achieved with this kind of sensors, specially on their miniaturization and on-chip integration. In [2], for example, a MEMS sensor chip that combines a hot-wire flow sensor with a pressure sensor is presented. An array with forty sensors was obtained in an area of $70\times70\text{mm}^2$, with a sensor spacing of 1.5mm.

However, hot-wire sensors are limited by mainly two reasons: (i) the heat loss to the substrate can lead to wrong measurements; and (ii) it is impossible to measure the flow direction (and consequently the vortices orientation) with a single hot-wire.

In the same context, two consecutive European projects (called AeroMEMS and AeroMEMS II), aimed at engineering the integration and cost/benefit assessments of AFC using MEMS technology applied to improve the performance of wings, engine nacelles and turbo-machinery components. Prototype MEMS flow sensors and actuators were developed under the scope of those projects.

To address these limitations, in [29], the authors present a double hot-wire sensors which they called hybrid AeroMEMS sensor array. It is a combination of a flexible printed circuit board and a number of double and single hot-wire MEMS sensors with different setups. Each sensor features an area of $800\times600\mu m^2$. As an improvement over the single hot-wire mentioned previously, the doubled sensor allows measuring the flow direction. Tests were made in a wind tunnel demonstrating its applicability in determining flow speed and directions, showing efficacy on determining the flow characteristics. It also presented improved frequency responses thanks to numerical optimizations done during the design phase.

In the same project context, with a different approach, in [62] the authors described how they designed a high-resolution AeroMEMS sensor array for pressure distribution measurement. The sensor cell is composed by a diaphragm with a piezoresistor located on the edges of it. Measurements are taken indirectly, by measuring the longitudinal and transversal piezoresistance. A $2.5\times4.5\times0.3$ mm^3 chip was obtained. The sensing element consists of a a 900×900 μm^2 diaphragm with the appropriate sensitivity and frequency response of up to 160kHz. An array of 13 sensors was mounted on cylinder in a wind tunnel in order to compare the measurements with the theoretical expected values. It resulted in a fully functional and reliable sensor.

In [27] the same authors presented a sensor array for transient wall pressure and wall shear stress measurements. They deployed 154 sensors over the surface of a cylinder, in order to obtain extensive data about the wall pressure distribution and pressure fluctuations as well as transient pressure data on its surface. The sensor array is shown in Figure 2.3(a).

Previously, the same authors compared two designs, with two sensors each, in a 2x3 mm^2 chip area. They combined a piezoresistive pressure sensor with a diode for temperature measurement [63] or with an on-chip wall hot-wire for wall shear stress measurement [64]. The sensor chip described is shown in Figure 2.3(b).

In [65] the authors have developed another kind of shear stress MEMS sensor. It consists of a micro fence 300 μm high and 5 mm long, equipped with piezoresistors that stick in one of the fence sides. As the fence bends with the pressure due to the airflow, the resistance in the piezoresistor changes proportionally. It provides high resolution on measuring the shear stress, and its temporal resolution is up to 1 kHz.

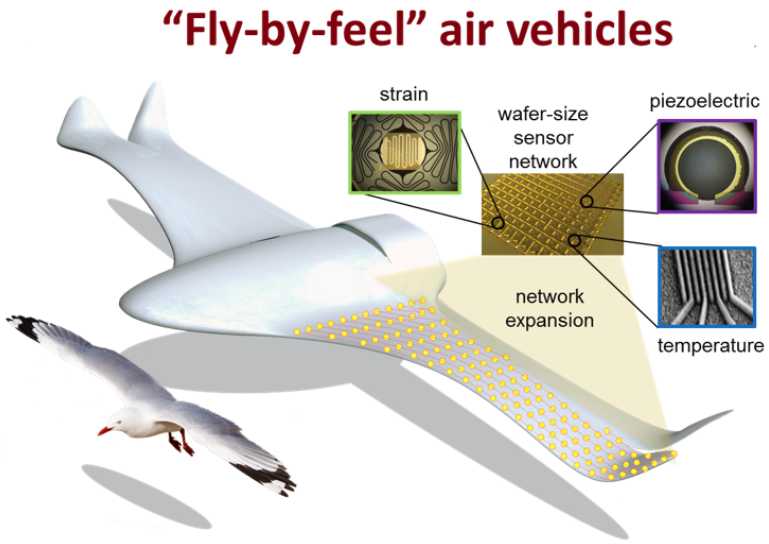


Figure 2.4: Schematic representation of an aircraft concept with a multi-modal sensor networks embedded inside the composite structural components to enable "fly-by-feel" (Figure taken from [3]

).

With a similar approach, in [66] and [67] the authors proposed micro-pillar MEMS sensor using doped-silicon piezoresistive strain gauges and an epoxy pillar for the wall shear stress measurement. Inspired by the hair cell or the lateral line of a fish, it consists of a micro pillar that bends in the direction of the flow, proportionally to the pressure imposed. The authors reported that water flows as low as 1 mm/s can be measured with their sensor.

In all the above solutions, to be tested, sensors are wired individually to analog-to-digital converters, which limits their use to laboratory conditions due to the complex setup requirements. To the best of our knowledge, despite the immense effort towards sensor development, no interconnection solutions is proposed.

Inspired by bird's feathers that can feel surrounding aerodynamic forces, NASA proposed a design concept called "fly-by-feel" for next generation aircrafts [26]. The basic concept is to equip aircraft's wing with a self sensing capability. This is an effort towards providing integrated sensing solutions for wings by enabling dense sensor networks for AFC. Figure 2.4 illustrates this concept.

Towards enabling "fly-by-feel", in [3] a novel approach was given to the problem. The authors introduced what they called self-sensing intelligent composite materials with state-sensing capabilities. Their objective was to develop aerospace intelligent structures that can sense the environmental conditions and their own structural state, and effectively interpret the sensed data to achieve real-time state awareness.

To enable that, sensor networks were designed to be embedded in the composite wing in order to provide it with sensing capabilities. Piezoelectric sensors were used to sense the vibration of the wing in order to identify the coupled airflow-structural dynamics, while strain gauges are used to determine the strain distribution of the wing and identify potential critical areas for the considered operating conditions. Stochastic signal processing and identification techniques are employed in order to accurately interpret the sensing data and assess the actual structural state.

The experimental evaluation and assessment of the intelligent composite wing is demonstrated via wind tunnel experiments for the identification of the airflow dynamics and investigation of the wing strain distribution. The authors successfully integrated the sensor network within the composite material of the wing. The results demonstrate as well the effectiveness of the data interpretation algorithms. More recently, in [68], the authors proposes data processing algorithms that use neuronal networks to identify the flight state in real-time. The authors believe that this research may enable new perspectives in developing intelligent self-awareness capabilities for the next generation of smart wings.

Despite the excellent results provided by the "fly-by-feel" initiative, the solutions presented rely on X-Y scanners[2] to read each sensing element individually. This approach presents similar limiting factors as pointed out earlier, which are mainly the following: it's fixed topology and read-out scheme, limits the network to the maximum sampling rate provided by the read-out circuitry, which do not provide opportunities for in-network data processing and complex feature extraction.

Active Airflow Sensors

Using a network to interconnect the sensors, in [70] and [71] the authors propose a human-like flexible array composed of numerous MEMS sensors capable of detecting multiple stimuli. To deal with the vast amount of data produced by the stretchable network of sensors, local processors are integrated into the network to perform local data processing. The network uses local processors to collect and process data from a small set of nodes around it, and a global processor to collect the data from the local processors, process it, and take decisions based on the collected data. Even though a multi-stage data acquisition scheme is used with local data pre-processing, the network topology and the way nodes are grouped is fixed.

A commercial solution for airflow sensing on airplanes is called Pressure Belt [72]. It consists of a strip of sensors mounted crosswise on the wing of an aircraft, with up to 254

[2] X-Y scanners are commonly used to scan matrix of sensing elements, as the ones widely used on multi-touch screens of tablets and cellphones [69].

sensors nodes. In one end, a coordinator sample and log the data from the nodes on an external data-logger situated inside the airplane. The network uses a full-duplex RS485 shared bus to interconnect nodes, and can communicate at up to 5 Mbps with packets of 48 bits each. The coordinator uses a clock synchronization scheme and time division multiple access (TDMA) for sampling the other nodes on the network.

Both the above approaches use active sensor, with integrated processors which are connected in a network. Despite that, because the network is based on shared buses, it does not provide opportunities to execute more elaborate algorithms for distributed data processing and feature extraction.

2.3.2 Other Dense Deployments of Sensor and Actuators

General Purpose Dense Sensor Networks

Not directly related to airflow sensing and AFC, different general purpose sensor networks have been proposed to address the challenges related to dense sensing.

A multi-modal sensor network with up to hundreds of sensors per square meter was proposed in [15]. The authors present a sensor network in which sensor nodes are based on microcontrollers. Nodes communicate using an infrared transceiver, which can send and receive data from its surrounding neighbors. One full-duplex serial port is used for its wireless link. The authors propose solutions for data management, localization, data aggregation and routing. Extensive experiments of their platform is presented in [73].

However, due to the wireless nature of the links (infrared), contentions and collisions substantially increase the cost of communication. Their research leans more towards wireless sensor networks whose performance does not suit the application scenarios we focus on.

Other researchers have extended this concept, and have proposed some alternative topologies in [74]. The network is based on a wired grid, master-slave communication scheme, which decreases the distributed processing opportunities due to shared links. The number of nodes on the network is also limited, therefore, it is not scalable.

Instead of wireless links, in [75], the authors use wired links to deploy a few sensors in a grid network, to act as an electronic skin. Even though nodes are deployed in a matrix, they are still interconnected using shared buses. More recently, the authors of [76] presented a modular dense sensor network in a form factor of a tape, tailored to wearables. Master-slave buses are used to interconnect nodes (through SPI or I2C [77]), regardless of the shape of the network.

Not only shared buses drastically decrease the opportunity for distributed processing due to their finite bandwidth that does not scale along with the number of nodes, but

they also constraint the number of nodes due to the limited address space and the related electrical limitations. They are therefore not a scalable solution.

Tactile skins

Electronic skins, or e-skins, are inspired by biological skins. These are multi-functional structures of great interest for robots and medical prostheses, in which sensors and actuators are closely integrated with microelectronic circuits. They should be flexible, stretchable, and robust devices that are compatible with large-area implementations and integrated with multiple functionalities. It represents a new kind of integrated electronics with a large area and flexibility ranging from electronic muscles [78] and textiles [79].

A survey of current available e-skins is provided in [80]. Authors comment on various different approaches, from organic stretchable electronics, optical capillary sensors and passive smart materials, including arrays of actuators. For example, a conceptual hardware architecture of skin-like circuits is presented in [79]. An elastomeric skin carries rigid islands on which active subcircuits are made. The subcircuit islands are interconnected by stretchable metalization by using stretchable conductors.

In [81] the authors designed a conformable tactile sensor skin. The skin is organized as a network of self-contained modules consisting of tiny pressure-sensitive elements that communicate through a serial bus. Another e-skin solution was presented in [82]. It consists of a stretchable active matrix sheet of sensors that can be stretched by 70% without mechanical or electrical damage. The elastic conductor allows for the construction of electronic integrated circuits, which can be mounted anywhere, including arbitrary curved surfaces and movable parts, such as the joints of a robot's arm.

The most common strategy for device readout is direct addressing, in which each device is contacted by a separate connection. Good temporal resolution can be achieved with this method, but large arrays quickly lead to an unmanageable number of connections. X-Y scanners is one of the direct readout strategies. It consists of two sets of parallel electrodes with the sensing elements located at the crossing points of perpendicular electrodes. Appropriate readout mechanisms allow reading as large matrix, with the disadvantage of slow readout speeds.

It is important to mention that, despite the vast number of innovative designs, most authors emphasis has been on the sensors and on the mechanical challenges related, and the read-out system has largely being ignored. This is pointed out by [83], where authors shows that there are only a few tactile sensing arrays with any kind of mixed mode (analog and digital), or simply electronic circuitry on chip with sensors. Very few of the reviewed work even mentions any of the constraints and challenges posed by the system,

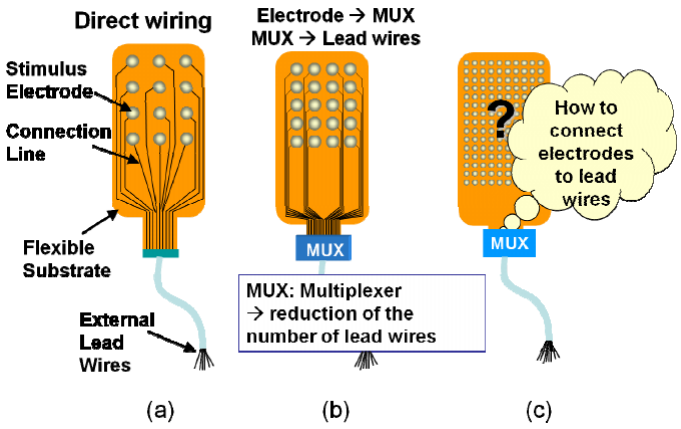


Figure 2.5: (a-b) Different interconnection schemes of electrodes of a brain implantable stimulator and (c) the interconnections limitations faced as the number of electrodes scale. (Figure taken from [4]).

like the embedded electronics, computing of the data, interconnection schemes, networking, wiring, power consumption, robustness, manufacturability, and maintainability.

Biomedical Devices

Electronic biomedical devices are broadly used in medical applications with different purposes. In many cases, arrays of MEMS devices are desirable for sensing and/or actuating withing the human body. For example, implantable arrays of electrodes are used to sense electrical signals from different parts of the brain, or to stimulate muscles artificially.

For example, in [84] the authors present two implantable electronic devices meant to provide artificial vision in blind patients. It is composed of a retinal prosthesis, and a brain-implantable stimulator. In simple terms, the retinal prosthesis works like a low resolution camera, that sends the sensed signals to the brain stimulator, that in turn provides the visual information to the brain using electrical stimuli.

The stimulator is a MEMS device with an array of electronically controlled actuating electrodes, in order to build a high resolution stimulator, with numerous stimulus electrodes. According to the author, the stimulators should ideally count with more than 1000 electrodes to enable good quality artificial vision. In [4], in attempting to fulfill these requirements, the authors use multiple shared buses in a tree scheme in order to interconnect many electrodes to a main controller. Figure 2.5 shows the different schemes utilized.

More recently, a fully intraocular CMOS prosthesis with 512 channels was presented in [85]. It is a stimulator with 16 shared buses, with 32 stimulators individually addressable per bus. One central logical controller, which is connected to outside the network, controls each stimulator by writing to its communication interface.

In a slightly different context, electroencephalogram (ECG) is the recording of the electrical activity along the scalp, which can be seen as a signature of the brain activity, and can be used to quantify and diagnose humans health among others. This is another application example of the biomedical field that might require hundreds of sensing elements. In [86] the authors comment on different ECG techniques, and on the need for complex signal processing techniques. In [11], the authors compare works with different number of electrodes, and recommends at least 125 channels for reliable results. Such density of electrodes leads to challenging setups due to wiring (small) sensors individually to each channel, which can be challenging. It is prone to installation issues, and does not provide opportunities on real-time distributed processing, but only off-line analysis.

These examples of biomedical devices turn out to have the same scalability issues as seen in various use cases surveyed.

2.4 Summary

In this chapter we reviewed networking and sensing technologies which inspired and motivated XDense. We also review the different typologies, routing and communication protocols used by common NoCs, which is the network solution that most resembles XDense. We analysed in more details the Epiphnay processor due to its similarities with XDense. We also identified the differences and the limitations.

We also reviewed some available air-flow sensing technology, and their current limitations. We review general propose arrays of sensors and actuator that try to address the challenges of dense sensing within different application contexts. From the analyzed solutions, its clear that the interconnections issues have been largely ignored, since most of the research focuses only on the sensing elements and the mechanical aspects.

Chapter 3

Survey of Distributed Data Processing Techniques for Dense Sensing

3.1 Introduction

Having reviewed network architectures and a broad range of sensor and sensor networks, in this chapter we survey the techniques and methodologies utilized to endow XDense with distributed feature detection capabilities, and real-time communication capabilities.

We start by reviewing distributed data processing algorithms that allow detecting and extracting features of interest from raw data, both in the field of sensor networks as well as in the context of many-core processors with NoCs. We want to understand how distributed algorithms from both areas intersect, as we believe the most suitable strategies for XDense to reside exactly in this intersection.

Enabling feature detection and extraction by distributed processing is not sufficient. We have to provide time guarantees, as we target closed-loop applications like AFC, with real-time requirements, where timeliness is fundamental. We review different approaches that provide real-time guarantees for NoCs with similar network topologies to XDense, aiming at finding the approach that best meets the requirements of XDense.

3.2 Feature Detection and Extraction

3.2.1 Using Many-core Processors

Due to the similarity between the topology of XDense and some many-core architectures (as presented in Chapter 2), it is important for us to understand how some distributed processing algorithms perform on many-core processors, and how they benefit from each

architecture specifically. Our interests are specially on image processing algorithms, that have a lot in common with the algorithms we want to use with XDense. These similarities regard mainly the underlying objective of feature detection, that is a common goal.

Extraction and matching of 2-D features in image and video is important in many computer vision tasks like object detection, recognition, structure from motion and augmented reality. Many computer vision and image processing algorithms map well into distributed processing models and have high inherent parallelism. Image processing tasks, which can process multiple pixels independently (for example convolution) can be performed very fast by fragment programs (computation kernels) by exploiting parallelism [87].

For example, in [88] the authors benchmark some feature tracking and feature extraction algorithms that run on graphic processing units (GPUs), suitable for video analysis in real-time vision systems. A GPU-based feature tracking algorithm called KLT [89], tracks about a thousand features in real-time at 30 Hz on 1,024 $\times$ 768 resolution video, which represents a 20 fold improvement over a quad core central processing unit (CPU). Another GPU-based feature extraction algorithms called SIFT [90], extracts about 800 features from 640 $\times$ 480 video at 10 Hz which is approximately 10 times faster than an optimized CPU implementation. Both KLT and SIFT have been used for a wide range of computer vision tasks ranging from structure from motion, robot navigation, augmented reality to face recognition, object detection and video data-mining with quite promising results.

In addition to GPUs, other many-core architectures can also provide great levels of parallelization. For example, the 2-D network topology of the Epiphany processor is appropriate for tasks in which the data may be decomposed into two dimensions. This property favors many image processing kernels, that have a natural 2-D domain decomposition whereas inter-process communication occurs among neighboring processes at neighboring cores within the 2-D computational domain [41].

In [91] the authors benchmark the performance of the Epiphany chip running different image processing kernels. More specifically, they show that image processing algorithms can benefit of this property, such as the 2-D Fast Fourier Transform (FFT) with high-pass filter for edge detection; Gaussian blur, used for image noise reduction, and a Sobel filter, used for edge detection. Gaussian filters are often used as part of larger image processing tasks that rely on inter-core communication for sharing image edges between neighboring cores. The Epiphany architecture can perform basic 2-D arranged operations, like a Gaussian filter or the Sobel operator with remarkable efficiency.

To parallelize tasks using the Epiphany processor, the authors use a lightweight implementation of the message passing interface (MPI) programming model to perform 2-D

arranged operations efficiently. However, due to the limited memory resources at each core, algorithms must be re-designed for lower and more efficient memory utilization.

Despite the similarities between the Epiphany and the XDense architecture, the memory and computing power available in a XDense node is much higher since each node is a complete system on chip with much larger memory and CPU capacity. In the other hand, the parallel links that interconnect the cores of the epiphany chip have much higher throughput when compared to serial links. This settles different challenges and opportunities.

The Tile64 many-core processor also shares similarities with the XDense regarding its network architecture. In [92], the authors evaluate its capacity to perform real-time feature detection using its distributed processing power. Specifically, they provide strategies for parallelizing algorithms for hazard detection and avoidance for an autonomous vehicle. The hazard detection algorithm constructs a topographic map of the area using light detection and ranging (LIDAR) sensing data, to then generate estimates of surface slope, roughness and potential hazards. Given the detected hazards, the hazard avoidance software generates a map that encodes the distance to the nearest hazard at each pixel in the map.

Because of the in-place nature of its memory architecture, some algorithms have to be rewritten in order to cache data locally while the other processors finish their work, to avoid overwriting of data yet to be processed. If one core needs to access data from another core, it can also access the other core's cache memory, without being delayed by accessing the shared memory. Taking these optimizations in mind, for this type of application, the Tile64 architecture provides excellent performance.

After reviewing applications such as the above, the benefits of parallelism of algorithms for image processing become notorious. It is also important to notice that many-core processors with architectures similar to the one from XDense (namely the Epiphnay and Tile64 processor) can perform very well on running distribute image processing algorithms; specially the algorithms we intend to use with XDense for airflow feature detection.

3.2.2 Feature Extraction in Sensor Networks

Even though XDense utilizes NoC-like topology and protocols, it is ultimately a sensor network, which makes it important to understand how distributed processing is performed in the context of sensor networks in general. We review wireless sensor networks (WSN), since a lots of effort have been put towards efficiently extracting data from large deployments through distributed data processing.

There are various techniques found in the literature about feature detection and extraction on sensor networks. More specifically, boundary and edge detection and tracking is a common functionality of interest on sensor networks. A good survey of such techniques can be found in [93].

Some work in the area is worth mentioning. In [94], sensors acquire measurements of neighboring nodes to determine whether they are near the edge of two inhomogeneous (two or more smoothly varying regions separated by boundaries) fields. The work proposes three approaches for edge detection using a WSN (one based on a statistical approach, one based on a high-pass filter and another based on a classifier), where each sensor gathers information from its neighbors, and independently determines whether it is on the edge of an event.

The approach in [95] is to disseminate a query such that it propagates by following the contour line (isoline). The work assumes a tree-based routing scheme, which is used to disseminate this query. Upon receiving the query, each node decides if it is part of the contour line (isoline) or not according to its sensor value, the sensor values of neighboring nodes and the contour line tolerance band. Nodes also perform local modeling on sensing values within its neighborhood to obtain an estimation of the gradient direction, needed to construct the contour map.

Some other works employ in-network data processing techniques, aiming at reducing the total number of transmissions. For example in [96], the observations from sensors are aggregated and confidence intervals around the true boundary are obtained for a set of points. The procedure requires a hierarchical structure of communication. The sensor nodes must be able to detect not only local physical properties such as local temperature, but also measure the properties within a certain distance. Many other examples of aggregation techniques can be found (e.g. [97, 98, 99]).

Another interesting approach is to define a model of the physical phenomenon being monitored [100]. Using this model, nodes can then individually decide if the phenomenon is behaving as expected, and significantly reduce communication by only reporting deviations.

A cross-layer approach to contour nodes interference of monitored physical phenomena with data fusion in WSN is presented in [101]. Sensor nodes that are close to the desired contour line within some distance are defined as contour nodes. To determine the contour line location, the probability for a sensor node to be a contour node is calculated at a central node called fusion center. Additionally, the authors design an adaptive data fusion scheme to avoid excessive packet retransmissions.

It is important to notice that the work reviewed was developed to track physical phenomena such as temperature changes over a large area. These techniques mentioned were

not designed to cope with the requirements of very dense sensing, and highly dynamic phenomena. While they provide good insights into techniques to achieve the same overall goals, the requirements of very dense sensing, and highly dynamic phenomena call for a more integrated approach. Other works such as [102, 103, 104], although addressing similar dense deployments, they built on top of dominance based network media access control (MAC) protocols, which have important practical limitations. Thus, in our work, we propose a design that integrates the sensing architecture (including the interconnect) together with the algorithms for data extraction.

3.3 Real-time Communication

As stated in the Introduction, providing real-time communication guarantees for XDense is of crucial importance to enable AFC; a great source of temporal non determinism comes from communication delays (as we discuss further in Chapter 6). We review the different approaches given to the problem of providing real-time guarantees for communication networks that resemble the one we use in XDense.

3.3.1 Real-time Guarantees for NoCs

We start by reviewing some relevant work that aim at providing real-time guarantees for NoCs.

As pointed out in [60], often real-time guarantees are provided for NoCs as different quality of service (QoS). Generally, NoC systems provide two QoS: best-effort (BE) and guaranteed service (GS). These are two different levels of commitment that differ on the predictability of the communication behavior in terms of: (i) data delivery; (ii) data integrity; and (iii) performance guarantees.

In most NoCs, BE communication provides guarantees on data delivery and data integrity, whereas for GS communication performance guarantees are provided in addition. GS communication guarantees are usually correlated with an analytical verification framework.

For instance, the authors of [105] provide GS by proposing a worst-case analysis for priority-preemptive, wormhole switched NoCs. This approach was later extended in [106], in which an end-to-end schedulability analysis was proposed for many-core systems.

Additionally, in [107] authors provide methods to efficiently calculate the worst-case bandwidth and latency bounds for real-time traffic streams on wormhole-switched NoCs with arbitrary topology.

ÆTHEREAL [59], NOSTRUM [108], MANGO [109], SONICS [110], aSOC [111], SoCBUS [57] and also the NoCs presented in [112], and the static NoC used in the RAW multiprocessor architecture [113], are some other examples of NoCs that implement GS.

Most of the works referred above implement GS by using variants of time division multiplexing (TDM). TDM is used to implement connection-oriented packet routing, thus guaranteeing bandwidth on connections.

The clockless NoC MANGO uses virtual channels to establish virtual end-to-end connections. Hence, limitations of TDM, such as bandwidth and latency guarantees which are inversely proportional, can be overcome by appropriate scheduling. In [109], a scheme for guaranteeing latency, independently of bandwidth, is presented. Differently, in [114], an approach for allocating individual wires on the link for different connections is proposed. The authors call this spatial division multiplexing as opposed to TDM.

However, all the above solutions are tailored to parallel links found in NoCs, which are designed for very high bandwidth, where large amounts of data transfer between cores are required. Because XDense relies on non-prioritized packet switched serial communication, these approaches do not fit the XDense network architecture.

3.3.2 Real-time Guarantees for General Purpose Networks Using Traffic Shaping

More in line with our approach, initially proposed in [115], network calculus enables real-time communication for packet switched multi-hop point-to-point networks, addressing the issue of guaranteeing the delivery of messages with time constraints.

Network calculus has been extensively researched since then; in [116], the authors survey the state-of-the-art of deterministic and probabilistic network calculus, by providing a review of service curve models of common schedulers along with different types of networks and methods for identification of a system's service curve representation. With a slightly different concept, the authors in [117] propose a feasibility analysis of periodic real-time traffic in packet-switched networks using first-in-first-out (FIFO) queuing. Their framework fits real-time analysis of switched Ethernet, and can provide better results as compared to network-calculus in some cases.

Traffic shaping has also been used along with network calculus in order to achieve tighter deterministic bounds. For example, in [118] the authors use rate controlled Earliest-Deadline-First scheduling in conjunction with per-hop traffic shaping to provide deterministic end-to-end delay guarantees. They identify the shaping parameters that result in maximal network utilization. This has also been studied for NoCs in [119] for worst-case response guarantees and buffer space optimization. Traffic shaping is also used on

Ethernet networks to provide real-time guarantees by shaping the packet sources [120]. The authors provide an asynchronous traffic scheduling algorithm, which gives low delay guarantees, while keeping low implementation complexity.

In [121] the authors investigate a scheduling scheme for packet injection in a NoC with a ring topology. A basic scheduling is used to favor traffic already in the ring while providing high network utilization. A weighted scheduling scheme is also used to prioritize and serve different cores in the system with the trade-off of network utilization. In [122], the authors also study the impact of varying the packet injection in the system, by providing an injection control mechanism in order to maintain a fixed buffer size.

3.4 Summary

In this chapter we reviewed two areas of interest: feature detection and extraction using many-core processors and sensor networks, and real-time communication.

By reviewing strategies for feature detection and extraction on many-core networks processors, we were able to understand why and how feature detection algorithms benefit from many-core processor architectures. Specially regarding the inter-core communication strategies, that allow local data sharing for decentralized processing of data, for increased CPU utilization with lower network congestion. These properties are specially interesting for the development of XDense, as we plan to apply the same strategies on our network.

Distributed processing strategies utilized in sensor networks also show that it is possible to benefit from distributed processing for feature detection, even with low capacity links with high congestion.

In the second half of this chapter we review different techniques utilized to provide real-time communication guarantees for NoCs and some other wired mesh networks. We focused mainly on networks that have some degree of similarity with the XDense design, as we believe that some of these techiniques can be adapted to XDense in order to provide it with real-time capabilities.

Part II

Proposed Novel Design: XDense

Chapter 4

Network Design and Principles of Operation

4.1 Introduction

Having discussed the application requirements and revised the available technology and related work in Chapters 1 and 2, we now design the architecture and define the principles of operation of XDense.

We start by the design of the internal architecture of the node, and each of its internal components, essential to its operation on the network. We also define all the protocols required for nodes to communicate on the network, and to allow XDense to accomplish its main goal of performing dense sensing in a timely fashion.

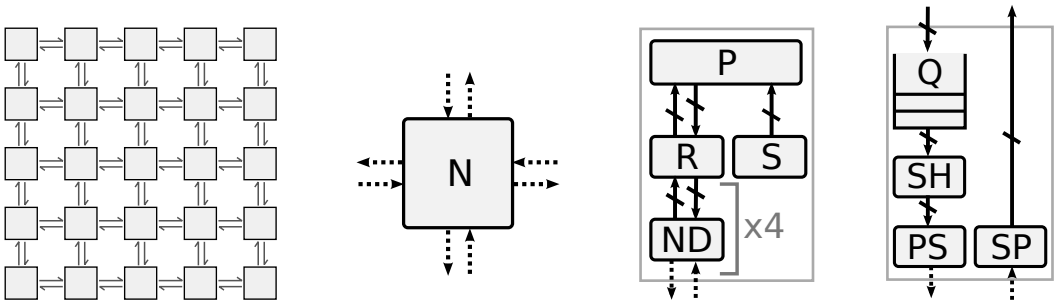
Considering that one of our main objectives is to have a feasible sensor network (based on low cost COTS components) for us it is essential to consider the hardware implementation during this phase. This is a co-design problem, in which every design decision, including the principles of operation, must be taken considering their impact on the hardware design. On the other hand, the hardware chosen imposes the resource availability, the processing power and, consequently, the maximum achievable performance.

While taking design decisions, it is always better to choose components from existing general purpose networks with proven performance, than to design a network from scratch with a topology matched to the problem.[1] Therefore, the selection of a practical network design and architecture is a job of fitting requirements to available technologies. The trade-off between functionality, performance and cost also needs to be considered.

[1] This is a common pitfall for network designers, that wastes valuable time on design and verification [123].

In this chapter we present the design specifications, which describes the network and node architecture and principles of operation. We also present the basic aspects of XDense's communication timeliness. Following that, we detail XDense operation by looking at the protocols and its implementation, at each of the abstraction layers individually.

4.2 Network Design



(a) Network (b) Node (c) Node internals (d) ND internals

Figure 4.1: Overview of XDense architecture. (a) It is a 2-D mesh network; (b) Node pinout: two channels per port for transmitting and receiving data; (c) Node internals: processor (P), router (R), net-device (ND) and the sensor (S); (d) net-device's internals: output queue (Q), traffic shaper (SH), and a parallel-to-serial/serial-to-parallel (PS/SP) converters.

XDense architecture is a 2-D mesh network. Nodes are connected in a grid, with up to four neighboring nodes, physically located in the four directions (north, south, east, west). This architecture shares some similarities with NoCs regarding network topology, routing schemes, timing properties and distributed computing capabilities [33]. The topology enables low-latency local communication between nodes, allowing the user to exploit its in-network data processing capabilities, and execute algorithms distributively.

Despite the similarities with NoC, it differs greatly in physical dimension and node count, since XDense is meant to be deployed on surfaces (like wings in the aeronautical usecase), and node count will tend to be much higher as compared to number of cores on NoCs. It also differs in terms of link bandwidth, since NoCs tend to use wide parallel links to interconnect nodes [41], while XDense relies on full-duplex serial links for lower integration cost (given the dimensions of the nodes and network). Also, data is generated locally at each sensor node, which imposes other restrictions and opportunities.

It also differs in some ways from traditional wireless sensor network approaches. We use short wired serial links, which are very little susceptible to concurrency or noise issues, compared to wired shared buses or wireless links, and also allows higher communication rates. We consider a far denser deployment scenario than traditional SNs (of the order of hundreds of nodes per square meter). Nodes may share power supply and the impact of communication on power is negligible as compared to battery powered radio links.

An example scenario of a 5×5 network is show in Figure 4.1(a). Each node is connected to its neighboring nodes using two directional links as seen Figure 4.1(b). Each node can be tough off as a self-contained system, with dedicated hardware peripherals and a CPU. Figure 4.1(c) illustrates the components of a node's internals at different levels of abstraction. A node is composed of a sensor (S), a processor (P) and a router (R) and is connected to its neighboring nodes located in the four cardinal directions using bidirectional communication ports (termed networking devices (ND)). Because they are bidirectional ports, we refer to their input and output independently as input and output ports.

Any set of nodes can be a potential multi-hop communication link from any node to the sink, enabling fault tolerant protocols. Multiple sinks are supported, and any node can be configured to be a sink.

In the rest of this section, we define each of the XDense components and their operation.

4.2.1 Networking Device

At the lowest abstraction layer in the node's architecture, networking devices are node's component responsible for connecting with other nodes. Each node contains four networking devices that connect them to their four immediate neighbors in the grid.

Networking devices internals are shown in Figure 4.1(d). Its design resembles that of a full-duplex asynchronous serial port, with some differences. At its output, there is a queue (Q), a traffic shaper (SH) and a serializer (PS). Outgoing packets delivered by the router (R) are first queued, dequeued by the traffic shaper, serialized and transmitted. At the input port, there is only a deserializer (SP). Input packets are deserialized and served by the router (R) immediately, and queued at the destination output port, and/or at the input of the processor (P).

The serializer is used by XDense nodes to convert packets from its in-memory form (inside the node), to its serial form so it can be transmitted serially (outside the node). The deserializer does the opposite. The purpose of the traffic shaper is to provide determinism

to the output traffic, and consequently make it amenable to real-time analysis. Briefly, its function is to implement a release offset on the output traffic to make the transmission periodic. This enables us to formulate the output traffic as a linear cumulative function of the input traffic. We will discuss our traffic shaping techniques in detail in Chapter 6. All network transfers are non-preemptive packet-switched, and all packets have a fixed and equal size.

4.2.2 Router

The router (R) is the interface between each networking device (ND) and the processor (P). The router is able to receive and transmit packets in parallel, from/to the processor and networking devices. Packets generated by the processor are transferred and queued at the router, in a dedicated queue, which is served in the same way as those from the networking devices (ND).

Packets are transmitted in the network in a multi-hop fashion from a source node to a unicast, multicast or broadcast destination. At each hop, the router analyses and updates the incoming packet's headers. The routing protocols determine where to forward each packet, through one or many NDs and/or to the processor (P); that is based on the routing protocol configured and on the origin and destination of the packet.

Packets may compete for a single output port, in which case they are served using a defined arbitration policy. In this work, we use FIFO arbitration. In FIFO, input packets are immediately processed by the router, and only get queued at the output port it is competing for. This was a decision made to favor analysis for applications with real-time requirements and is further discussed in Chapter 5.

4.2.3 Processor

The processor (P) is where the application runs. It is connected to the sensor through a dedicated link that poses no interference on any other communication link inside the node. On the other side, it communicates over the network through the router (as mentioned above). A queue at the input link between the processor and the router queue input packets to be server by the processor. The purpose of the processor is to provide high level functionalities to fulfill XDense goals like, enabling distributed data acquisition and processing. We do that by implementing different predefined primary operations to enable the exchange of sensed data between nodes, and the execution of a data processing algorithm on the data exchanged by any node.

According to the application goals, based on these primary operations, we define application protocols as a sequence of these operations executed by nodes. Different application protocols may enable more elaborate applications to be executed on the XDense network. Each application protocol will be discussed further on this chapter as we talk about XDense's application protocols and its principles of operation.

The processor should be able to interface with any kind of sensor, according to the application's monitoring goal. Also, it should be able to accommodate applications that might require measurements of quantities of various kinds of phenomena, from more than one sensor. For example, to enable high-precision AFC, pressure and temperature sensors are desirable, and can jointly provide better sensing of the airflow [1].

4.2.4 Sensor

The sensors interact with the physical world, and transduces the physical stimulus into a measurement quantity that is feed to the processor. The sensor is application specific, and depends on the nature of the phenomena to be monitored and its requirements.

The nature of the data may influence on node's behavior, and therefore on network load and performance. This is important in cases where the user is intended to investigate on distributed processing algorithms, when it is indispensable to have access on the expected input data (for AFC for example). We provide an implementation model that will be detailed in the next chapter, as we present the simulation model.

4.3 Assumptions and System Definitions

In this section we define the basic concepts of the temporal behavior of the network, the packet structure and the addressing system adopted.

4.3.1 Network Temporal Behavior

As we use serial links, it is important to remark that the delays imposed by communication is orders of magnitude higher than the delay due to routing a packet inside the node. In other words, moving packets inside nodes consists of copying data between different memory regions using dedicated parallel buses at the CPU clock rate. While, to communicate them serially using an universal asynchronous receiver transmitter (UART) port, the packet is serialized and asynchronously transmitted in a fraction of the clock of the CPU.

Because we co-design the network model and implementation model, we performed a statistical survey on the time required by different hardware platforms to route a packet. The results have shown that, depending on the implementation details and hardware capabilities, the internal delays can be as low as four clock cycles using an FPGA, to one hundredth of the time required to transmit a packet using a 100 MHz microcontroller. The complete results are presented later in Chapter 7.

Given the insignificance of the internal delays as compared to the communication delays, we neglect internal delays for the rest of this work and only account for delays imposed by communication. This simplifies the temporal analysis of the network. We define that the time taken for one node to fully transmit a packet to its neighbouring node as 1 transmission time slot, or 1 TTS. A transmission time slot is defined as the packet duration, calculated as:

$$1\ TTS = \frac{\text{PacketSize} \times (8+2)}{\text{baudrate}} \tag{4.1}$$

The packet size (given in bytes) is multiplied by eight bits per byte *plus* two bits added per byte by the UART physical layer overhead (start and stop bits). One TTS is then the total size of the packet in number of bits, divided by the communication baudrate given in bits per second (*bps*). As the baudrate is an implementation specific parameter, we normalize our temporal base according to TTS.

4.3.2 Packet Structure

The XDense packet structure was designed for low resource utilization and for ease of routing and processing by simple low cost COTS hardware. We choose a fixed size packet structure to allow fast handling of packets by the hardware; the size is dependent on the input and output buffers' size at each node.

We choose a packet size of 16 bytes, which is a typical buffer size of UART ports on COTS microcontrollers.[2] Out of these 16 bytes, 6 bytes are for protocol overhead and error checking and 10 bytes for the actual payload. It is important to notice that, because the UART ports are asynchronous, for each byte transmitted, two extra bits are added as start and stop bits, so nodes can synchronize their bit rate at each transmitted/received byte. This means that, although the packet can carry 128 bits (16 bytes) of useful data, its total size is 160 bits. The detailed packet structure, excluding start and stop bits, is shown in Table 4.1.

[2]For example, the Atmel SAM4N8A, a microcontroller with ARM Cortex M4, has five UART ports with 16 bytes buffer at both input and output. More details on the node hardware implementation are provided later in Chapter 7.

Table 4.1: Packet structure and size in bits, totaling 128 bits (16 bytes).

	Protocol			Addressing				Payload	Checksum
	RP	CP	AP	x_a	y_a	x_b	y_b		
Size (bits)	3	2	3	8	8	8	8	80	8

The packet header carries protocol and addressing information. The first byte, allocated to the protocol, is subdivided into 3 independent fields, dedicated to: Routing (RP), communication (CP) and application (AP) protocols. The three combined with the 2 coordinate pairs used for addressing will tell the router how to interpret and serve the packet. Packets are processed by node's router and are consumed (forwarded to the node processor) and/or forwarded to a single or multiple destinations, using the defined routing algorithm. CM, RP and AP protocols will be detailed in Section 4.3.3.

The coordinate pairs $[x_a, y_a]$ and $[x_b, y_b]$ are used with different purposes according to the communication protocol utilized. One functionality common to most protocols is to carry information about the origin and destination of the packet. Another functionality is to define multiple destinations, by designating a region of the network that should consume the packet specified by $[x_b, y_b]$. Again, the way the router interpret these coordinates depend on the communication and routing protocols specified on the header of the packet.

Since we use wired short range communication links, which are little prone to interference, we do not consider link errors. Hence, no error correction nor recovery algorithms are considered. If required, error recovery protocols can be implemented at the application layer.

4.3.3 Addressing

The coordinate pairs $[x_a, y_a]$ and $[x_b, y_b]$ at each packet, refer to a relative coordinate system whose origin location ($[0,0]$) is the node who originated the transmission.

Relative addressing adds scalability in many aspects, but mainly because nodes do not need to be uniquely addressed in a distinguished setup phase, and it allows having a network greater than the actual address space. The address space only limits nodes to a confined "horizon", which is how far each node can communicate to.

As from Table 4.1 each coordinate x_a, y_a, x_b and y_b is represented by an 8-bits signed integer. The coordinate size in bits is $c_s = 8$, which defines each node address space ($2^{c_s} = 256$). Meaning that each node is able to communicate with nodes in the range of -128 to 127 hops in the x and y directions.

Because we use relative addressing, the size of the network is not strictly limited to a square of $2^{c_s} \times 2^{c_s}$ nodes. On the contrary, the address space only confines each

individual node to this square communication range around itself, while the network can grow beyond these limits.

We chose $c_s = 8$ bits because a network with $2^{c_s} \times 2^{c_s}$ nodes gives us enough resolution for capturing phenomena in the usecases we are interested in. Otherwise, c_s could be increased, sacrificing the payload size for a greater address space. On the other hand, by increasing the network size without increasing the number of data sinks on the network may lead to undesired contention, increased delays and consequently slower sampling rates.

4.4 Networking Protocols

In this section, we outline the protocols that comprise the XDense communication stack. The layered protocols consist of the routing, communication and application protocols, as introduced earlier in this chapter. The routing protocols allows delivery of a packet from its origin to its destination on a deterministic route. The communication protocols allow nodes to perform one-to-one, one-to-many or one-to-all transmissions. Finally, the application protocols enable the nodes to establish complex transactions, such as request/response communication, handshaking, configuring, among other possibilities.

4.4.1 Routing Protocols

We implement multiple routing algorithms to enable applications to exploit a diversity of routes in the network. This allows the system to reduce contentions, reduce bottlenecks, deliver better load balance and provide better means to benefit from the network capacity overall.

Moreover, given that communication delays are the predominant source of delays, exployting routes diversity becomes even more important for real-time applications, in which routes without interfering traffic are essential to establish temporal predictability.

The basic idea is to provide multiple possible orthogonal routes, allowing transmissions occurring in one direction to not affect others occurring in an orthogonal or opposite direction. This is possible because we assume that routers can route packets in parallel, simultaneously, as long as the incoming packets do not compete for the same output port.

We use multiple variants of the common dimensional ordered routing (DOR) protocols [124]. In DOR protocols, all packets follow the same order when traversing. First, the progress occurs on only one of the axis, and upon reaching the desired coordinate of the destination, (if necessary) the transfer is continued along the other axis, until reaching the destination. The advantage of DOR protocols is that they always use the shortest

path between the source and destination nodes (considering rectilinear distance, a.k.a. Manhattan distance) and are proven to be deadlock and livelock free [125].

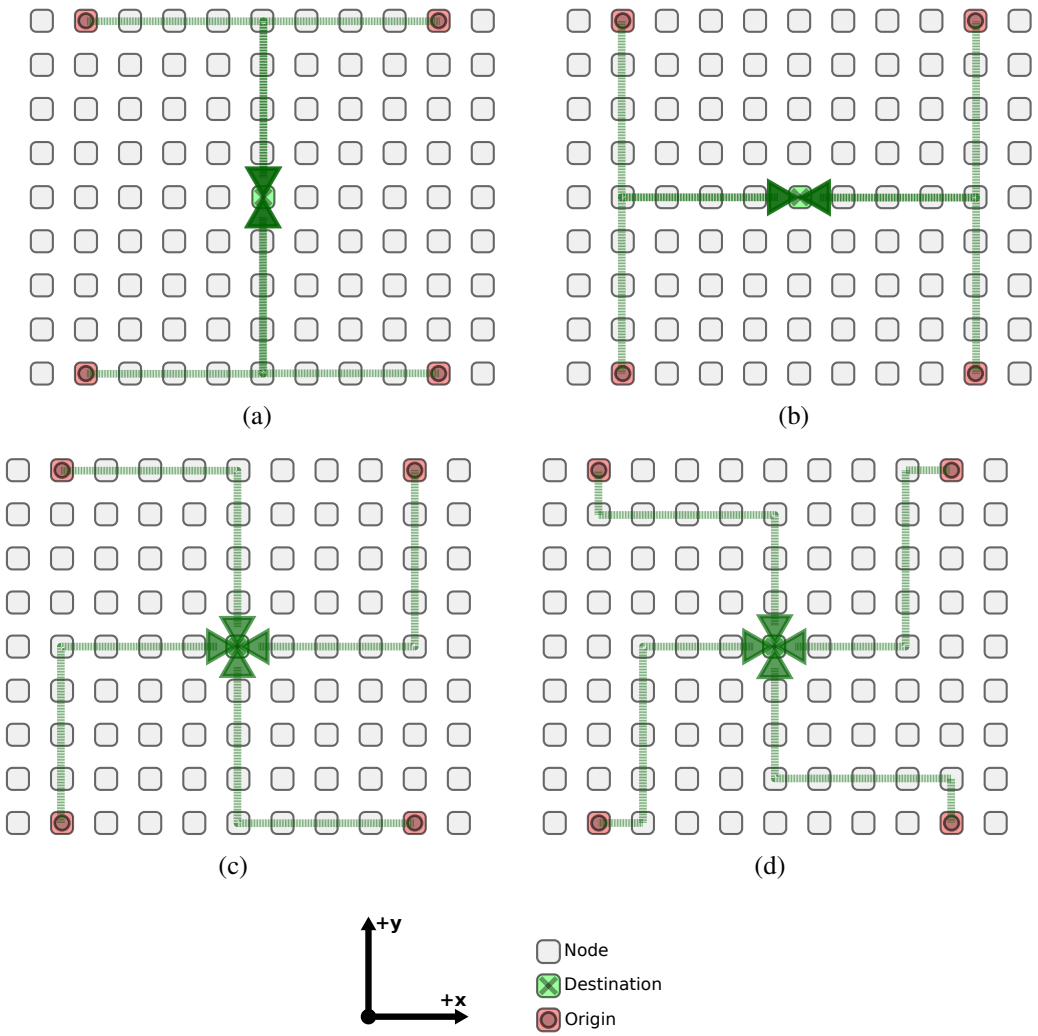


Figure 4.2: Routing protocols - Nodes unicast to the node in the center using different routing algorithms: (a) XY; (b) YX; (c) Clockwise; (d) Shifted-clockwise.

XDense implements six DOR protocol variations. This allows enough diversity of routes, yet with low implementation complexity (due to their commonality), and low communication overhead. More specifically, we provide *X-Y*, *Y-X*, clockwise, counter-clockwise, shifted-clockwise and shifted-counterclockwise routing protocol variations.

Figure 4.2 shows different many-to-one scenarios, in which nodes transmit to the node in the center. Each node transmitting is highlighted, as the packets being transmitted, which are highlighted at the respective networking device it is traversing. Each scenario shows a different routing protocol, which we define as follows.

XY Routing In X-Y routing (resp. Y-X), packets are first routed along the X (resp. Y) dimension and then along the Y (resp. X) dimension (see Figures 4.2(a) and 4.2(b) respectively).

Clockwise Routing In the clockwise routing (resp. counterclockwise) the starting dimension (X or Y) depends on the quadrant in which the destination node is, relatively to the origin of the packet (see Figure 4.2(c)). The rest of the route is calculated as for the other DOR protocols.

Shifted Clockwise Routing Another routing protocol, hereafter referred to as Shifted Clockwise (resp. counterclockwise) routing, adds an initial change in dimension on the first hop and then uses a regular clockwise (resp. counterclockwise) routing. For example, in Figure 4.2(d), the packet sent from the upper-left node towards the central node, first travels in Y, then in X and then in Y again. A change in direction is add for route diversity.

This or any of the presented routing algorithms do not present any particular advantage over the other when considered separately. The actual benefits of having multiple routing algorithms is actually related to, as mentioned earlier, route diversity and concurrence avoidance.

4.4.1.1 Protocols List and Packet Content

Table 4.2: List of routing protocols and content of the RP packet field.

Routing Protocol	Packet Content
XY	0b000
YX	0b001
Clockwise	0b010
Counterclockwise	0b011
Shifted Clockwise	0b100
Shifted Counterclockwise	0b101

The routing protocol is the first field to be analyzed by the router. We reserved 3 bits for it, allowing us to specify on the header of the packet up to 8 different routing protocols. In our use-cases, having only six routing protocols have shown to be enough, and therefore we only provide these variations, as from Table 4.2.

4.4.2 Communication Protocols

Next, we explain the communication protocols that compose our communication stack.

4.4.2.1 Unicast

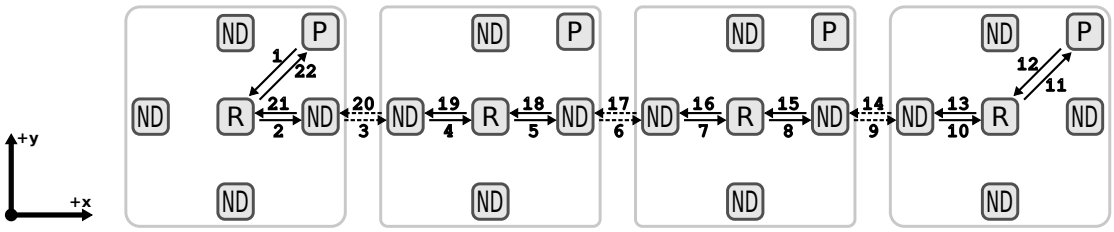


Figure 4.3: Unicast example: Node on left requests data from node on right with a unicast request. The figure shows all internal logical steps taken in the process of exchanging data between nodes.

Table 4.3: Unicast example - Packet content at each numerated instant of Figure 4.3.

Steps	1-4	5-7	8-10	11	12-15	16-18	19-21	22
x_a, y_a	0,0	-1,0	-2,0	-3,0	0,0	1,0	2,0	3,0
x_b, y_b	3,0	2,0	1,0	0,0	-3,0	-2,0	-1,0	0,0
Payload	-	-	-	-	DATA	DATA	DATA	DATA
Checksum	CS_1	CS_2	CS_3	-	CS_4	CS_5	CS_6	-

Unicast refers to a one-to-one transmission from one node in the network to any other node inside its address space, located at $[\Delta x, \Delta y]$ from it, transmitted in a multi-hop path using any of the routing protocol presented in Section 4.4.1.

Figure 4.3 shows a unicast communication. The start of communication is triggered by the processor (application layer) on the first node, which sends the request, through its router, to the rightmost node. At the intermediate nodes, the packet is received, processed and forwarded by the router, without the interference of the processor. At its destination, the router forwards the packet to the processor. The node may then perform a given action depending on the content of the protocol. In the request for data example in Figure 4.3, the receiver is expected to reply by transmitting a packet with its sensed value.

At each hop, as the packet traverses a node, its router updates the value of origin and destination coordinate pairs $[x_a, y_a]$ and $[x_b, y_b]$. This is done so that it reflects the current distance of the packet to its origin and destination respectively (in terms of the Manhattan distance). The packet reaches its destination the moment the coordinate pair $[x_b, y_b]$ becomes zero. At the same time, the coordinate pair $[x_a, y_a]$ will reflect the coordinates of the origin of the packet. This information is required by the application layer to reply to any kind of request.

The above sequence is explained with help of Table 4.3. From steps 1 to 11, as the packet traverses each hop in the positive direction of the x-axis, the router subtracts a unit from the coordinates x_a and x_b. The inverse occurs when it travels in the opposite direction of the x-axis, during steps 12 to 22. At each hop, before the packet leaves the router, the checksum is updated by the router according to the new content of the packet. On its way back, the packet payload contains the data requested. The checksum is meant to allow error verification on the data exchanged between each node. Currently, it is not utilized, as we assume error-free communication; even though we found important to consider it in case we see the need of it while experimenting with the hardware prototype.

Table 4.4: Unicast example - TTS: Transmission Time Slot (TTS) in which each logical step from the example of Figure 4.3 occur.

Steps	1-3	4-6	7-9	10-14	15-17	18-20	21-22
Transmission time slot (TTS)	0	1	2	3	4	5	6

In addition, Table 4.4 shows the TTSs of the occurrence of each step. As stated before, we only account for the delays imposed by communication, meaning that 1 TTS elapses while the packet leaves one node and arrives at its neighbor. Because we neglect internal delays, all steps that occur inside the same node happen in the same TTS.

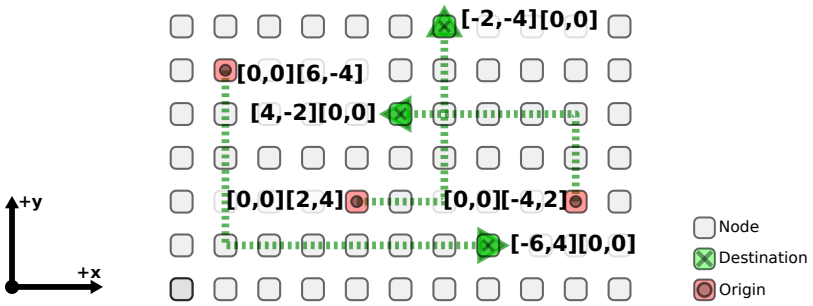


Figure 4.4: Example of three concurrent and non-interfering unicast transmissions. The coordinate pairs inside the square brackets show the content of $[x_a, y_a]$ and $[x_b, y_b]$ respectively, at the origin and destination of the packet.

Figure 4.4 gives a larger picture of the unicast communication. In this setup, three non-interfering unicasts happen simultaneously. The square brackets show the content of packet's origin $[x_a = 0, y_a = 0]$ and destination $[x_b = \Delta x, y_b = \Delta y]$ respectively. It is important to notice that, in this example, two flows intercept at a single node locate at $[6,4]$ (absolute coordinates). In this case, there is no interference, as there is no competition for resources (output ports) for two reasons: because the flows arrive at node $[6,4]$ at different time instants, since their origins are at different distances; and because the flows output ports are distinct, and therefore can be handled in parallel by the router.

Finally, we define the deterministic property of unicast transmissions. We state that, in a contention-free scenario (as the one in Figure 4.4), the end-to-end delay of a packet unicast is deterministic and defined by:

$$\text{DelayU} = abs(\Delta x) + abs(\Delta y)\ TTS \tag{4.2}$$

4.4.2.2 Multicast

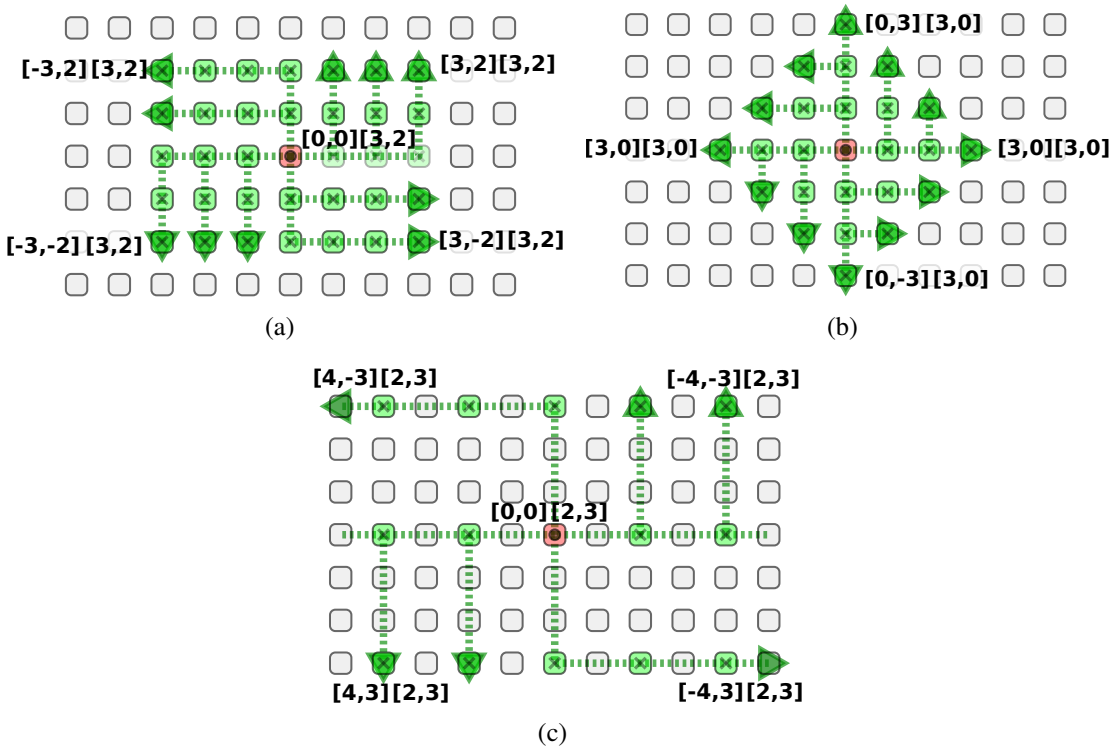


Figure 4.5: Multicast example - with relative addressing. The two coordinate pairs show the packet content of x_a, y_a and x_b, y_b at origin and destination.

Multicast refers to a one-to-many packet transmission, from one node to a group of nodes. This functionality is required, for example, in scenarios where a node wants to request data from specific groups of nodes. It also allows a node to rule clusters of nodes, in order to perform some coordinated operation. With multicast protocols we intend to enable some sort of selective communication, constraining the network activity to a limited number of nodes contained in a group or cluster. The node initiating the transmission is able to transmit the same packet in the four directions in simultaneous.

In that sense, in order to address the challenges related to our AFC use-case, we define three different types of multicast, each one meant to address application requirements differently. In the next paragraphs we present each variant.

Multicast Area: We call the first variant, Multicast Area (MA). Nodes work on the basis of forwarding the packet received only inside a defined rectangular area around the sender (and not outside it).

In this case, the two coordinate pairs in the packet header are used differently. The first coordinate pair (S_x, S_y) is constant, and defines a rectangular area around the sender. The second coordinate pair (O_x, O_y), contains the address of the origin of the packet, to allow the recipients to reply to any kind of request. The total number of nodes reached by this kind of multicast is given by $(2S_x + 1) \times (2S_y + 1)$ (including the sender).

At each node the router looks at the (O_x, O_y) coordinates to calculate their distance to the sender, in order to decide to forwarding the packet to its neighbors or not. If its distance in x or y matches that from the maximum area size S_x or S_y, it is no longer forwarded on that axis, and the multicast finishes. Every receiver of the packet consumes it; that is, internally forwards it from the router to the processor, to be processed by the application layer.

A multicast area example with $(S_x, S_y) = (2, 3)$ is shown in Figure 4.5(a). It shows that, as the packet is forwarded, only the (O_x, O_y) coordinates are updated by the router, while the cluster area (S_x, S_y) remains constant.

In a contention-free scenario, the time taken to perform a MA is given by:

$$\text{DelayMA} = (S_x + S_y)\ TTS \tag{4.3}$$

This is the time required for the packet to travel from the sender, located in the center of the rectangle, to the farthest nodes.

Multicast Radius: Another variation we use is Multicast Radius (MR). It is conceived with the same objective of allowing nodes to communicate within a cluster of nodes around the sender, but in this case, this cluster is no longer defined by an area, but by a radius.

The packet carries the coordinate pair (O_x, O_y) with the same objective of keeping track of the origin of the packet. In this case, the second coordinate pair carries a single integer $(S_r, -)$, which specifies the maximum distance from the sender to which the packet should be forwarded, given in number of hops.

An example with $S_r = 3$ is shown in Figure 4.5(b). Like MA, only the (O_x, O_y) coordinates are updated by the router, while the cluster radius S_r remain constant.

In a contention-free scenario, the time taken to perform a complete MR transmission is given in terms of TTS by:

$$\text{DelayMR} = S_r\ TTS \tag{4.4}$$

which is the time required for the packet to travel from the sender, located in the center to the farthest nodes.

Multicast Alternative: The last multicast protocol we propose is Multicast Alternative (MT). The coordinate pair (x_b, y_b) no longer defines a cluster around the node. Instead, it defines which nodes within the sender address space are supposed to consume the packet. This arbitration is with respect to their position relative to the sender. The arbitration is made at each node's router, as they receive the packet, by testing if its relative position O_x and O_y coordinates are both multiple of x_b and y_b, respectively. If true, the packet is consumed and forwarded to the next hops. If false, the packet is forwarded without being consumed. In short, the coordinate pair $(x_b, y_b) = (M_x, M_y)$ defines which nodes are meant to consume the packet in both axis, in an alternated pattern.

Note that in this case, not all nodes who receive the packet will necessarily consume it, but may serve only as a relay to forward the packet to the next hop. Also, the packet will travel until the end of the network or until the end of its address space, since the packet does not carry information that limits the transmission range.

As in the other multicast protocols, as the packet is forwarded, only the (O_x, O_y) coordinates are updated by the router. An example is illustrated in Figure 4.5(c), for $(M_x, M_y) = (2, 3)$.

The time taken to execute a MT transmission will also be limited by either the network size, or by the address space limitation and is given by:

$$\text{DelayMT} = min\Big(\big(max(De_{+x}, De_{-x}) + max(De_{+y}, De_{-y})\big), \lceil 2^{c_s}/2 \rceil\Big) TTS \tag{4.5}$$

where De refers to the distance between the sender and the edge of the network, in the direction of the axis that represents the farthest distance to the edge of the network on that direction. c_s is the coordinates size in bits, which defines the address space (as introduced earlier in Section 4.3.3).

This multicast protocol is specially important in scenarios in which establishing clusters of nodes is desired. For example, the sender can ask nodes at specific positions to aggregate the data of their surrounding nodes and send it back. It may also allow sampling data from the network in a more sparse way and dynamic way, for scenarios in which smaller granularity is required, or when the nature of the data is homogeneous and does not require sampling every node on the network.

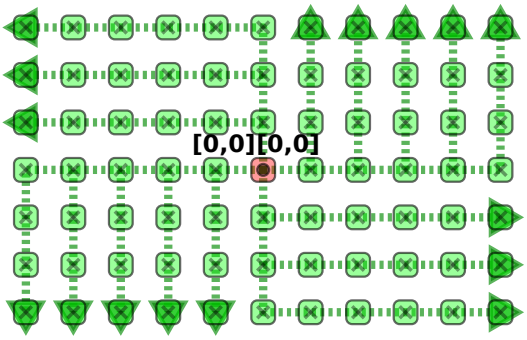


Figure 4.6: Broadcast example: packet flow from origin to destination, using relative addressing scheme.

4.4.2.3 Broadcast

The Broadcast (BC) protocol can be seen as a particular case of the MT protocol. The difference is that the packet will be consumed and forwarded by every node inside the sender's address space. That is, we set $(M_x, M_y) = (1,1)$, which results in all nodes that receive the packet will consume the packet (since in MT protocol, its coordinates will always be multiple of $(1,1)$.)

Just like MT, only the (O_x, O_y) coordinates are updated by the router as the packet travels. An example is illustrated in Figure 4.6. The time taken to execute a BC transmission will be limited in the same way, either by the network size, or by the address space limitation, given by:

$$\text{DelayBC} = min\Big(\big(max(De_{+x}, De_{-x}) + max(De_{+y}, De_{-y})\big), \lceil 2^{c_s}/2 \rceil\Big) TTS \qquad (4.6)$$

4.4.2.4 Protocols List and Packet Content

Table 4.5: List of communication protocols and content of the CP packet field.

Communication Protocol	Addressing				Packet Content
	x_a	y_a	x_b	y_b	
Unicast (UC)	O_x	O_y	D_x	D_y	0b00
Multicast Area (MA)	O_x	O_y	S_x	S_y	0b01
Multicast Radius (MR)	O_x	O_y	S_r	–	0b10
Musticast Alternative (MT)	O_x	O_y	M_x	M_y	0b11
Broadcast (BC)	O_x	O_y	1	1	0b11

The communication protocol occupies only two bits of the header, giving four different protocols possible. For each protocol, the meaning of the coordinate pairs $[x_a, y_a]$ remains the same, as corresponding to the origin of the packet, while the coordinates

$[x_b, y_b]$ is specific to each protocol variation. The meaning of each of these fields was presented earlier in this chapter.

4.4.3 Application Protocols

The application layer sits on top of the routing and communication layers and aims at enabling high-level functionalities. These include functions to perform data exchange between nodes, or groups of nodes, with the ultimate goal of fulfilling the sampling objectives of the targeted application scenario. However, as application objectives may vary, there are different approaches to fulfill these goals. So we address the potential requirements, while keeping design flexibility in mind.

We propose a set of high level application protocols with which we aim at giving the user the ability to program XDense, built using a set of application protocols, to customize for the application target.

Each of the proposed application protocol may be built with the utilization of any combination of communication and routing protocols. To decide on which application protocol to use, the designer should take into consideration the network load distribution, as well as the delays associated with each strategy. The application protocol is transmitted on the packet within the packet payload, making it completely independent of the routing and communication layers. The same payload may be used to carry sensor data.

We do not propose any specific strategy in this chapter, but only the building blocks to allow us to build and present our use cases later in this Thesis.

4.4.3.1 Data Request

We start by providing three different application protocols to allow nodes to directly request data from any other node, or groups of nodes. They may be located anywhere on the network, but inside the sender address space. After receiving the request, receiver node(s) reply with the requested data. This request can be a unicast, multicast or broadcast, leading to one, multiple or all nodes replying to the sender with the desired data. This is an essential functionality of our sensor network, but its implications have to be considered as this may easily overload the network.

Furthermore, when a data request is done, nodes can specify whether they want a node or a clusters' data. These possibilities may differ on the nature of the data, as explained below.

Node Data Requesting for nodes' data means that the receiver node(s) will reply to the request with their individual data (fetched by the receiver on its own sensor or internal

memory). Another possibility is to request nodes to send their data on an event driven basis. This allows a node to configure other node(s) to send back the requested data whenever an event occurs, such as a threshold reached. The node will then maintain this behavior until it is reconfigured again to perform another protocol.

The requested data can be from different sensors. The data may also vary. They might be raw sensed data or preprocessed data. For example, temporal derivatives and frequency of variation. The sender has to explicitly specify the desired data with its request.

Cluster Data Another possibility is for a node to ask any other node or groups of nodes to transmit back the data sensed by its surrounding nodes, or cluster. In turn, the node which gets the request, called cluster head, asks the nodes around it for their data. On reception of all the data, it processes/aggregates the data, and replies to the first node with the requested data. This application protocol will make use of the nested request function, in which the cluster heads will perform a *multicast area* or *multicast radius* (depending on request) to ask its cluster data.

Another possibility is to request cluster heads to send the aggregated data on an event driven basis. Each cluster head will continuously receive data from its cluster and analyze it. If an event of interest is detected, it transmits the processed data to the first node. The events of interest may be extracted from spatial and temporal data. This requires the user to program cluster heads to detect more complex features on the analyzed spatial and temporal data. Doing so will result in increased local network communication load, and reduced long range communication, leading to better network load distribution and reduced congestion.

With respect to our main use case (AFC), enabling local in-cluster processing allows extracting high level aerodynamic information of the airflow. This can then be transmitted back in a smaller number of packets when compared to transmitting raw data. The pre-processing and compression algorithms to be used are application-specific. We discuss application specific issues in Chapter 5.

Data Announcement Data Announcements are used to send data and can be done from any node to any other node or groups of nodes.

Data announcements can be set as periodic or event driven, depending on the application goals. In most cases, Data Announcements are triggered by Data Requests.

Table 4.6: List of application protocols and content of the AP packet field.

Packet content	Application Protocol		Associated payload
0b000	Request	A	N. of samples, period, offset,
0b001		B	
0b010		C	
0b011		D	
0b100	Announce	A	Data
0b101		B	
0b110		C	
0b111		D	

4.4.3.2 Protocols List and Packet Content

The application protocols are also summarized in Table 4.6. We allocate 3 bits on the packet header to specify the application protocol to be used. Out of these three bits, one is used to specify if the packet is either a data request or a data announcement. The other two bits are used to specify which data the packet is requesting or announcing, among four different data types.

4.5 Example Scenario

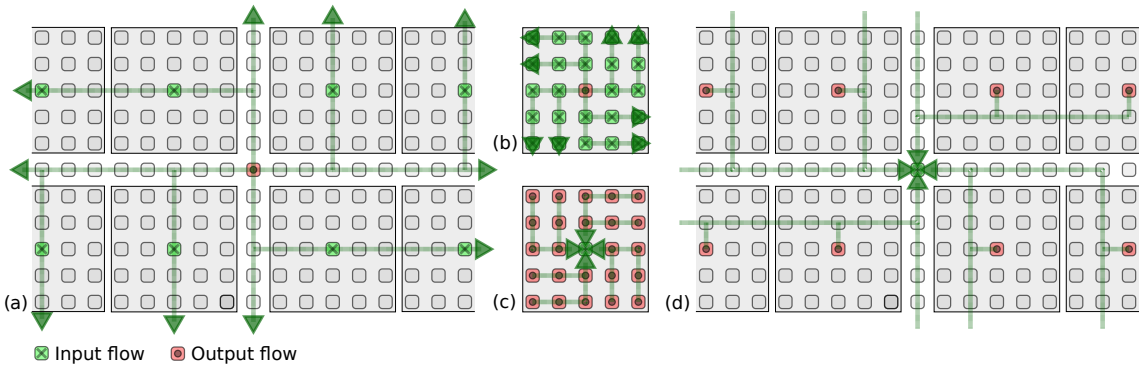


Figure 4.7: Example of networking protocols utilization. The application execution consists of: (a) Nodes requests data from cluster of nodes using multicast-alternative data request using counter-clockwise routing; (b) Cluster heads in turn request data from their cluster using multicast area with counter-clockwise routing; (c) Nodes unicast sensor data back to the requester using conter-clockwise routing; (d) Cluster heads process received data unicast it back to the requester using shifted couter-clockwise routing to avoid concurrency and contentions.

Figure 4.7 shows a complete application scenario that utilizes many of the protocols provided in order to extract pre-processed data from the network efficiently. The same application scenario is analyzed in detail later on in Chapter 6.

4.6 Concluding Remarks

In this chapter we have presented the details of the XDense design, including its architecture and principles of operation. The network uses custom architecture, designed to suite COTS hardware, with custom protocols conceived to have low overhead on the target hardware. The protocols provided are the building blocks of the sensing application scenarios we present in this Thesis.

The evaluation of the content presented in this chapter will be done in the context of fluid dynamics sensing application scenarios in the following chapters.

Chapter 5

Simulation Model for Fluid Dynamics Sensing

5.1 Introduction

In sensor networks, simulation models play a major role in allowing the users to debug protocols and evaluate performance metrics in complex scenarios that cannot be analyzed with other means. Simulation models also allow the user to evaluate traffic control and design strategies without committing expensive, time-consuming resources necessary to perform the same on a real deployment.

In order to pursue a simulation environment for XDense, we developed a simulator that is customizable enough to allow implementing its architectural specificities, and, at the same time, complete enough to enable a detailed analysis of its performance.

In the next sections we detail its implementation methodology and architectural options behind it, and we also present the tools we developed for importing fluid dynamics data and for post-processing the simulation results. Then we perform experiments with XDenseSim to evaluate the different feature extraction algorithms we propose. The source code of the simulator, as well as the pre and post-processing tools, were presented in [5]; it is open source and available online at [126].

5.2 Simulation Model

We have built a simulator for XDense on top of Network Simulator 3 (NS-3) [127] that we name XDenseSim. It allows us to evaluate many dimensions of XDense, including its protocols, distributed processing algorithms and various networking performance metrics.

In order to obtain a network model with characteristics and performance that faithfully approach the real system (which ever is under consideration as use case), the design of XDenseSim is customizable. This configurable nature is with respect to links, packets, communication ports, router, protocols and applications; This makes XDenseSim suitable to other 2-D mesh network architectures (NoCs for example). Because its based on NS-3, XDenseSim is scalable, and allows simulations of very large networks with low processing cost; this is an important criteria arising from use cases. It also makes available various metrics to study the network performance, including, network bandwidth, links utilization, load distribution, queue sizes, end-to-end delays, or any other metric of interest.

XDenseSim also provides the means of debugging distributed feature extraction algorithms. For that, we provide the means to "feed" each node on the simulated network with data from the phenomena of interest, in order to create a simulated sensing environment that reproduces the real scenario.

In our case, we use data from an airflow phenomena in order to simulate our study case reliably, with application-specific temporal and spacial granularity. We provide a framework that allows importing data from reliable fluid dynamics scenarios into XDenseSim from different sources.

We use NS-3 as the base of our simulator because of its proven scalability, which allows simulating very large networks with a low computational cost. Because it is open source, it also allows to easily couple the required tools to import sensing data from computational fluid dynamics applications and tools.

The different abstractions of XDenseSim were implemented as independent classes that inherit basic functionalities from its NS-3 parent classes. The main functionalities inherited relate to: (a) the logging system that provides means to selectively access runtime statistics from numerous sources; (b) the tracing system that provides means to selectively access packet exchanges statistics from various sources; (c) the callback system that allows implementing and reusing multiple-layer implementation with ease; and (d) the possibility of integrating custom-made C++ classes into the simulator's discrete-events system. Apart from these general functionalities inherited from the parent classes, each custom class has its own specificities which will be discussed in this section.

XDenseSim classes hierarchy is as shown in Figure 5.1, which shows the simulation model components and the pre and post-processing tools developed. We explain each of these components next.

Node: We use the main *Node* class provided by NS-3, with no modifications. The *Node* class is the highest in the hierarchy, and serves mainly as a container that keeps track of

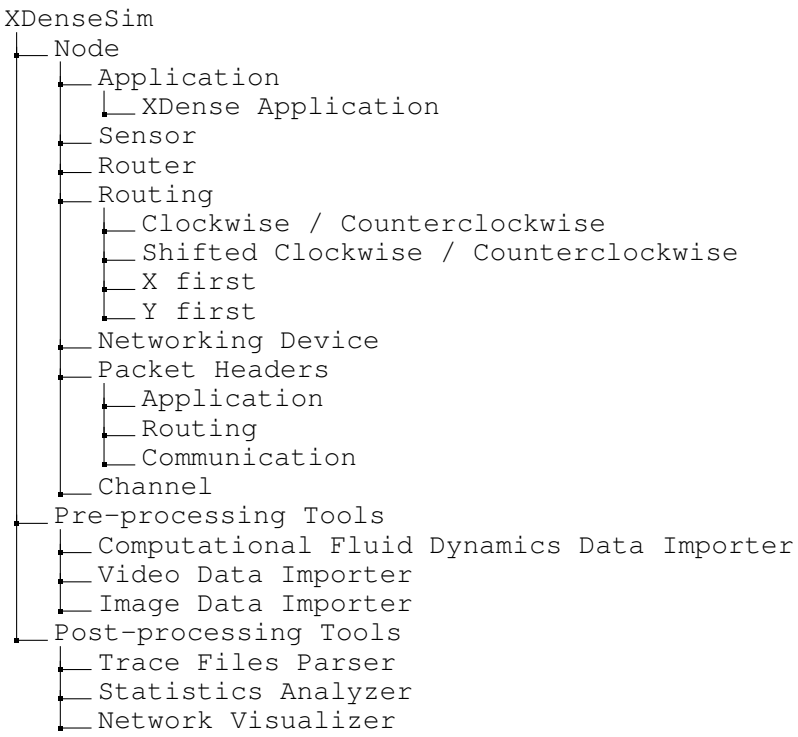


Figure 5.1: XDenseSim list of classes and their hierarchy.

other class instances that get "installed" into it. This association helps to confine each class instance to inside the node it is contained in, while also enabling internal communications to happen more easily by using function calls and callback mechanisms provided.

Application: Inside the *Node*, at the top layer it is the *Application* class, which implements the application protocols presented in Chapter 4. This layer is only able to communicate with the *Sensor* and *Router* classes, in order to sample data and communicate through the router.

Sensor: The *Sensor* class is the interface of each node to its spatial and temporal sensing data. We implement several pre-processing tools to attain this data from different fluid dynamics data sources. The mechanism of importing and pre-processing the input data is discussed in Section 5.2.1.

This is an interchangeable model, potentially useful to any NS-3 module in which the nature of the sensor's data would influence on the network operation.

Router: The *Router* class is responsible for packet IO and routing. It is also a container that keeps track of the node's *Application* and *NetDevices* instances, to transmit and receive packets to/from them, using function calls and callbacks. The routing algorithms

are implemented in a separate class *Routing* which is a static class, common to all nodes in the network.

The router does arbitration in case of contentions and is configurable (for example, round-robin or FIFO can be used at the input or output ports). To keep our experiments consistent, we only use FIFO arbitration to serve packets at output ports.

All *Application*, *Sensor* and *Router* components are based on classes that inherit the NS-3 Application Class. This provides the interchangeability required by these components, so that one or more instance of them can be "installed" at any node, and communicate between each other.

NetDevice: The *NetDevices* are the communication interfaces to simulate bidirectional serial links as UART ports. They implement communication characteristics like baudrate, jitter, queue (at input and/or output) and maximum queue size. *NetDevices* are meant to simulate resource-constrained hardware, present in COTS microcontrolers. For example, a standard UART communication port.

Packet Headers: The packet structure of our simulation model is as defined in Section 4.3.2 (Table 4.1). It is constructed *in situ* by the application, *Router* and *NetDevice* instances.

At its creation, the packet contains only the payload and the checksum footer. Three different headers are added to the packet before its transmission, specifying the communication protocols utilized. Each of the three headers correspond to the application protocol (AP), routing protocol (RP) and communication protocols (CP). These headers are added to the packet by the *Application*, *Router* and *NetDevice*, respectively, as the packet cross these layers.

Channel: The *Channel* class is the bottom layer, meant to simulate the serial link. The link model allows to simulate interference and eventual link failures, even though, as part of the assumptions of our model, we consider a perfect channel (see Chapter 4.3).

Two channel instance are shared between two interconnected nodes, one in each direction. This means that, each *NetDevice* is connected to two channels each, whereas each node is connected to eight channels, two to each neighboring node.

Illustrative XDenseSim example: Figure 5.2 shows the main steps involved on the simulation of a *Data Announcement* between two nodes through one hop using XDenseSim. The transmission starts at the top-left, at the sender node, whose application layer starts by reading its sensor (steps 1 and 2), creating a packet and then forwarding it to the

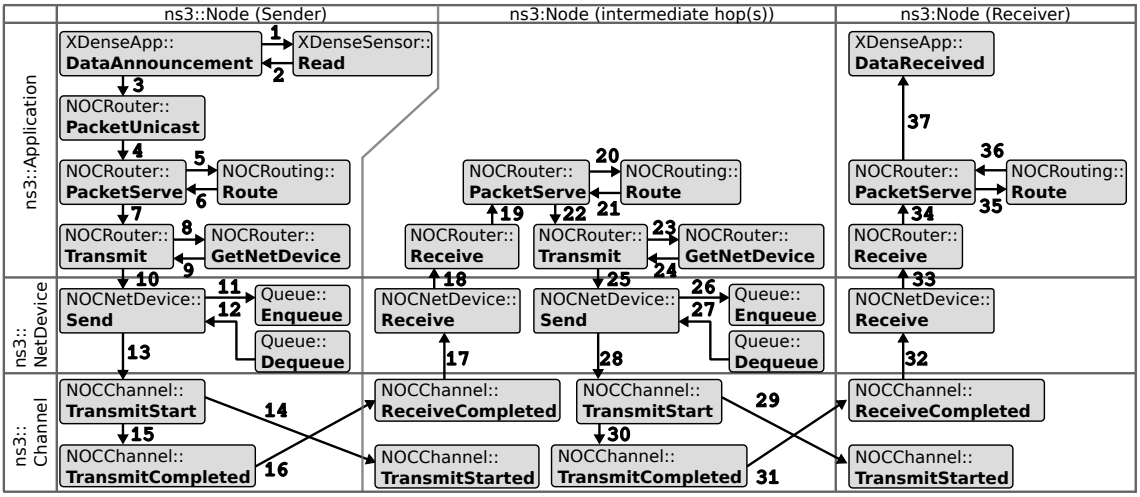


Figure 5.2: XDenseSim example: main steps involved on the simulation of a *Data Announcement* between two nodes, unicast with one intermediate hop.

router (step 3). At this point the packet contains the sensor data, destination and origin addresses, and the intended application protocol; in this case a *Data Announcement*.

After receiving the packet (step 4), the *Router* adds a header to the packet with the communication protocol chosen (unicast). Given the application and communication protocol, and the destination of the packet, the route is calculated (steps 5 and 6), and the packet forwarded internally to be transmitted (step 7). The last step inside the *Router* consists of fetching the corresponding *NetDevices* (steps 8 and 9) and forwarding the packet ready to send to each *NetDevice* individually (step 10). Packets received by each *NetDevice* are queued (step 11); the first on the queue is dequeued (step 12) and transmitted to the next hop (steps 13 to 16).

The channel layers of both nodes interact twice during a transmission (between steps 14 and 16), one at the start of the transmission (step 14), and one at the end of the transmission (step 16). This is an implementation detail, meant to allow nodes to take actions on serving consecutive packets.

Since we do not use input queues, the packet received by the next hop is then immediately forwarded by the net device to the router to be served (steps 17 to 19). The *Router* analyzes the header content to forward the packet to the next hop following the same logic previously presented (steps 20 to 31). Again the packet is received and forwarded internally up to the *Router* (steps 32 to 34). In this case the receiving *Node* is the final destination of the packet, so it is forwarded to the *Application* (steps 35 to 37).

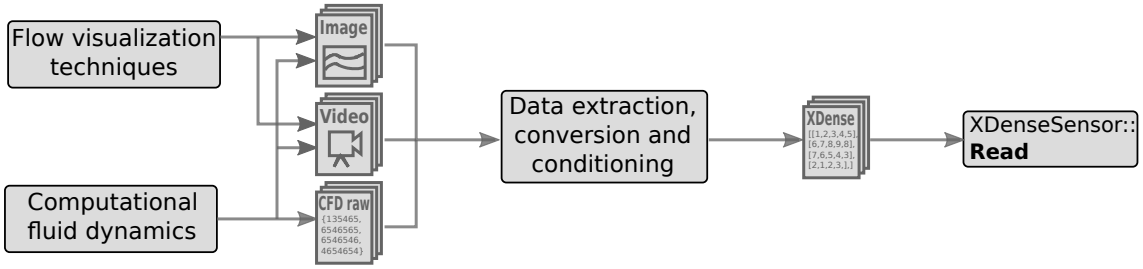


Figure 5.3: Steps required to import data from different external sources into XDense sensor input module.

5.2.1 Pre-processing Tools

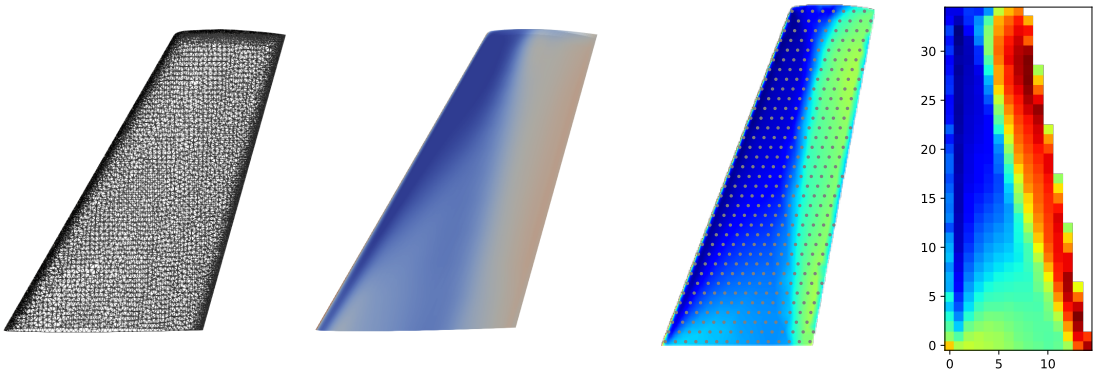
To analyze the XDense performance on airflow sensing, we need to provide XDenseSim with spatial and temporal data from a reliable source. For that purpose we feed data into XDense node's sensor, that correspond to the environmental fluid dynamics data that is to be measured. For our AFC use case, the phenomena to be measured is the pressure distribution on the surface of a wing (where XDense is deployed) due to airflow.

In fluid dynamic studies, data is often presented visually, either as static representations of the studied phenomena or as videos used to depict the phenomena temporally. The main reason is the large storage requirements of raw Computational Fluid Dynamics (CFD) simulation data. The images and videos can be the result of post-processing of data from a camera footage of real airflow [7] or from CFD simulations converted to image or video files.

For camera footage, researchers use special visualization techniques in laboratory conditions, usually in wind tunnels [7]. These techniques provide means of extracting detailed visualization data from the real phenomena, which is hardly achievable using CFD models, as discussed in [128].

CFD simulations, on the other hand, allow extracting image and video files after post-processing the raw data files. It is also possible to work directly with the raw data files, with no influence of data losses introduced by image compression or noise introduced by camera capturing limitations. The draw back is the excessive amounts of computing power and storage required to obtain high temporal and spatial granularity. This becomes affordable only in simple situations, and/or when the dynamics of the phenomena and its evolution occur with low temporal complexity and/or in small time-windows.

We provide three means of importing fluid dynamics data into XDenseSim: (i) from static images; (ii) from video files and (iii) from CFD simulation data. Figure 5.3 shows the sequence of steps required to import data from the sources mentioned.



(a) (b) (c) (d)

Figure 5.4: (a) Pressure distribution over a wing's surface; (b) Data of a single time-frame, from Computational Fluid Dynamics (CFD) simulation, as input for XDense; (c) Sensors displacement; (d) Normalized data, as seen by each sensor.

The process of importing the data includes intermediate steps meant to extract, condition and convert the data to XDenseSim *Sensor* input format as follows.

We define a spatial region of interest at the data source, from which we extract data points, according to the desirable sensing granularity (number of nodes in the network). We extract data only from the points in which sensor nodes would be physically located (having defined the deployment region and granularity).

Following, we condition the data extracted by converting it from its original full scale into a new one, that corresponds to the one of the real sensor considered. The data new scale is represented using and integer scale, ranging from 0 to 2^{res}, where *res* is the sensor resolution given in number of bits.

Finally, the data is converted to a specific file format, which can be imported by XDenseSim. This is a text-based custom format, suitable to both static and dynamic data inputs. In essence, it is a 3-D array, with the first two dimensions representing a matrix in which each cell represents a node, whereas the third dimension is the time. It is important to remark that these steps apply to scenarios with both single and multiple frames of data (both from CFD, image or video input), with the difference that it is applied to every frame, which are aggregated into a single XDenseSim file.

One approach is to extract temporal data from a video file that shows the data of interest. Figure 5.5 shows the graphical tool used to extrat data from a video that shows a planar air-flow over time.

A more elaborate approach consists of using 3-D CFD raw data to generate the input for XDenseSim. The CFD simulation scnario we use consists of an ONERA M6 wing in

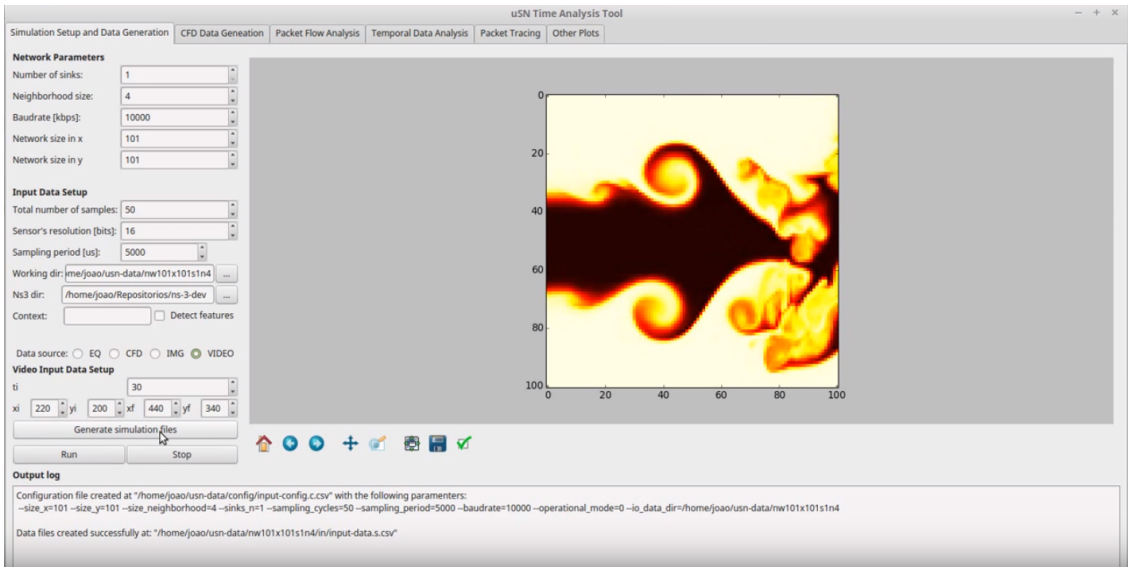


Figure 5.5: Tool for generating temporal sensor data from a video files of a CFD simulation.

viscous flow simulation [129].[1] This workflow is shown in Figure 5.4.[2] It consists of the following steps:

(i) Generate wing model: A 3-D mesh structure of a wing is generated and imported into the CFD simulator (Figure 5.4(a));

(ii) Simulate wing performance: The CFD simulator is run to simulate a wing pitching through a high speed air flow. Temporal pressure and temperature data of the surface of the wing are produced. (Figure 5.4(b));

(iii) Extract sensor data: These temporal pressure and temperature data are extracted, but only from the points in space that correspond to the XDense node deployment. Figures 5.4(c) and 5.4(d) illustrate sensor deployment and sensor data, respectively. The coordinates in Figures 5.4(d) matrix refer to node's absolute location coordinates on the network (represented by the gray dots).

We use the SU2 integrated computational environment for multi-physics simulation [130] to generate this data.

Note that there are fewer nodes per row towards the tip because of the wing's tapered shape. This results in a network with uneven distribution of nodes. This should be kept in

[1]The ONERA M6 wing was designed as an experimental geometry for studying three-dimensional, high Reynolds number flows with complex flow phenomena (transonic shocks, shock-boundary layer interaction, separated flow). It has become a classic validation case for CFD codes due to its simple geometry, complicated flow physics, and availability of experimental data.

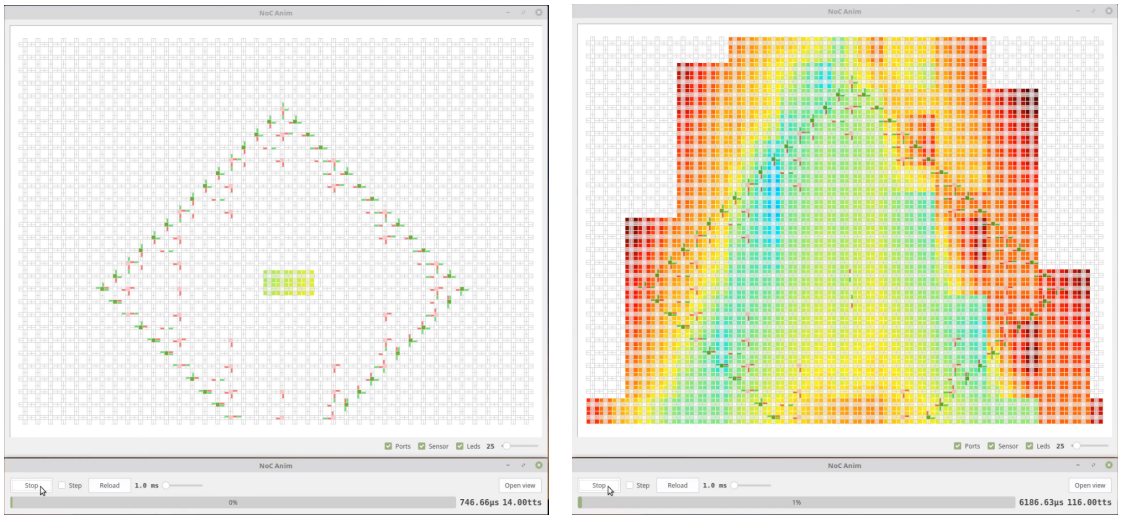
[2]Note that, for simplicity, Figures 5.4(a-d) shows only the top side of the wing.

mind when choosing the routing protocols, that should address potential issues related to the absence of nodes in the matrix.

5.2.2 Post-processing Tools

Post-processing tools are a key component of the simulator, since they allow the user to debug, analyze and extract usage and performance metrics.

We provide a set of tools to enable an extensive analysis of XDense. In this section, we showcase some of these tools, which are used throughout this research work ir order to extract each of the results presented.



(a) (b)

Figure 5.6: The node in the center (sink) requests and receives back compressed data from clusters. (a) and (b) show different snapshots of the reconstruction of the data by the sink, as the data is received.

We provide a sensor data visualizer for visualization of sensed data for debuging. It shows nodes activity overall, from the processors, networking device and sensed data. Figure 5.6(b) shows two snapshots of a network deployment with $40 \times 30 = 1200$ nodes on the bottom and top of a wing (seen unfolded longitudinally at the front of the wing). The input data is obtained from CFD, as detailed in the previous section. In this scenario, the node in the center (sink) requests and receive back compressed data from clusters. The data is reconstructed as it is received by the sink.

The first snapshot (Figure 5.6(a)) shows the initial time instants. The first request by the sink is still propagating on the network and the sink has only received data from the nearest clusters. Figure 5.6(b) shows the same scenario, some time instants later, after the sink reconstructing the data from the entire (active) network.

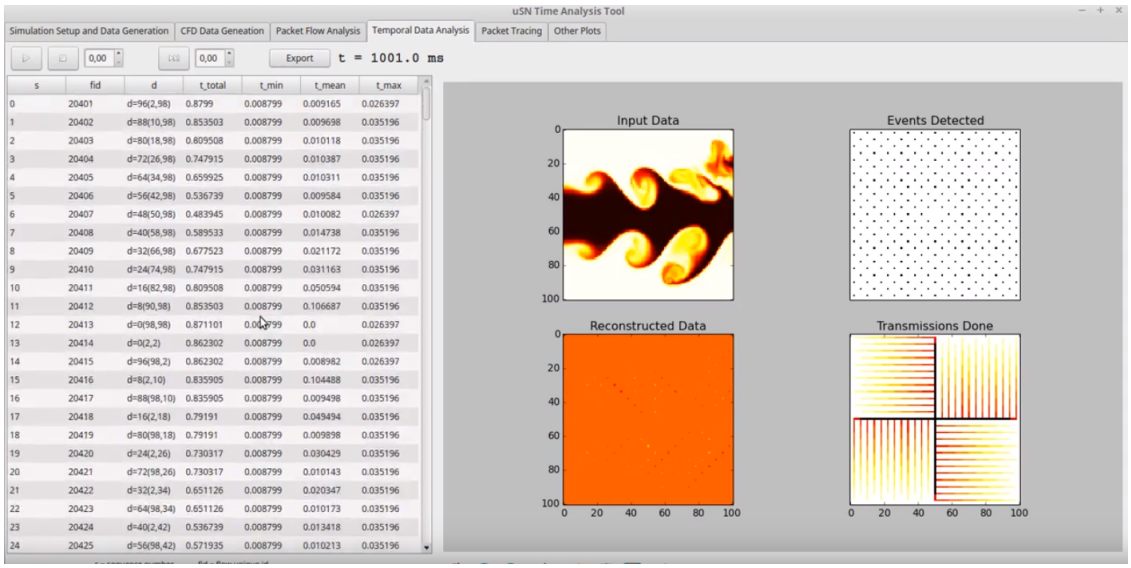


Figure 5.7: Packet trace (in the left) and extracted information on the right. It shows the input data and node's activity heatmaps.

Figure 5.7 shows the packet trace in the left, and the extracted information on the right. It shows the input data and node's activity heatmaps.

Figure 5.8 shows the trace of a single packet traveling on the network. It shows the time instant at which each transmission occurs, and consequently the time the packet stayed at each hop because of queuing.

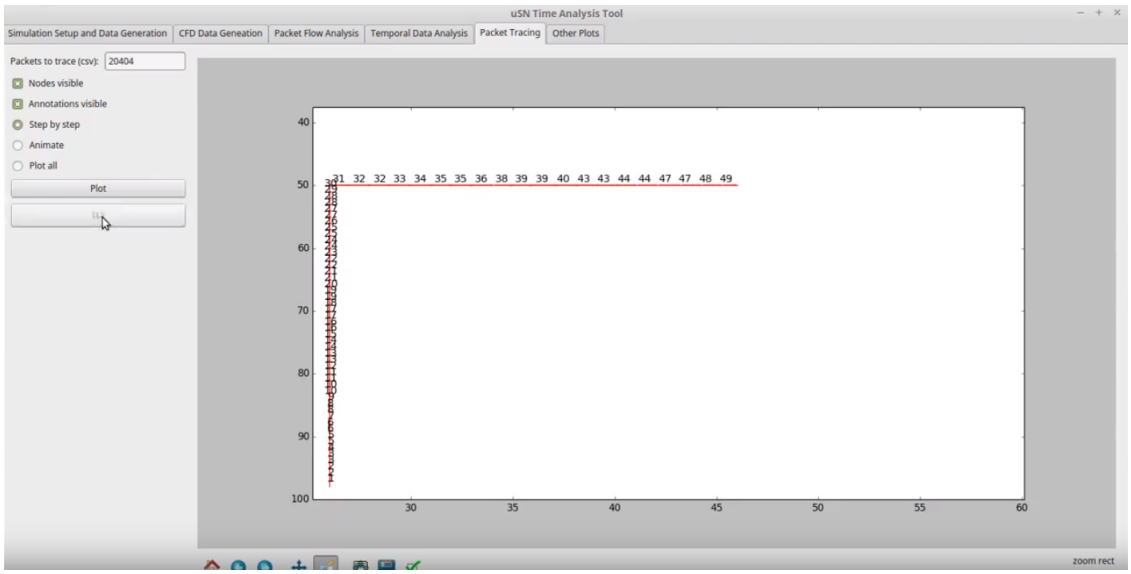


Figure 5.8: Single packet trace. It shows the time instant at which each transmission occurs.

5.3 Performance Evaluation With Airflow Input Data

We use XDenseSim to evaluate XDense's distributed processing capabilities and to provide metrics on the data acquisition delay, load on the network, queue size and reliability of acquired data.

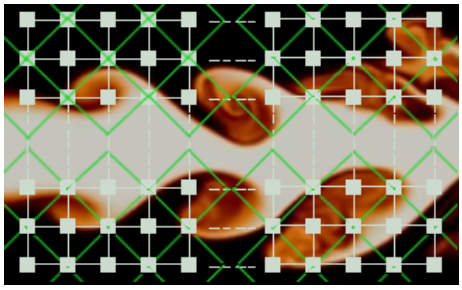


Figure 5.9: XDense network superimposed on the CFD dataset snapshot from [6], showing clustering for $n_{\text{radius}} = 1$.

For the representative input phenomena, we use a planar airflow emitting from a nozzle into a room filled with air. Figure 5.9 shows the input data before being imported. This scenario is commonly used to study the evolution of airflows. It allows capturing the characteristics and the role of the vortices on turbulent flows and at the transition region, in order to better understand the phenomena [25]. Studies based on the planar airflow scenario are important due to their low complexity, which allows it to be performed both using real airflow and imaging setups, but also using CFD with relatively low computational complexity. Its importance is also related to the possibility it gives on comparing

Parameter	Value
Network dimension	101×101
Num of Nodes	10200
Sinks	1 (centered)
Clusters size (n_{radius})	0 to 4

Table 5.1: XDense simulation parameters.

experiment results obtained using imaging setups and CFD, commonly used to validate CFD models [6].

The planar airflow scenario, and the wing scenario presented earlier in Introduction (Figure 1.1), exhibit similar mixing phenomena of laminar and turbulent flows, and this transient region is of interest in both cases. Thus, we use the planar airflow as our validation scenario.

We perform four experiments to demonstrate and evaluate the different feature extraction algorithms proposed. The first experiment is a straightforward case in which we examine the efficacy of XDense in sensing, extracting in-network compressed data and reconstructing the sensed data. We compare the extracted data with reference data in order to measure the reliability of the extracted data. In this experiment we use CFD data as our input. This allows us to get a base case for XDense's performance.

In a second experiment we perform distributed feature detection using XDense on the same CFD input data. We detect the transition region in the planar airflow, and we reconstruct the sensed data. The reconstructed data is then compared with the reference data in order to measure the reliability of the extracted data.

In a third experiment, we also detect the transition region on a planar airflow scenario. But instead, we use airflow data from an imaging setup. We evaluate the accuracy of the detection by comparing it with a reference from literature, obtained using image processing techniques to perform the same detection.

These three experiments are all temporally static in nature, and we analyze only one snapshot of an input phenomena. In the fourth experiment, we look at how XDense handles CFD temporal inputs.

The XDense deployment inter-space (that is, space between nodes), depends on the minimum size of the observed phenomena. Therefore, this has to be smaller than the minimum turbulent structure size [29]. Therefore, we use a network with $101 \times 101 = 10201$ nodes with one sink in the center. With believe that this density is more than enough to observe the features of the airflow we are interested. Deployments with less nodes are tested later in Chapter 6. Table 5.1 summarizes the system parameters defined, and the ones varied in order to evaluate XDense.

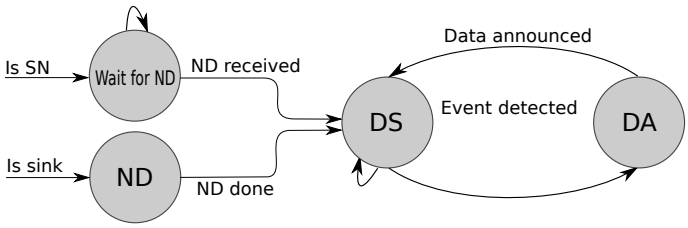


Figure 5.10: State diagram for an XDense node.

5.3.1 Distributed Application Execution

A naive solution to the dense sampling problem is to request each node to continuously sense information about the airflow and send it back to one or more sinks. The information collected by the sinks is reconstructed and used to compute the airflow's properties. Clearly, this approach provides the highest resolutions possible, but also generates a tremendous load on the network, requiring large buffers in each node, and imposing significant delays between the time at which the information is requested and the time it is fully received by the sink. Moreover, the sensed information may have a maximum lifetime.

Instead, we use XDense to efficiently build a global picture of the airflow by organizing the nodes in clusters and perform local data processing. In each cluster, one node serves as the cluster-head node. It performs data aggregation within its cluster and is responsible for processing (and/or compressing) the data locally to send only meaningful information to the sink.

In another appplication scenario, we program the cluster-heads to inform the sink *only* upon the occurrence of meaningful events (e.g., airflow changes from laminar to turbulent and conversely). Keeping in mind that routing protocols should ideally exploit the network topology to avoid congestion; we use the application protocols defined in Chapter 4 to allow coordination of clusters by the sink.

These applications are based on three operative principles: (1) the nodes are clustered and one node in the center of each cluster (cluster-head) is in charge of aggregating, pre-processing the data and sending it to the sink; (2) the execution of the application is divided logically in subsequent phases, each one using different application protocols; (3) the network uses different routing protocols to guarantee spatial isolation between the clusters.

The distributed protocol consists of three operating phases (Figure 5.10), namely:

- Phase ϕ_1: *network discovery* (*ND*);
- Phase ϕ_2: *local data-sharing* (*DS*);

- Phase ϕ_3: *remote data-announcement* (*DA*).

These three different phases are briefly explained next.

Network discovery (ND): At the *ND* phase, the network is initialized, with the sink broadcasting its location. On the reception of the packet, given the application protocol defined on the packet, nodes compare their relative position with the size of the cluster specified in the packet, and whether if they are supposed to become cluster-head or not (depending on their relative position inside their own cluster), and if not where is the nearest cluster-head located.

The parameter n_{radius} defines the size of the cluster, by defining the maximum distance (in terms of Manhattan distance) a node can be from the cluster-head to be considered part of that cluster. It is an application dependent parameter, and it is selected according to the expected characteristics of the phenomena to observe. The naive case is $n_{\text{radius}} = 0$, which means that each node is a cluster-head of itself, sending data directly to the sink without aggregating from any other node. The greater the value of n_{radius}, the greater the number of nodes N in the cluster. The number of nodes N in a given cluster is given by:

$$N = 4 \times \sum_{i=1}^{n_{\text{radius}}} i = 2 \times n_{\text{radius}} \times (n_{\text{radius}} + 1) \tag{5.1}$$

This is the sum of nodes in each of the 4 quadrants of the cluster (except for nodes on the edges of the network, which are inactive).

For clustering, as nodes switch from *ND* to *DS* phase, they verify three conditions related to its location and cluster size (n_{radius}), in order to perform cluster-head election:

$$\begin{cases} x \bmod n_{\text{radius}} = 0 \\ y \bmod n_{\text{radius}} = 0 \\ (x+y) \bmod (2 * n_{\text{radius}}) = 0 \end{cases} \tag{5.2}$$

The rational behind this equation is that the three conditions are only satisfied for nodes located in the center of a cluster with $n_{\text{radius}} = n$. In this case, the node set itself as cluster-head.

At the reception of this packet, nodes switch to the *DS* phase.

Local data sharing (DS): During the *DS* phase nodes continuously sense the environment and communicate their sensed values to their cluster-head by unicasting their value inside their cluster, so that each cluster-head receives values from all nodes on its cluster.

Remote data announcement (DA): Depending on the findings of the cluster-heads, according to the application goals, they may switch to the *DA* phase and send their data to the sink, switching back to the *DS* phase immediately after, and continuing the cycle. As the sink receives data from cluster-heads from the entire network, it is able to reconstruct the observed phenomena with increasing accuracy and coverage.

5.3.2 Experiment I: Sensing Compressed Static CFD Data

In this first experiment, we evaluate XDense performance when extracting compressed data from the network, by reading aggregated data from clusters of nodes.

We aim at finding the balance between local transmissions and long range transmissions to sinks, that provides the best network load distribution and at the same time, giving satisfactory resolution of acquired data.

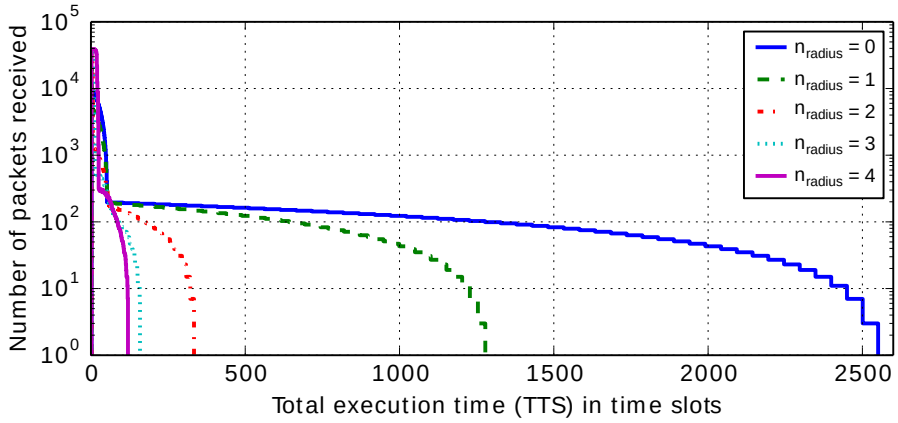
Clusters of nodes are established using the multicast alternative communication protocol in order to elect the cluster heads. For data aggregation, we use the least square regression [132]. This is a widely used data fitting algorithm that is suitable for embedded devices applications due to its low computational cost. It demonstrates significant data reduction and resulting increased performance. There are many other possible approaches for this goal; we use this one as a placeholder to demonstrate XDense.

Aggregation is done by cluster-heads, on data received from the nodes in the cluster, as follows. It does it by fitting a plane (of shape $z = ax + by + c$) to the data of the N nodes on the cluster, being (x, y) the position of the node inside the cluster, relatively to cluster-head, and z the measured value by that node. N is defined in Equation 5.1 as the number of nodes in each node's cluster. We calculate the coefficients a, b, c using the least square regression, used to determine the plane that best fit the data set. The least square regression finds the plane that minimizes the distances between itself and the points (x_i, y_i, z_i).

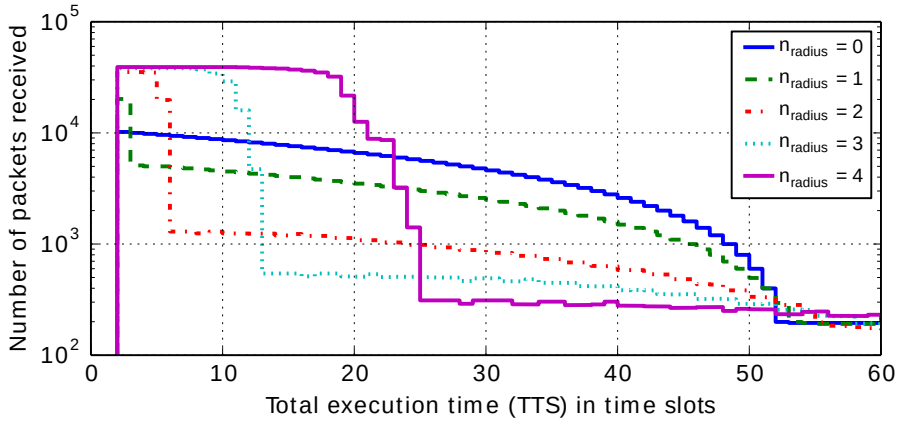
On switching to the *DA* phase, the coefficients of the fitted plane are transmitted to the sink in a single packet that carries a, b and c.

We evaluate the network load for different cluster sizes (n_{radius}). Figure 5.11(a) shows the number of transmissions on the entire network per transmission time slot (TTS) for different values of n_{radius}. Because packets have fixed size, the number of transmissions over time gives us information on the overall network load, which is proportional to the number of packets exchanged on it.

For $n_{\text{radius}} > 0$, a maximum load spike in the beginning (between $0 \leq t \leq 53$) is caused by the *DS* phase (shown in Figure 5.11(b)). This maximum load increases with increase



(a) Duration of entire simulation



(b) Zoom in on the transmission time interval (0 to 60)

Figure 5.11: Reading sensed data: Number of receptions over time for different values of n_{radius}.

of n_{radius}. Figure 5.11(b) shows a zoom in the time window $0 \leq t \leq 60$, showing the drop in communication activity as nodes switch from *DS* to *DA*.

Comparing $n_{\text{radius}} = 1$ and 4, this trade-off is due to the increasing number of *DS* and decreasing number of *DA* transmissions as the n_{radius} increases.

Next, we evaluate queue sizes for different values of n_{radius} and analyze the impact caused by the the variation of the number packets exchanged during *DS* and *DA* phases. We assume infinite queues, with no packet drops. Figure 5.12 shows that the queues caused by *DS* are greater for smaller values of n_{radius}. This is because as number of data announcements (*DA*) increases, congestion towards the sink also increases.

Additionally, while not shown, nodes in the path to the sink are those that suffer the biggest queues. As all packets compete for paths leading to the sink. The closer a node is to the sink, the longer its queue size. Figure 5.13 complements this analysis by showing the balance between the total number of *DS* and *DA* transmissions for different values of

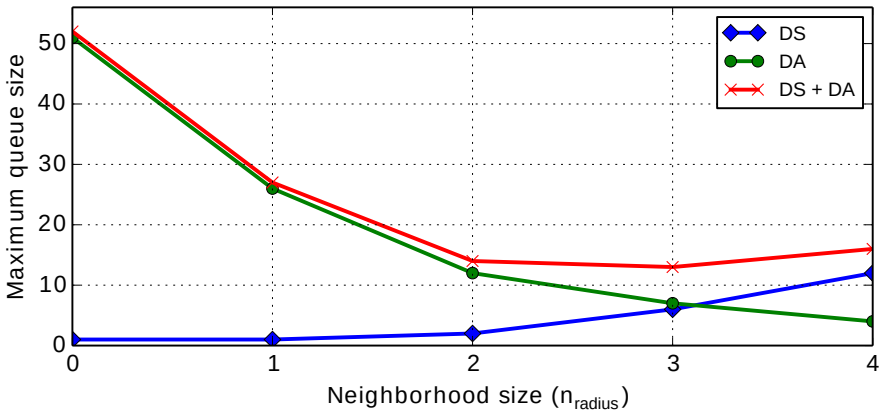


Figure 5.12: Reading sensed data: Maximum queue size for $n_{\text{radius}} = 1$ to 4.

n_{radius}.

Figure 5.14 provides a qualitative comparison of output data. For $n_{\text{radius}} = 0$, all nodes transmit their readings and this is the maximum possible resolution of the phenomena. Intuitively, greater the n_{radius} value, greater are the errors when fitting a plane to the data-set, resulting in greater data loss. In Figure 5.14(d), it is possible to see more clearly how the increase in the number of nodes in the cluster affect the quality of the sensed data.

Figure 5.15 shows a quantitative comparison of the trade-off between data accuracy and maximum total acquisition time delay. Accuracy is the mean square error (MSE) of the sum of differences between the best possible case (which is $n_{\text{radius}} = 0$) and scenario with n_{radius} varying from 1 to 5. The MSE measures the average squared difference between the sensed values and the values of reference.

The MSE grows with increasing n_{radius}, while maximum acquisition delay decreases. Looking at these results, $n_{\text{radius}} = 2$ seems to be a good choice, since the smaller turbulent structures are still distinguishable (as shown in Figure 5.14). It also represents a reduction in the maximum acquisition delay of approximately 75% of the worst case scenario, with $n_{\text{radius}} = 0$. With increasing n_{radius}, better data aggregation algorithms could lead to a smaller slope of the error, but the trend would remain the same.

We compare our results with Pressure Belt, a master-slave, shared bus based network proposed in [72], for monitoring pressure over an aircraft wing (presented earlier in Chapter 2).[3] The authors calculate the time required to read all the nodes simply as: *number of nodes*$\times$*packet duration*. Normalizing Pressure Belt results, and considering one mas-

[3]Pressure Belt is a strip, mounted crosswise on the wing of an aircraft, connected in one extremity to a coordinator, and has an embedded data-logger situated inside the airplane. It runs over two parallel full-duplex RS485 shared links, compatible with the IEEE 1451.2 Standard for a Smart Transducer Interface for Sensors and Actuators. Up to 255 nodes can communicate at 5 Mbps with packets of 48 bits each. By using a clock synchronization scheme, time division multiple access (TDMA) is used to communicate with the nodes.

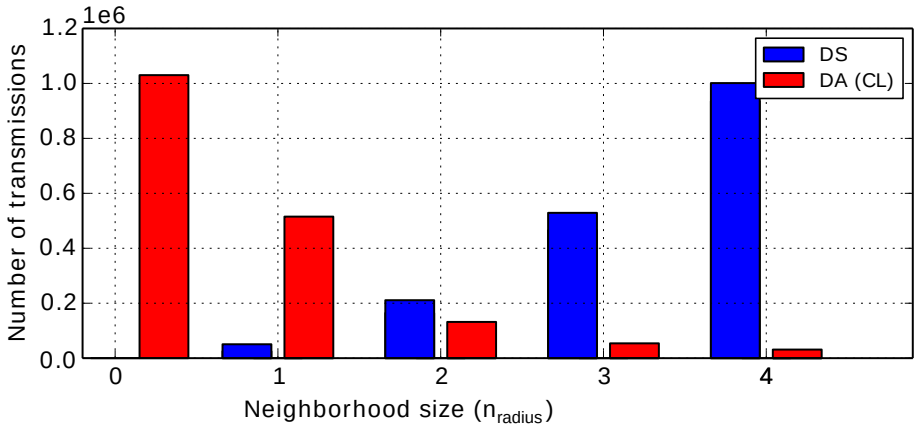
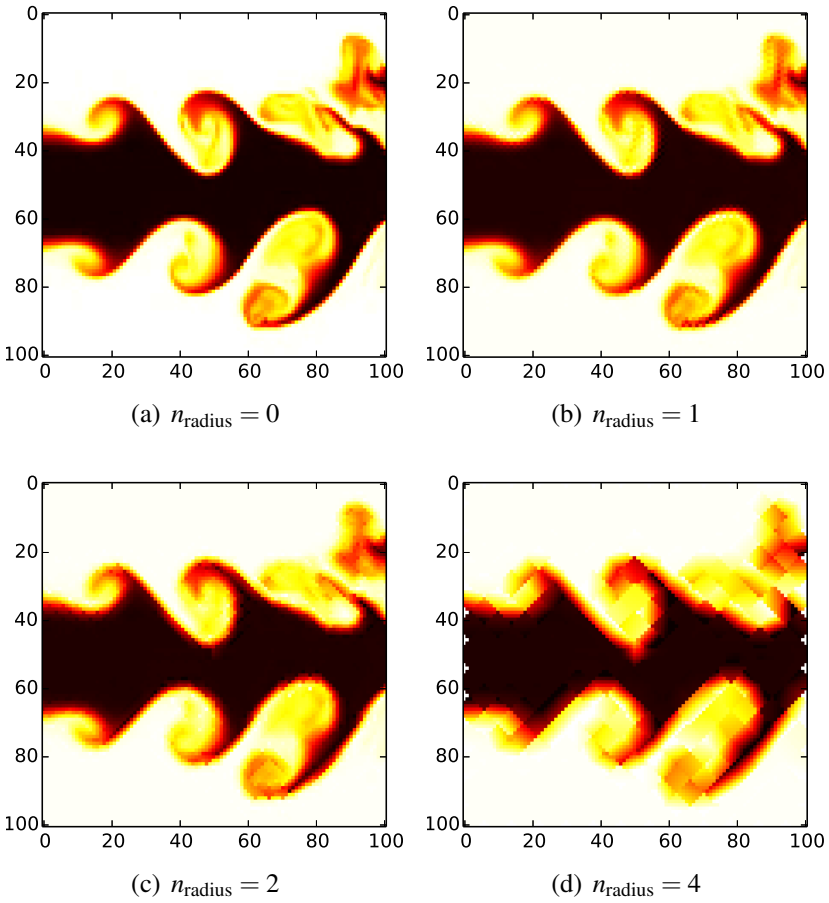


Figure 5.13: Total number of packets transmitted in the network with $n_{\text{radius}} = 1$ to 4, using clustering (*CL*).

ter for each quarter of all nodes, Pressure Belt would have the same performance of the $n_{\text{radius}} = 0$ scenario. Due to physical and electrical limitations, and due to the maximum address space, their solution may not be suitable for the density of nodes we consider. Also, their solution was designed for offline processing, acting as a simple data logger, making it inapplicable for real-time applications.



(a) $n_{\text{radius}} = 0$ (b) $n_{\text{radius}} = 1$

(c) $n_{\text{radius}} = 2$ (d) $n_{\text{radius}} = 4$

Figure 5.14: Reading sensed data: Extracted data for different values of n_{radius}.

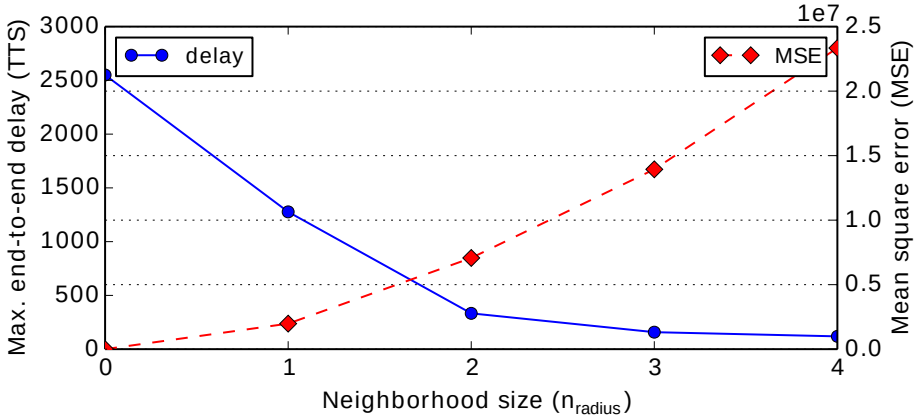


Figure 5.15: Reading sensed data: Trade-off between mean square error and maximum acquisition delay delay for different values of n_{radius}.

5.3.3 Experiment II: Detecting Transition Region on Static CFD Data

The response of the system in sensing data from the entire network, even when compressed, may not be satisfactory for some applications' spatial and temporal requirements. Figure 5.11 shows that even if we use $n_{\text{radius}} = 1$, total acquisition delay delay is still as big as 50% of the worst case ($n_{\text{radius}} = 0$).

In this experiment, we introduce and evaluate a distributed feature detection algorithm that is useful for our application scenario's real-time nature. We use a feature detection algorithm to detect the transition region of a flow. As explained earlier, the transition region is where the fluid transits from a laminar to a turbulent regime.

The feature detection algorithm is distributed as the detection happens independently at cluster-heads in the network. We show that this approach improves the response time due to localized distributed processing and thereby decreasing the load on the network.

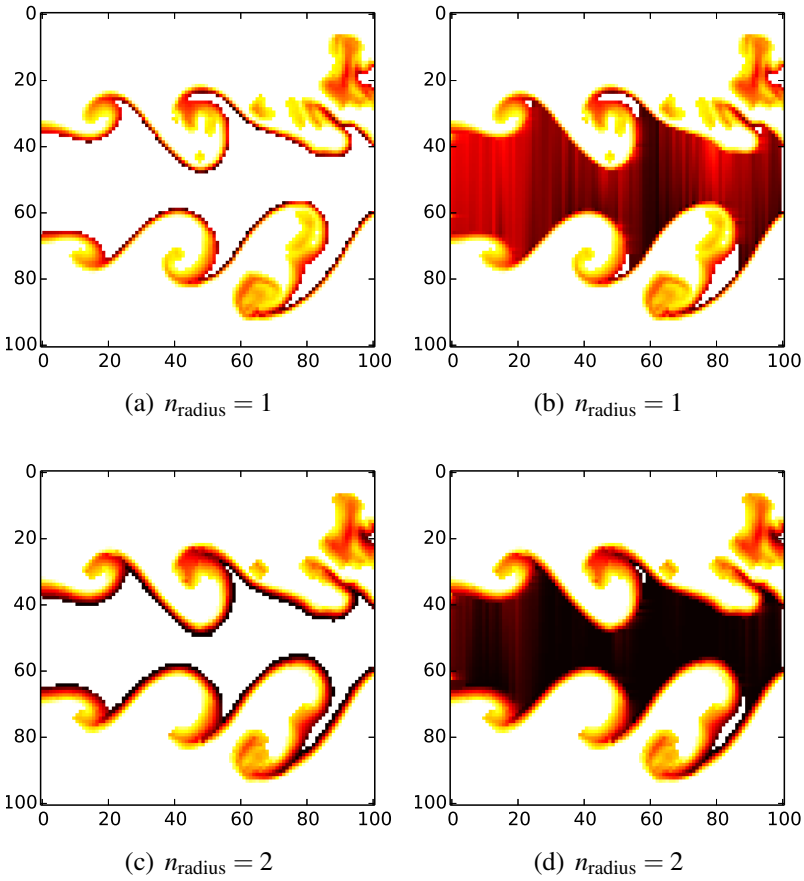
To detect features of interest at each cluster-head, we use an edge detection algorithm. We propose a variation of the Sobel operator that is widely utilized in the image processing domain [133]. The algorithm proposed fit to the XDense network capacity, since it is an algorithm that consumes few computation resources when compered to more complex image processing algorithms.

In brief, edge detection happens as follows. Each cluster-head performs a 2-D spatial gradient measurement on an image and emphasizes regions of high spatial frequency that correspond to edges. This measurement is the weighted sum of values of all nodes in the cluster, which is proportional to the distance from the cluster-head (rectilinear distance).

After receiving the data from its cluster, the cluster-head applies the edge detection algorithm to this data and checks if the result exceeds the predefined threshold. This threshold value defines the minimum sharpness of the edges that we want to detect (as defined by the application).

If an edge is detected, the cluster head again performs linear regression with the data gathered from its cluster (as in the previous experiment). This information is then transmitted to the sink, which is then able to reconstruct the scenario with the information received. The sink receives data only from cluster-heads that detected an edge; that is, where transitions occurred. In this case the sink is not able to build the complete picture of the event, but is able to estimate it.

To reconstruct the complete phenomena picture, the sink performs linear vertical interpolation on the aggregated data. The interpolation is done around the center region of the jet, between the upper and lower limits that define the transition regions of the air flow. Although we are interested on detecting the transition regions, we reconstruct the



(a) $n_{\text{radius}} = 1$ (b) $n_{\text{radius}} = 1$

(c) $n_{\text{radius}} = 2$ (d) $n_{\text{radius}} = 2$

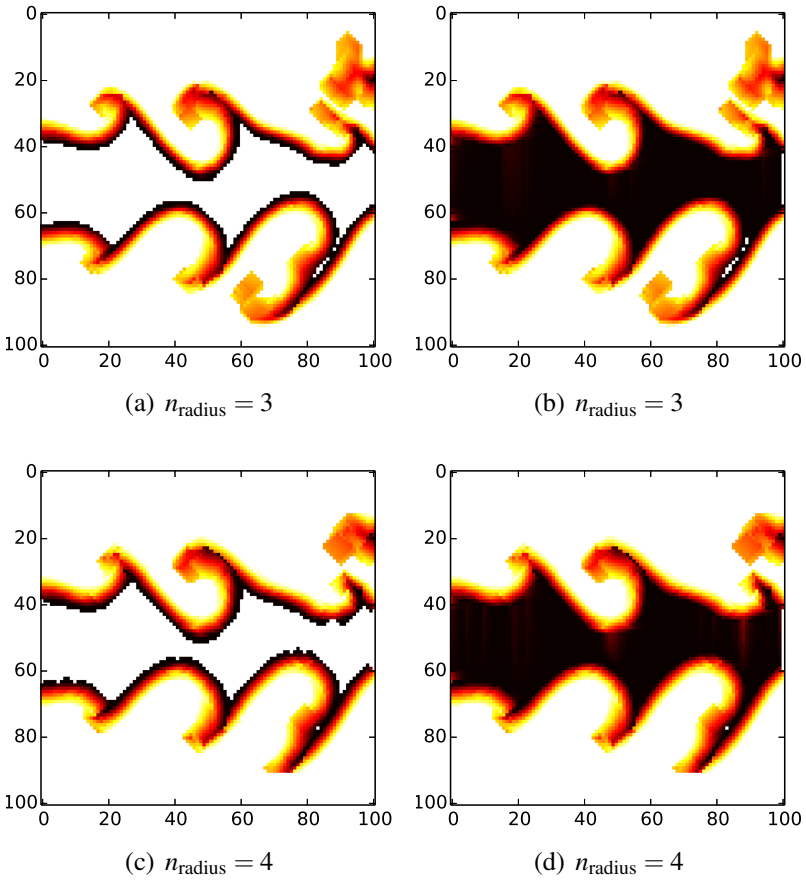
Figure 5.16: Feature detection: Extracted boundary data, and reconstruction of boundary data for $n_{\text{radius}} = 1$ to 2.

complete scenario to be able to compare the quality of the acquired with the previous experiment scenario reference.

Figures 5.16 and 5.17 show qualitative results of feature detection with our algorithms able to correctly detect the transition regions. As expected, smaller n_{radius} lead to more accurate detection, providing finer details about regions of greater turbulence (for example where vortices's indentations are). The same trend as from the previous experiment can be observed: With greater n_{radius} values, high frequency shapes on the phenomena data is degraded, like in the vortices's indentations.

We also analyze the impact of varying n_{radius} on maximum queue size and network load metrics. Results are shown in Figure 5.18 and 5.19. They show that queue sizes and minimum acquisition delay are inversely proportional to n_{radius}, as less nodes are transmitting to the sink to inform their findings. In this sense, these are similar results to the ones obtained in the previous experiment.

Figure 5.20 shows a quantitative comparison of the trade-off between MSE and max-



(a) $n_{\text{radius}} = 3$ (b) $n_{\text{radius}} = 3$

(c) $n_{\text{radius}} = 4$ (d) $n_{\text{radius}} = 4$

Figure 5.17: Feature detection: Extracted boundary data, and reconstruction of boundary data for $n_{\text{radius}} = 3$ to 4 for comparison.

imum acquisition delay. As in the previous experiment, the MSE grows with increasing n_{radius}, while maximum acquisition delay decreases. Again, by looking at these results, $n_{\text{radius}} = 2$ seems to be a good choice, since the smaller turbulent structures are still distinguishable (as shown in Figure 5.16). It also represents a reduction in the maximum acquisition delay of approximately 88% of the worst case scenario, with $n_{\text{radius}} = 0$.

The maximum acquisition delay is shown in Figure 5.19. The delays are much lower than that for sensing data from the entire network (previous experiment). For $n_{\text{radius}} = 1$, the delay drops ten-fold when compared to the previous experiment. Even more, the MSE also dropped by a factor of 2.

Furthermore, Figure 5.21 shows the balance between the total number of packets exchanged during *DS* and *DA* when performing feature detection. It is important to notice that both experiments have the same results for the *DS* phase, which is common among them. This figure gives us a sense of proportion to the impact of using different distributed data processing algorithms on the network load.

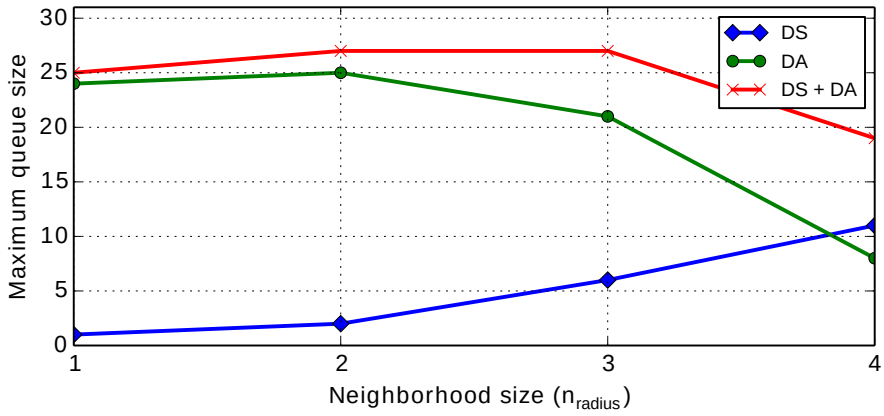


Figure 5.18: Feature detection: Maximum queue size for $n_{\text{radius}} = 1$ to 4.

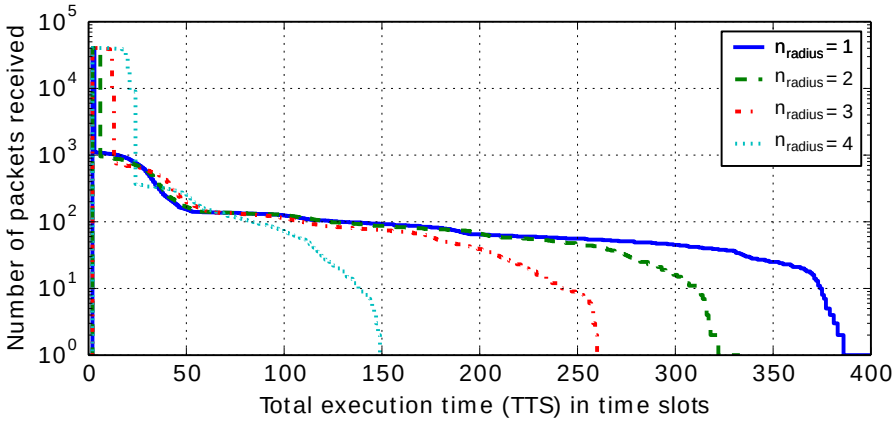


Figure 5.19: Feature detection: Number of receptions over time for different values of n_{radius}.

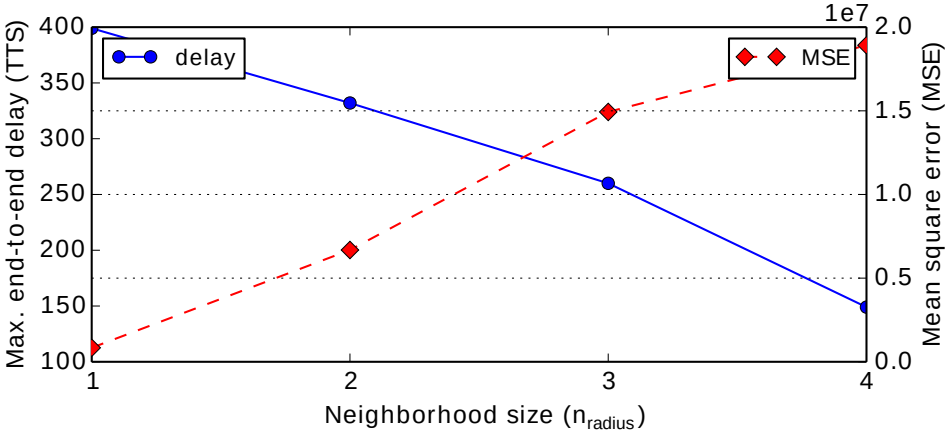


Figure 5.20: Feature detection: Trade-off between mean square error and maximum acquisition delay for different values of n_{radius}.

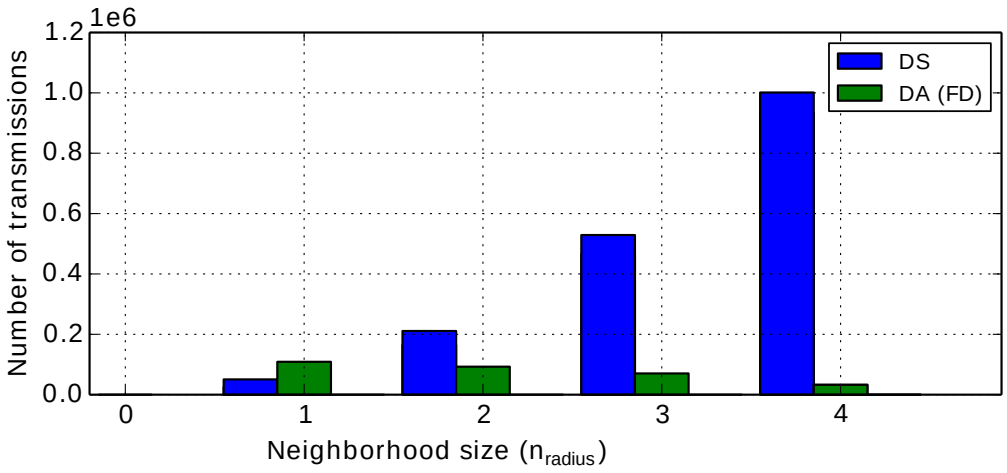


Figure 5.21: Total number of packets transmitted in the network with $n_{\text{radius}} = 1$ to 4, using feature detection (*FD*).

5.3.4 Experiment III: Detecting Transition Region on Static Image Data

Similar to the previous one, in this experiment we detect the transition region between turbulent and laminar air flow. In this case we use a different, but more realistic, input data source. Our data source is an image from [7], where the authors use a camera setup to indirectly measure the speed distribution of a planar airflow on free air by applying tracers on the flow.

The authors apply image processing techniques to detect the transition region, which is the envelope of the turbulent airflow. Figures 5.22(a-c) show the application of the processing techniques on an image in sequence. First, the image is binarized using a fixed threshold, then all the contours are traced, whereas only the contours with the greatest area are chosen; finally indentations are removed.

This is a common image processing approach for feature detection. We use this technique to detect the transition region in the original data, using common image processing techniques. The result (see Figure 5.22(c)) is then used as a reference, to compute the accuracy of the detection algorithms run on XDense. We do that as follows. First, we use the same flow snapshot image from [7] as input to XDenseSim. We use XDenseSim to detect the transition in network using the modified Sobel operator proposed on the previous experiment. The output is timely compared with the reference transition region computed.

Figure 5.23 shows the results of our simulation. It depicts the original snapshot (Figure 5.14(a)), the events sensed by our setup (Figure 5.23(b)), and the detected transition region by the sink (Figure 5.23(c)) superposed by the original output from Figure 5.22(c). We get a measure of our simulation's accuracy by comparing the distance between our findings to the reference transition region.

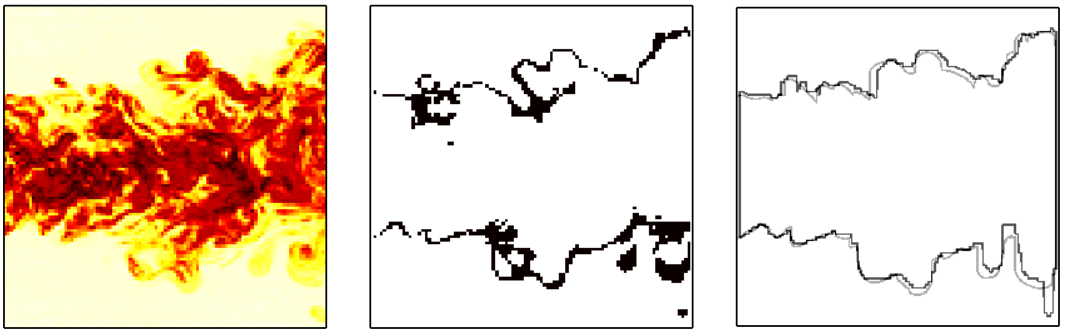
Using this process, we evaluate the behavior of our system under varying cluster sizes (n_{radius}). We are interested in observing: (i) the accuracy of the transition region detected; (ii) the total number of transmissions and (iii) the total acquisition time, which is the time required to perform the detection.

Figures 5.24 and 5.25 show (a) the detected transition by the sensor nodes and (b) the calculated transition by the sink. The images allow visual comparison that support our numerical results, and provide intuition for the impact on the accuracy of the detection algorithm by varying the cluster size between $1 < n_{\text{radius}} < 7$.

The cumulative density function (CDF) shown in Figure 5.26 gives a measure of the effect of utilizing different n_{radius}. It shows that for $2 < n_{\text{radius}} < 4$, more than 80% of the snapshot has an error lower than 3%. Moreover, increasing the cluster size does

(a) (b) (c)

Figure 5.22: Process steps for boundary computation described in [7]: (a) Original image; (b) binarized image; (c) contour tracing and contour smoothing.



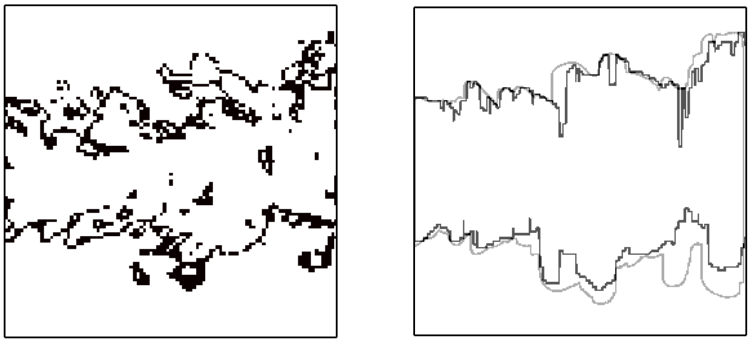
(a) (b) (c)

Figure 5.23: Processing steps of our network. (a) Is the phenomena as seem by our network, after downsampling the full resolution image to 101×101 pixels. (b) is the gradient detection by the sensor nodes with $n_{\text{radius}} = 3$, and (c) is the contour smoothing post-processing done by the sink.

not necessarily implies a greater accuracy, but generally in fewer events detected. The other way around, the minimum $n_{\text{radius}} = 1$ presents noisy detection, whereas intermediate n_{radius} values provide the best trade-off.

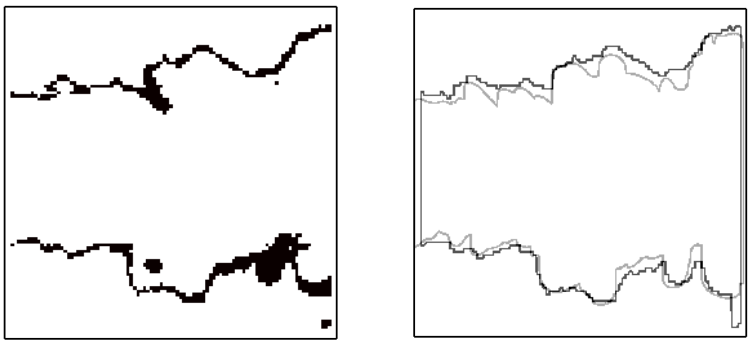
Figure 5.27(a) shows the trade-off between the maximum acquisition delay in TTS, and mean square error (MSE) of the transition region found. This gives an idea of how responsiveness varies for the different values of n_{radius}, and the impact on the accuracy. With $n_{\text{radius}} = 3$, MSE is minimum, with a minimum cost in time, when compared to $n_{\text{radius}} = 2$, which provides the best response time, but with a higher MSE.

Figure 5.27(b) presents the number of transmissions, therefore providing a picture of the actual load on the network. It shows that with increasing of the cluster size, the number of global transmissions (*DA*) goes down (although the local transmissions (*DS*)



(a) (b)

Figure 5.24: Processing steps of our network. (a) Transition region detection by the sensor nodes with $n_{\text{radius}} = 1$, and (b) is the contour smoothing post-processing done by the sink in black, superimposed on 5.22(c).



(a) (b)

Figure 5.25: Processing steps of our network. (a) Transition region detection by the sensor nodes with $n_{\text{radius}} = 7$, and (b) is the contour smoothing post-processing done by the sink in black, superimposed on 5.22(c).

increases accordingly). This in turn affects the total transmissions time (as can be seen back in Figure 5.27(a)).

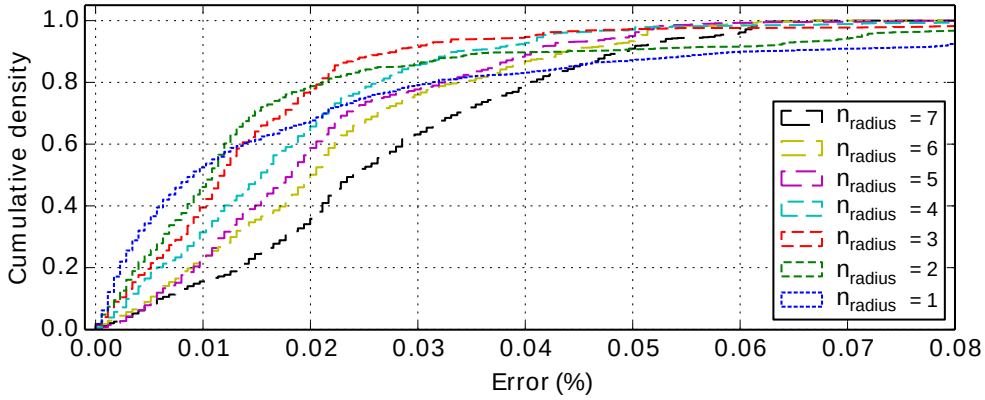
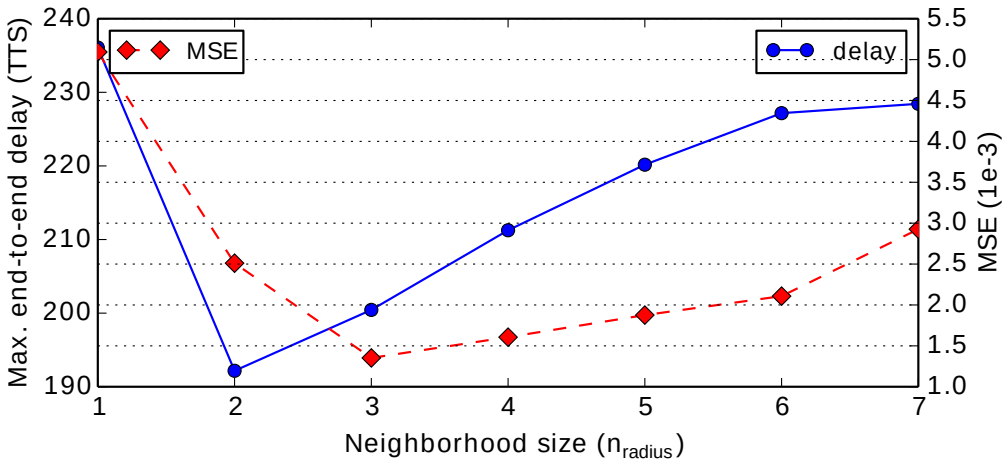
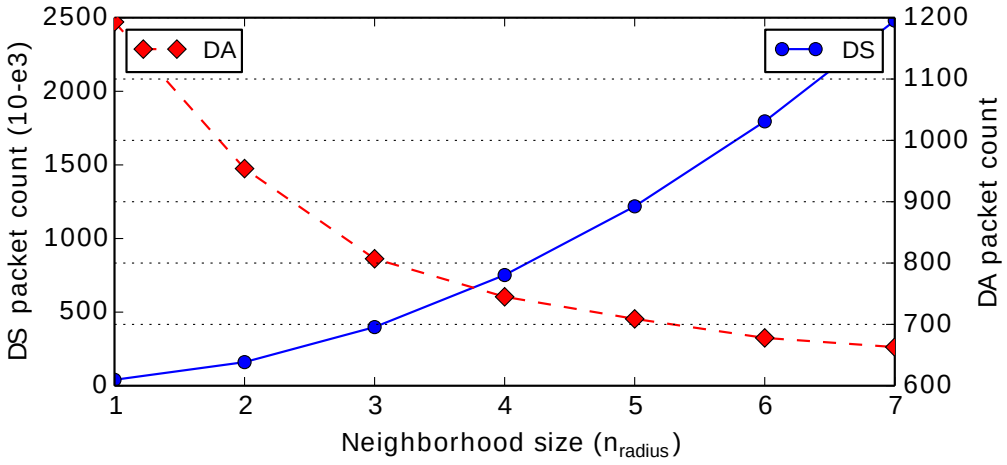


Figure 5.26: Cumulative density function for different n_{radius}, of the error between .



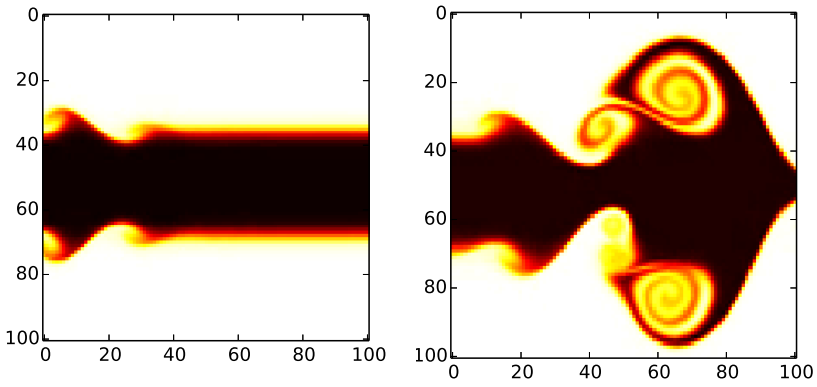
(a)



(b)

Figure 5.27: (a) Trade-off between mean square error (MSE) and maximum acquisition delay (TTS) for different values of n_{radius}, and (b) is the total number of transmissions for the different protocols, for the same values of n_{radius}

5.3.5 Experiment IV: Sensing Temporal CFD Data



(a) $t_{initial} = 0$ (b) $t_{final} = 120$

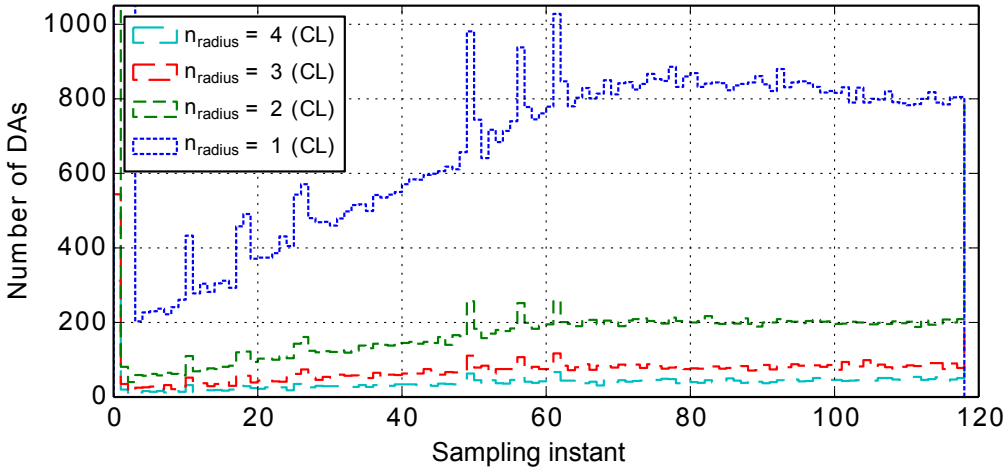
Figure 5.28: Real-time sensing: Input data at (a) $t = 0$ and (b) $t = 120$ sampling time slots (STS).

In this experiment, we address one of the larger goals of XDense, which is to sense and process data in real-time. We extend the previous analyses, which had static CFD input data, to temporal CFD input data.

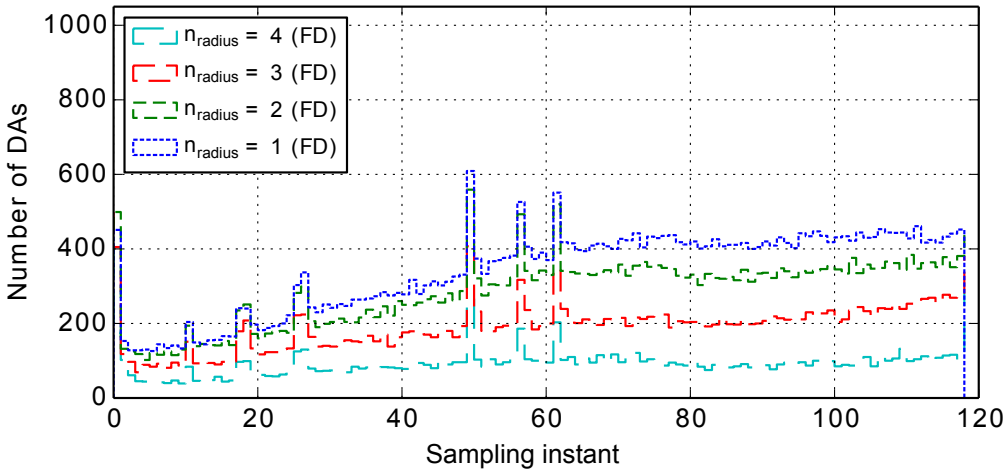
Figure 5.28 shows two sampling instants, out of 120 total, of a temporal phenomena that lasts for 600 ms. Within this period, turbulent airflow with vortices emerge. Our aim in this experiment is to see how our detection algorithms react to temporal data. We evaluate (using clustering) both data compression and feature detection algorithms of the previous experiments.

Figure 5.29 shows the network activity with respect to time using both data compression and feature detection algorithms. Note that in this case, we measure the time using sampling instants (instead of TTS). A sampling instant (SI) is each period of time in which a node sample new data. We assume that the interval between each sampling instant is long enough to fit enough TTSs, enough to all transmissions (DS and DA transmissions) due to the new sample; that is such that all transmissions of one sampling instant finish before the next one starts.

The network continuously monitors the evolution of the flow in this time period. At $samplinginstant = 1$, there is a peak of transmissions since all data is new and hence transmitted (this is also experienced in both the previous static experiments). For $n_{\text{radius}} = 1$, the initial activity peak lasts for approximately three samples, a period in which the network gets overloaded. A few additional samples are required for the nodes to service their queues. This is a situation to be avoided as it affects the response time of the system.



(a) Clustering



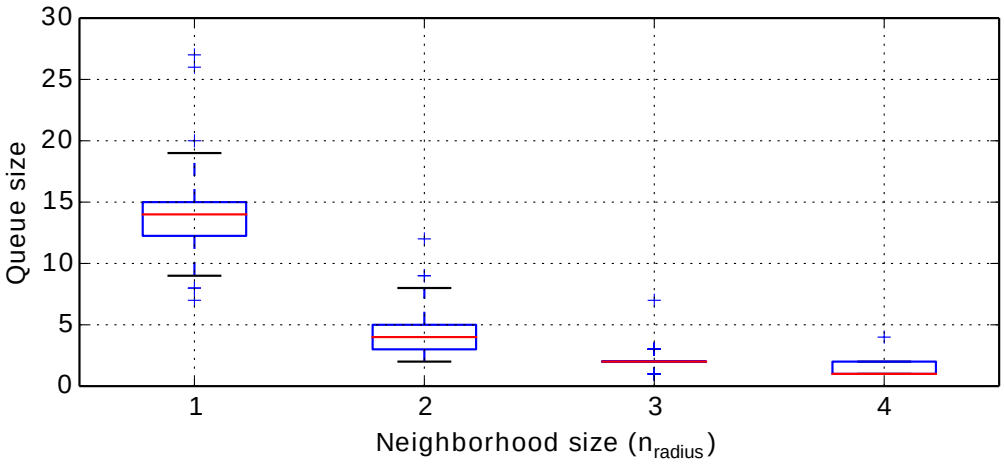
(b) Feature detection

Figure 5.29: Real-time sensing: Network activity, shown as number of *DA*s per sampling instant, over 120 sampling instants, for $n_{\text{radius}} = 1$ to 4.

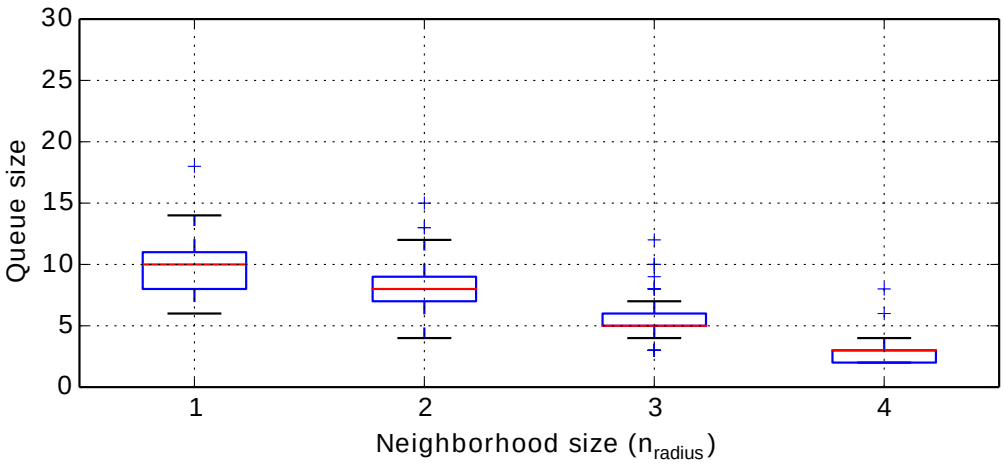
In all the scenarios, there is an increase in network activity as the flow becomes more turbulent. A common pattern of peaks can also be identified. This is due to nodes reacting similarly to abrupt variations in input data, as the appearance of a new vortices's in the observed area.

Moreover, for designing a stable system, we need to make sure that one entire cycle of execution, (that is, cycle of *DS* followed by *DA*s or the total execution time of one snapshot of Figures 5.11 and 5.19) is contained in between two sensor samples.

Figure 5.30 shows the distributions of the maximum queue size during the *DS* phase, per sampling instant (from $t = 0$ to $t = 120$). As observed earlier, using the clustering algorithm and $n_{\text{radius}} = 1$ causes overload and queuing as noted by the outliers. Also, queues



(a) Clustering



(b) Feature detection

Figure 5.30: Realtime sensing: *p1* queue sizes for $n_{\text{radius}} = 1$ to 4.

quickly drop with the increase in n_{radius}, with more contained variations and bounded behavior, as seen in Figure 5.30(a). Using the feature detection algorithm, Figure 5.30(b) shows that the maximum queues are smaller but, with comparably larger variations.

5.4 Concluding Remarks

Larger cluster sizes lead to increased accuracy but also increased traffic. But most importantly, performing in-network data compression, and feature detection and extraction leads to drastic reductions on the time required to sample the data. These results should be used to specify the most adequate systems parameters for the operation of XDense, depending on the application scenario.

We use base techniques in this evaluation and, in future work, we intend to investigate further techniques for comparative analysis. Alternative routing protocols need to be evaluated in order to decrease the number of transmissions required for a detection and, consequently, the network utilization and total latency.

Despite the potentials of reducing communication delays by introducing distributed data processing techniques, we also notice that both queues and delays can be high, with unbounded behavior. In order to enable the use of XDense in real-time applications, we need the means to allow reducing delays and queue size, and more importantly, to achieve real-time communication with bounded delays and queue sizes. An analytical framework is presented in the next chapter in order to address these issues.

Chapter 6

Analytical Model for Real-time Sensing Using Traffic Shaping

6.1 Introduction

The practicality of XDense for efficient feature detection and extraction is a necessary but not sufficient condition. We also need to provide guarantees on execution time and bounds on resource utilization. Timeliness is important for real-time applications like AFC, for which timing guarantees are essential to achieve closed-loop actuation. Providing bounds on resource utilization is also crucial to correctly dimension the network, in order to avoid overload and consequent data loss. These are factors that have great influence on hardware requirements, cost and consequently on the applicability of XDense.

In this chapter, we extend XDense with real-time capabilities. We do this by implementing traffic shapers in each node such that the overall traffic is predictable and analyzable. Further, we propose an analysis framework to accurately model the network in terms of communication delay characteristics and memory requirements. Specifically, we present the following three contributions: (i) develop a mathematical framework to model and analyze applications and network; (ii) propose heuristics to shape traffic in the network; (iii) provide upper-bounds on communication delays, application execution time, and maximum buffer requirements.

With this framework, we analyze homogeneous and heterogeneous traffic sources to compare the performance of the heuristics with a base case best-effort approach.

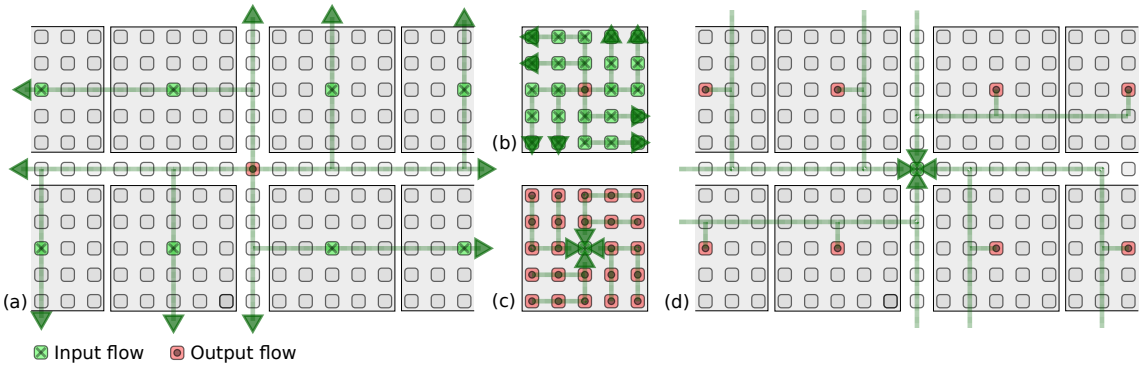


Figure 6.1: Example 45×45 network, with a single central sink. In this case, with $n_{\text{radius}} = 2$. Application phases: (a) ϕ_1 – Sink requests data from cluster-heads; (b) ϕ_2 – cluster-heads in turn send a multicast request to nodes in their cluster; (c) ϕ_3 – Nodes send sensor data back to their respective cluster-head; (d) ϕ_4 – cluster-heads process received data and send result to sink.

6.2 Application Execution

Consider the AFC use case presented earlier in the Introduction as our working example. The objective is to collect information on the nature of the airflow and identify whether it is laminar or turbulent by quantifying its characteristics along the wingspan and identify where the transition region resides.

As previously discussed, we use XDense to efficiently build a global picture of the airflow by organizing the nodes in clusters and perform local data processing. In each cluster, one node serves as the cluster-head node. It performs data aggregation within its cluster and is responsible for processing (and/or compressing) the data locally to send only meaningful information to the sink. The routing protocols elected should ideally exploit the network topology to avoid congestion. It is also required to define application protocols to allow coordination of clusters by the sink.

To tackle the challenge of analyzing and computing upper-bounds on the application execution time and the buffer requirements of the nodes through distributed processing, XDense uses three operative principles: (1) the nodes are clustered and one node in each cluster (cluster-head) is in charge of aggregating and pre-processing the data; (2) the execution of the application is divided logically in subsequent phases; (3) the network implements routing schemes which guarantee spatial isolation between the clusters.

6.2.1 Clustering Nodes

As explained earlier, the reason for grouping the nodes into clusters is to reduce the load on the network by performing in-cluster data pre-processing at the selected cluster-heads. Our tested solution implements non-overlapping "square" clusters – the network topology being a 2-D grid of $X \times Y$ nodes, all clusters are non-overlapping and of size $n_{\text{size}} \times n_{\text{size}}$, with $n_{\text{size}} \leq X$ and $n_{\text{size}} \leq Y$. n_{size} must be a positive odd number and the cluster-head is the node located at the "center" of the square. The cluster size n_{size} is defined through the system parameter n_{radius} that denotes the maximum distance from the cluster-head to the farthest node in the cluster (considering rectilinear distance, a.k.a. Manhattan distance). Figure 6.1 shows a scenario with $n_{\text{radius}} = 2$. Thus, the resulting total number of nodes in each cluster is a function of the n_{radius} given by $(2 \times n_{\text{radius}} + 1)^2$.

Nodes arbitrate their role on the network at run time (to act either as cluster-head or normal node). They do this on reception of a packet from the sink containing the packet origin and the n_{radius} parameter. Each node then calculates, based on its position in the network relative to the sink, if it is supposed to act as a cluster-head or as a normal node.

The purpose of local in-cluster processing is to extract high level aerodynamic information of the airflow, which is transmitted in a smaller number of packets (when compared to the number of packets required to transmit the raw data). However, the pre-processing and compression algorithms to be used are application-specific and are not in the scope of this chapter. We have though discussed application-specific data processing issues in Chapter 5.

6.2.2 Temporal Isolation Through Phases

Similarly to the experiments presented in Chapter 5, the execution of the application is logically divided into a set of consecutive phases. In the previous experiments three phases were used, which was enough to ensure the feature extraction functionality. In this case, because we are interested in providing real-time communication, it was convenient to separate the execution into four phases: ϕ_1, ϕ_2, ϕ_3 and ϕ_4. The first phase is started by the sink, when it requests data from the cluster-heads. Specifically, the four phases are:

- Phase ϕ_1. The sink requests the cluster-heads of all clusters to send the processed data;
- Phase ϕ_2. On receiving the request from the sink, the cluster-heads in turn request the nodes of their respective clusters to send their data;
- Phase ϕ_3. Every node of each cluster transmits its sensed data to its cluster-head;

- Phase ϕ_4. The cluster-heads process the received data and transmit the result back to the sink.

Note that ϕ_2 was not used in the previous experiments, given that the request from the sink (ϕ_1) was used to trigger ϕ_3 directly. However, because our intention is to provide temporal predictability, the addition of ϕ_2 was convenient in order to add temporal predictability to each cluster's behavior and consequently simplify the analysis.

Also note that the clusters may *not* always be in sync with respect to the phase of their execution. The second phase (ϕ_2) for instance, starts in each cluster with a different time offset; this offset being proportional to the distance between the cluster-head of each cluster and the sink. We assume cluster-heads' clocks are synchronized during ϕ_1, at the time they receive the request from the sink. The same applies to sensor node's clocks, which are synchronized during ϕ_2, at the time they receive the request from their cluster-head. That is, all nodes co-participating on a phase (in the same cluster) have a common time basis, which is an important assumption for the proposed heuristics to work. We believe this is a reasonable assumption, since nodes' synchronization point happens just before synchronism is required, providing a momentary synchronism during a given phase, even when there is considerable clock skews between nodes.

Another important assumption is that, even though sensors data may arrive at different time instants to the sink, we assume the samples are done in simultaneous by all nodes. We assume this is to ensure consistency on the acquired data, as from the same time instant; even though this has no practical effect on the results provided in this chapter.

Despite their special role, the sink and cluster-heads sense as any other node. The sink is the only node to act as the gateway with the outside world and has a backhaul link (for example, a wireless link). Figures 6.1(a) to 6.1(d) show the four phases in a chronological order.

6.2.3 Spatial Isolation Through Routing Schemes

The four phases described above require spatial isolation so that packets do not compete with each other for network resources when traversing it. We use the dimensional ordered routing protocols presented in Section 4.4.1.

Phases ϕ_1 to ϕ_3 use the Counterclockwise Dimension Routing (see Figures 6.1(a)-(c)), while in phase ϕ_4 we use the shifted clockwise dimension order algorithm (see Figure 6.1(d)).

The nodes aligned with the sink are not part of any cluster. They provide an exclusive route for packets of ϕ_4, sent by the cluster-head to the sink. This routing scheme results

in flows from phase ϕ_4 to travel orthogonal to the flows from phases ϕ_1, ϕ_2 and ϕ_3, and therefore, they do not compete for the same output port at any node on the way. This enables spatial isolation between the flows from the different phases.

6.3 Real-time Networking Model

We endow XDense with real-time capabilities by shaping the traffic at every output port of every node in the network. In simple terms, by controlling how and when packets are sent by each node, we are able to compute the maximum buffer requirements and determine the precise maximum application execution time.

The real-time application deployed on the network is characterized by a set $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ of n consecutive event-triggered phases (communication and processing primitives) that constitute the logical part of the application execution. In this work, we assume $n = 4$ (as explained in the previous section) but the approach can be extended to any arbitrary number n of phases. Every phase $\phi_i \in \Phi$, with $i \in [1, n]$, is characterized by a set $\mathscr{F}_i$ of $m_i \geq 1$ communication traffic flows exchanged between the nodes involved in phase ϕ_i. Each flow $f_{i,j} \in \mathscr{F}_i$, with $j \in [1, m_i]$, consisting of one or more packets, has an unique source node from which the communication is initiated, and may have multiple destination nodes. Formally, a flow $f_{i,j}$ is modeled as:

$$f_{i,j} = \{O_{i,j}\ ,\ \sigma_{i,j}\ ,\ \beta_{i,j}\} \tag{6.1}$$

The offset $O_{i,j}$ is a constant delay before the sending of the first packet of flow $f_{i,j}$. The message size $\sigma_{i,j}$ is the number of packets that are sent in each flow $f_{i,j}$ and the burstiness $\beta_{i,j} \in [0, 1]$ represents the rate at which those packets are released. A burstiness of 0 means that no packets are transmitted, and a burstiness of $x \in]0, 1]$ means that a packet is transmitted every $\frac{1}{x}$ TTS. These three parameters together describe a finite constant-rate flow with an initial offset. The flow parameters σ and β were conceived to couple the application sampling requirements with the communication model, in the sense that they allow modeling application scenarios with different data sampling requirements. A few example flows are illustrated below.

Example 1 (9 Degrees of Freedom (DOF) motion sensor) *Consider a 9 DOF motion sensor whose data has to be transmitted as nine separate packets in a single flow (one packet for each degree of freedom). In this case, we want the data to be transmitted together. Therefore, we set $\beta = 1$ with $\sigma = 9$ for that flow.*

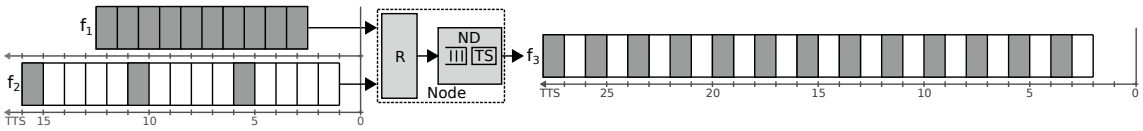


Figure 6.2: Traffic shaper example scenario: two input flows shaped by an intermediate node as an output flow. Parameters for the input flows are $f_1 = \{O = 2.5, \beta = 1, \sigma = 10\}$ and $f_2 = \{O = 1, \beta = \frac{1}{5}, \sigma = 3\}$. The resulting flow is $f_3 = \{O = 2, \beta = \frac{1}{2}, \sigma = 13\}$.

Example 2 (Pressure sampling) *Consider a use-case in which ten samples of pressure data need to be transmitted, using one packet per sample. We are interested in having periodic sampling, equally distributed in time. By setting the burstiness to* $\frac{1}{5}$ *for instance, one packet will be sent every 5* TTS. *Therefore, for that flow we set* $\beta = \frac{1}{5}$ *and* $\sigma = 10$.

6.3.1 Shaping Flows and Traffic Throughout the Network

As discussed above, the sending of all packets by the source node of the corresponding flow f is done according to its parameters (O, σ, β); these three parameters allow for a precise timing and sending rate at the source node of f. Note that for simplicity, we shall use hereafter the symbol f to denote a flow. We will mention the indexes i and j that indicate the phase and flow indexes respectively only if necessary.

Although the flows are shaped at their source, when multiple flows (say, $f_1^{\text{in}}, f_2^{\text{in}}, \ldots, f_k^{\text{in}}$) traverse the network at the same time, pass through the same router, and compete for the same output port, the resulting output flow f^{out} at that port is a superposition of all these competing flows. As such, f^{out} may present an irregular packet transmission pattern and a rate that can no longer be modeled using the three parameters (O, σ, β).

For example, let us look at Figure 6.2, which illustrates two input flows f_1^{in} *and* f_2^{in} competing for a same output port of a node. Each of these flows f_k^{in} starts at time O_k and has a duration defined as $\ell_k = \frac{\sigma_k}{\beta_k}$. That is, flow f_k sends all its packets after ℓ_k TTS, at $t = O_k + \ell_k$. In this example, thanks to the traffic shaper *TS*, the interference of these two input flows lead to an output flow f_3^{out} that is not a superposition of the two input flows, but rather it present deterministic patterns that can be modeled using the three parameters (O, σ, β).

That is, to make the network amenable to timing analysis, we shape the traffic *at every output port* of every node and make it fit the linear model (O, σ, β). For that, we first identify the set of input flows f_k^{in} (with $k = 1, 2, \ldots$) at every output port of every node in the network, and based on the respective parameters (O_k, σ_k, β_k) of these flows, we compute the parameters $(O^{\text{out}}, \sigma^{\text{out}}, \beta^{\text{out}})$ that are used to shape the resulting output flow at that output port.

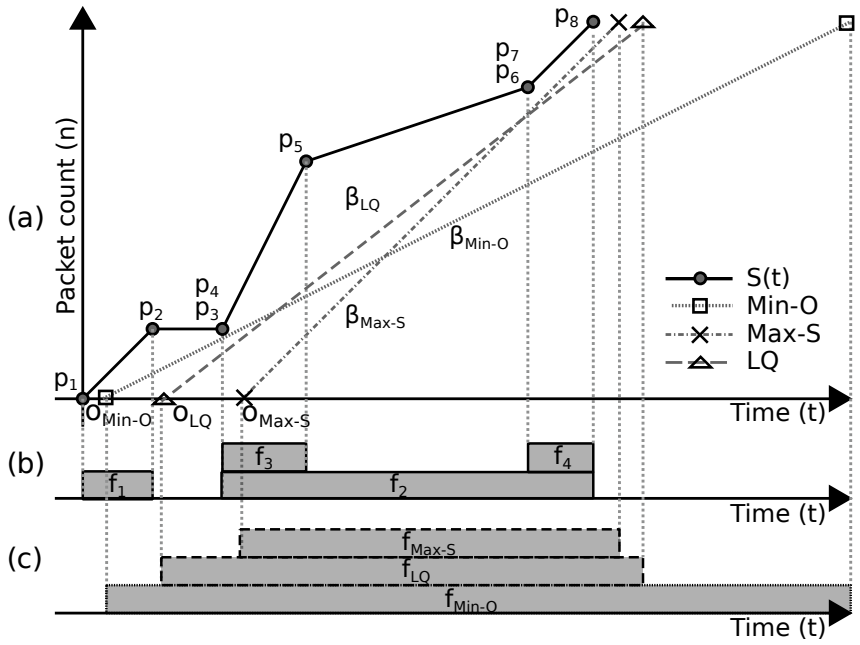


Figure 6.3: Traffic shaping heuristics: (a) input, and output flows using the proposed heuristics; time-line showing offset and duration of (b) arriving flows and (c) departure flows.

In Figure 6.3, we present a more detailed example to illustrate how the traffic shaping is done. Figure 6.3(a) shows packet arrivals curve $S(t)$ due to four input flows $f_1^{\text{in}}, f_2^{\text{in}}, f_3^{\text{in}} and f_4^{\text{in}}$ (see Figure 6.3(b)). The arrival curve corresponds to the input flows that define the number of packets to be sent over time from the output port, that depends on the starting time and duration of all the competing input flows. Three possible resulting flows f^{out} are computed and shown in Figure 6.3(c), each with its corresponding departure curves in Figure 6.3(a).

The computation of $(O^{\text{out}}, \sigma^{\text{out}}, \beta^{\text{out}})$ is therefore performed at every output port of every node in the network interactively, starting at the source node of every flow and iterating, one port at the time, throughout the network until a shaper is defined for all the output ports.[1] We make two important assumptions regarding the flows and their routing.

Assumption 1. During phases ϕ_3 and ϕ_4, in every node, all the packets entering by a given input port are assumed to exit through a single output port. In other words, a single input packet does not produce more than one output packet.

Assumption 2. There are no circular dependencies between the flows. For any output port, say p_1, the computation of the parameters of its traffic shaper requires each of its competing input flows to be modeled already by the three parameters (O, σ, β). If any of

[1] Note that it has been proven in [118] that to calculate optimal shaping parameters in a multihop scenario can be computationally intractable, and thus finding an optimal solution at runtime is not feasible.

these input flows, say f_k^{in}, comes from the output port (say p_2) of an upstream router, it is required that the parameters $(O_k^{\text{in}}, \sigma_k^{\text{in}}, \beta_k^{\text{in}})$ of the shaper of that upstream output port p_2 have been computed already. Similarly, this requirement must be satisfied for all the input flows competing for p_2, and interactively it must be satisfied as well for all the output ports of the upstream routers till the traffic shaper at the source nodes of all the interfering flows. Therefore, computing traffic shaping parameters is an iterative process that must be executed until $(O^{\text{out}}, \sigma^{\text{out}}, \beta^{\text{out}})$ is calculated for all nodes. In simple terms, there cannot be a flow f_1 competing for an output port with a flow f_2 that competes for an output port with a flow f_3, and so on until reaching a flow f_k that competes for an output port with f_1.

Assuming no cyclic dependencies between the flows, the parameters $(O^{\text{out}}, \sigma^{\text{out}}, \beta^{\text{out}})$ of every traffic shaper may be computed in many different ways for a same set of interfering input flows. In the next section, we propose three different methods of computation.

6.3.2 Shaping Traffic at a Single Output Port

We propose three heuristics to compute the parameters $(O^{\text{out}}, \sigma^{\text{out}}, \beta^{\text{out}})$ of the shaper used at a given output port. Let F^{in} denote the set of input flows that compete for the output port under analysis. Every $f_k^{\text{in}} \in F^{\text{in}}$ is characterized by the three parameters $(O_k^{\text{in}}, \sigma_k^{\text{in}}, \beta_k^{\text{in}})$. For each $f_k^{\text{in}} \in F^{\text{in}}$, we define the function $S_k^{\text{in}}(t)$ as:

$$S_k^{\text{in}}(t) = \begin{cases} 0 & t \leq O_k^{\text{in}} \\ \beta_k^{\text{in}} \times (t - O_k^{\text{in}}) & O_k^{\text{in}} < t < O_k^{\text{in}} + \ell_k \\ \sigma_k^{\text{in}} & t \geq O_k^{\text{in}} + \ell_k \end{cases} \tag{6.2}$$

Broadly speaking, every function $S_k^{\text{in}}(t)$ represents the number of packets sent by the flow f_k^{in} at a given time t (TTS). When t is earlier than the starting instant O_k^{in} of the flow, the function returns 0 since the flow has not sent a packet yet; For t larger than the finishing time of the flow $(O_k^{\text{in}} + \ell_k)$, the function returns the total number σ_k^{in} of packets sent by f_k^{in}, with ℓ_k being the duration of the flow; Between the two bounds O_k^{in} and $O_k^{\text{in}} + \ell_k$, the function increases steadily from 0 to σ_k^{in} with a constant slope of β_k^{in}.

Let $S(t) = \sum_{f_k^{\text{in}} \in F^{\text{in}}} S_k^{\text{in}}(t)$ be the sum of the functions $S_k^{\text{in}}(t)$ of all the input flows f_k^{in}. This function $S(t)$ is depicted in Figure 6.3(a). Informally, $S(t)$ gives the number of packets that arrive at the considered input port in a time window of length t (TTS). We further denote by $\mathscr{T} = \{t_1, t_2, \ldots, t_m\}$ the finite set of time-instants (sorted in chronological order) corresponding to the discontinuity points of the function $S(t)$. These discontinuity points are denoted as $p_1, p_2, \ldots, p_m$ in Figure 6.3. With these new notations, we can introduce our three heuristics for the computation of the parameters $(O^{\text{out}}, \sigma^{\text{out}}, \beta^{\text{out}})$ of the shaper used at the analyzed output port.

For a given shaper $(O^{\text{out}}, \sigma^{\text{out}}, \beta^{\text{out}})$ represented by a straight line L^{out} of slope β^{out} and passing through the point $(O^{\text{out}}, 0)$, the *vertical* distance dv_j^{out} between a point $(t_j, S(t_j))$ $\in S(t)$, $\forall t_j \in \mathscr{T}$, and the line L^{out} represents the number of packets being buffered at time t at that output port. The *horizontal* distance dh_j^{out} between a point $(t_j, S(t_j)) \in S(t)$, $\forall t_j \in \mathscr{T}$ and L^{out} represents the delay (induced by the shaper) that all the packets that have arrived at that output port at time t_j will incur because of the shaper.

We start by computing the output flow size σ^{out} that is the same for all the heuristics proposed. Since the shaper is not allowed to drop any packet, it is naturally the sum of the size of all the input flows f_k^{in}, i.e.

$$\sigma^{\text{out}} = \sum_{f_k^{\text{in}} \in F^{\text{in}}} \sigma_k^{\text{in}}$$

In the remainder of this section we discuss the intuition behind each heuristic and explain how they derive the two other flow parameters, O^{out} and β^{out}.

Minimum offset (Min-O)**.** This first heuristic aims at avoiding bursty traffic while coping as much as possible with the bandwidth demand of the input flows. This traffic shaper forwards the first packet as soon as it can, i.e. one TTS after the packet has arrived, at time $O^{\text{out}} = t_1 + 1$ TTS, and forwards all the subsequent packets at the highest admissible rate; that is, with the highest burstiness β^{out} such that the number of packets sent at any time $t \geq O^{\text{out}}$ never exceeds $S(t)$. This burstiness corresponds to the highest slope among the slopes of all the lines passing through the point $(t_1 + 1, 0)$ such that, for every $t_j \in \mathscr{T}$, the point of x-coordinate t_j in the line has an y-coordinate $\leq S(t)$ – In simple terms, the line is "below" the function $S(t)$, $\forall t \geq 0$. This slope is simply given by

$$\beta^{\text{out}} = \left[\min_{t_j \in \mathscr{T}} \left(\frac{S(t_j)}{t_j - (t_1 + 1)} \right) \right]_0^1$$

where $[x]_y^z = \max(\min(x, z), y)$. Note that by definition of t_1, we have $t_1 = \min_{f_k^{\text{in}} \in F^{\text{in}}}(O_k^{\text{in}})$. Figure 6.3(a) shows Min-O departure curve with β^{out} as $\beta_{\text{Min-O}}$.

Maximum slope (Max-S)**.** The second heuristic aims at *not* consuming any bandwidth for as much time as possible and then send all the packets in a burst. Similarly to the Min-O heuristic, the Max-S approach selects one "anchor" point of $S(t)$ and computes the *maximum* slope β^{out} such that the line with slope β^{out} passing through the selected

point is "below" the function $S(t)$. In Min-O, we selected the anchor point $(t_1+1,0)$ whereas Max-S selects the point $(t_m,S(t_m))$. The maximum admissible slope is such that the line remaining below $S(t)$ is given by:

$$\beta^{\text{out}} = \max_{t_j \in \mathscr{T}} \left(\frac{S(t_m) - S(t_j)}{t_m - t_j} \right) \tag{6.3}$$

Figure 6.3(a) shows Max-S departure curve with β^{out} as $\beta_{\text{Max-S}}$. The offset O^{out} in Max-S is simply set to the X-intercept of the line of slope β^{out} and passing through the anchor point $(t_m,S(t_m))$ to which we add 1 TTS, to make sure that packets are not forwarded before the first packet arrives (like we did in Min-O), for example:

$$O^{\text{out}} = t_m - \frac{S(t_m)}{\beta^{\text{out}}} + 1$$

After computing the offset O^{out}, it is now safe to readjust the slope as $\beta^{\text{out}} = [\beta^{\text{out}}]_0^1$ to model the fact that the shaper cannot forward a negative number of packets and neither it can forward more than one packet at a time. Note that this re-adjustment must be performed after computing O^{out} as doing it before would in some cases allow a packet to be forwarded before it has even arrived, that is, the line would not be completely below the function $S(t)$.

Figure 6.3(a) shows the departure line of Max-S, initially calculated with a slope > 1 as a result of Equation 6.3. That slope is then adjusted to $\beta^{\text{out}} = 1$ as depicted on that figure. As seen, after adjusting its slope, the line corresponding to the parameters of the Max-S traffic shaper does not intersect with the function $S(t)$ – It seems to be "too much shifted to the right". An easy patch to reduce this gap between $S(t)$ and the shaper is to set its offset to the *minimum* offset such that the line remains below all the points of $S(t)$. That is,

$$O^{\text{out}} = \min_{t \geq 0} \left\{ t \text{ such that } \beta^{\text{out}} \leq \min_{\substack{t_j \in \mathscr{T} \\ t_j > t}} \left(\frac{S(t) - S(t_j)}{t - t_j} \right) \right\} \tag{6.4}$$

Note that this value of O^{out} can be computed easily by positioning the line of slope β^{out} on every point $(t_j,S(t_j))$, $\forall t_j \in \mathscr{T}$, and retaining the maximum X-intercept of all these lines.

Least-square regression (LQ) The intuition behind this third heuristic is to minimize both the queue size and the delay by finding the line L^{out} that minimizes the distance

between every point $(t_j, S(t_j)) \in S(t)$, $\forall t_j \in \mathscr{T}$ and L^{out}. This line is commonly known as the *regression line* of the points $(t_j, S(t_j)) \in S(t)$. Using the least-squares method, which is the most common method for fitting a regression line, the slope of that line is given by:

$$\beta^{\text{out}} = r \times \frac{\sqrt{\frac{1}{m}\sum_{t_j \in \mathscr{T}} (S(t_j) - \bar{S})^2}}{\sqrt{\frac{1}{m}\sum_{t_j \in \mathscr{T}} (t_j - \bar{t})^2}} \tag{6.5}$$

where

$$\begin{aligned} \bar{t} &= \frac{1}{m}\sum_{t_j \in \mathscr{T}} t_j \\ \bar{S} &= \frac{1}{m}\sum_{t_j \in \mathscr{T}} S(t_j) \end{aligned}$$

and r is the correlation coefficient computed as:

$$r = \frac{\sum_{t_j \in \mathscr{T}} (t_j - \bar{t})(S(t_j) - \bar{S})}{\sqrt{\sum_{t_j \in \mathscr{T}} (t_j - \bar{t})^2 \sum_{t_j \in \mathscr{T}} (S(t_j) - \bar{S})^2}}$$

Once we have computed the slope, we choose the smallest offset O^{out} such that the line of slope β^{out} and passing through $(O^{\text{out}}, 0)$ is never above any point $(t, S(t))$, $\forall t \geq 0$. This is done using Equation 6.4.

6.3.3 Worst-case Per-hop Delays and Maximum Queue Sizes

Having stated the heuristics, we can now apply them to all the phases of the application. We perform this in a hop-by-hop strategy, starting from the output ports of the nodes for which the parameters $(O_k^{\text{in}}, \sigma_k^{\text{in}}, \beta_k^{\text{in}})$ of all the interfering flows f_k^{in} are known. For each such output port, the resulting output flow f^{out} is shaped using the same model $(O^{\text{out}}, \sigma^{\text{out}}, \beta^{\text{out}})$ that is then propagated as the input flow in the next hop. The process continues until the parameters of the shaper of every output port of all the nodes of the network are defined (the output ports that no flows ever traverse and that are thus unused are naturally ignored). As mentioned earlier, we assume that there are no cyclic dependencies between the flows at any output port, which implies that the process eventually terminates.

After that step, we can now compute at each output port the maximum transmission delay caused by its traffic shaper $(O^{\text{out}}, \sigma^{\text{out}}, \beta^{\text{out}})$, as well as its maximum queue size. To ease the explanation, we shall use the same visual representation as that used in the previous section for the shaper and the function $S(t)$. The shaper is represented by a straight line of slope β^{out} that intersects with the x-axis at the point $(O^{\text{out}}, 0)$. We denote this line L^{out} and write its equation as:

$$L^{\text{out}}(t) = \beta^{\text{out}} t - \beta^{\text{out}} O^{\text{out}} \tag{6.6}$$

We define $S(t)$ as in the previous section and keep the notations $\mathscr{T} = \{t_1, t_2, \ldots, t_m\}$ to express the finite set of time-instants (sorted in chronological order) corresponding to the discontinuity points of the function $S(t)$.

As explained previously, the number of packets buffered at the output port at any time-instant t is given by the *vertical* distance dv_j^{out} between the point $(t, S(t))$ and the point $(t, L^{\text{out}}(t))$ on the line L^{out}. This vertical distance is simply equal to:

$$\text{dv}_j^{\text{out}} = S(t) - L^{\text{out}}(t)$$

and thus the maximum number MaxQueue of packets buffered at that output port is given by:

$$\text{MaxQueue} = \max_{t \geq 0} \left(S(t) - L^{\text{out}}(t) \right)$$

Since $S(t)$ is a continuous piecewise function for which every sub-function is linear, it can easily be show that the maximum of the previous equation can be found by looking only at the time-instants $t_j \in \mathscr{T}$ rather than at all $t \geq 0$, i.e.:

$$\text{MaxQueue} = \max_{t_j \in \mathscr{T}} \left(S(t_j) - L^{\text{out}}(t_j) \right) \tag{6.7}$$

This holds true because every sub-function of $S(t)$ is a segment that is either:

- parallel to L^{out}. In this case, all the points on that segment are at the same distance from L^{out}, including its two extremities that are discontinuity points with an x-coordinate included in $\mathscr{T}$.

- converging towards L^{out}. In this case, the *leftmost* point on the segment (whose x-coordinate is an instant $t_j \in \mathscr{T}$) is the furthest to L^{out}.

- diverging from L^{out}. In this case, the *rightmost* point on the segment (whose x-coordinate is an instant $t_j \in \mathscr{T}$) is the furthest to L^{out}.

Similarly, the transmission delay at any time-instant t is given by the *horizontal* distance $\mathrm{dh}_j^{\mathrm{out}}$ between the point $(t, S(t))$ and the point of y-coordinate $S(t)$ on the line L^{out}. According to Equation 6.6, that point of y-coordinate $S(t) \in L^{\mathrm{out}}$ has an x-coordinate x such that $S(t) = \beta^{\mathrm{out}} x - \beta^{\mathrm{out}} O^{\mathrm{out}}$ and thus $x = \frac{S(t)}{\beta^{\mathrm{out}}} + O^{\mathrm{out}}$. The horizontal distance is then simply given by:

$$\mathrm{dh}_j^{\mathrm{out}} = \frac{S(t)}{\beta^{\mathrm{out}}} + O^{\mathrm{out}} - t$$

and thus the maximum delay (MaxDelay) at that output port is:

$$\mathrm{MaxDelay} = \max_{t \geq 0} \left(\frac{S(t)}{\beta^{\mathrm{out}}} + O^{\mathrm{out}} - t \right)$$

For the same reasons as those mentioned for MaxQueue, the maximum delay (MaxDelay) can be computed by looking only at the points $t_j \in \mathscr{T}$, i.e.,

$$\mathrm{MaxDelay} = \max_{t_j \in \mathscr{T}} \left(\frac{S(t_j)}{\beta^{\mathrm{out}}} + O^{\mathrm{out}} - t_j \right) \tag{6.8}$$

Note that the transmission delay is an interesting parameter to analyze the end-to-end delay or per-hop delays of individual packets. However, in this work we rather focus on estimating upper-bounds on the execution time of the phases and thus of the overall real-time application.

To compute the execution time of a given phase, we must know exactly when the phase starts and when it ends. However, phases may overlap in time and happen simultaneously. For instance, for the application scenario considered in this paper, a cluster-head located close to the sink may enter phase ϕ_2 long before a cluster-head that is far from the sink (since it receives the request from phase ϕ_1 sooner). For simplicity, we assume in this work that a phase ends when a given node has received all the packets sent to it. For example, the time at which all the cluster-heads have received their requested data marks the end of phase ϕ_3 and the time at which the sink has received all the processed data marks the end of phase ϕ_4. As such, we compute the execution time of a phase as the relative time-instant at which all the four input flows of that given node – a cluster-head for phase ϕ_3 and the sink for phase ϕ_4 – terminate, i.e. the four flows coming from the north, south, east, and west input ports of that node. the execution time of a phase is thus given by:

$$\mathrm{ExecTime} = \max_{\mathrm{card} \in [\uparrow, \downarrow, \rightarrow, \leftarrow]} \left(O^{\mathrm{in}} + \frac{\sigma^{\mathrm{in}}}{\beta^{\mathrm{in}}} \right) \tag{6.9}$$

where for each cardinal direction $\uparrow$, $\downarrow$, $\rightarrow$, and $\leftarrow$ (north, south, east, and west), the flow f^{in} characterized by $(O^{\mathrm{in}}, \sigma^{\mathrm{in}}, \beta^{\mathrm{in}})$ is the input flow coming from that cardinal direction.

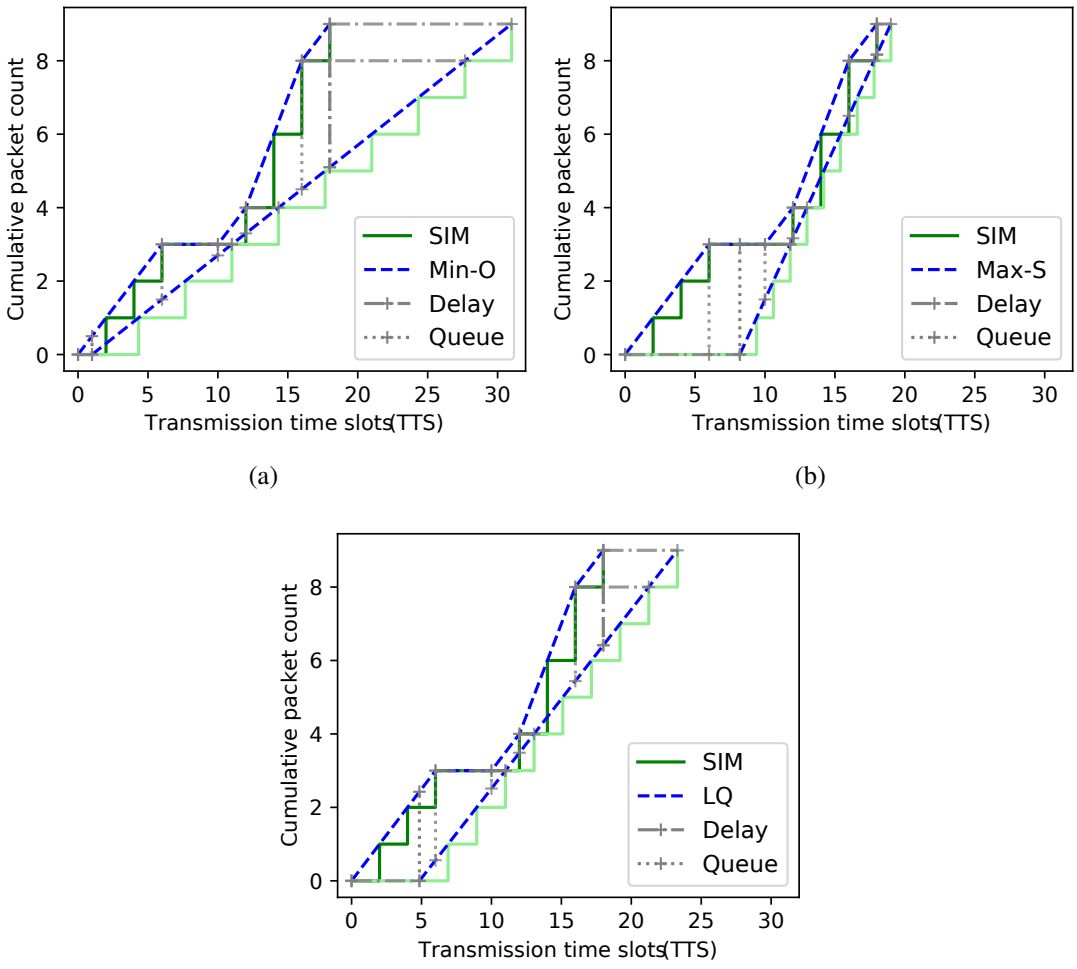


Figure 6.4: Cumulative arrival/departure curves for a single node, using (a) Min-O, (b) Max-S and (c) LQ heuristics.

6.4 Validation Example

To validate our theoretical model we define an example scenario and compare it with simulation. We define the following scenario: a single node receives an arbitrary number of known input flows, which are shaped into an output flow (using each of the proposed heuristics). We compare the arrival/departure curves calculated using our model, against the curves obtained in simulation.

Figure 6.4(a-c) shows the cumulative arrival/departure curves due to three input flows, and the resulting output flow obtained using each of the three heuristics proposed. In this case, the input flows characteristics were chosen in order to emphasize the effect of each shaping heuristic on the output flow.

We use the same input flows in all three scenarios, which are $F_x = [f_1^{in}, f_2^{in}, f_3^{in}]$, where

$f_1^{in} = \{O = 0, \sigma = 3, \beta = 0.5\}$, $f_2^{in} = \{O = 10, \sigma = 3, \beta = 0.5\}$ and $f_3^{in} = \{O = 12, \sigma = 3, \beta = 0.5\}$. The resulting output flow is different for each heuristic. These are $f_{Min-O}^{out} = \{O = 1, \sigma = 9, \beta = 0.3\}$, $f_{Max-S}^{out} = \{O = 8.2, \sigma = 9, \beta = 0.83\}$ and $f_{LQ}^{out} = \{O = 4.8, \sigma = 9, \beta = 0.49\}$.

The arrival curves obtained through simulation (common to the three cases) is a stair function, resulting from the superposition of all the arriving flows. Because the output flow is shaped, the corresponding departure curve is a homogeneous stair function. As expected, the arrival/departure curves calculated using our model precede the simulated ones in every point. It also shows the queue size and delay calculated at every point in which the arrival and/or departure curves start, finish or change its slope.

For the given F_x, from Figure 6.4(a), Min-O performs badly as compared to the other heuristics, as it adds large delays between the arrivals and departures, which leads to equally large queues and long execution time. We notice from Figure 6.4(b) that the departure curve obtained with Max-S approaches maximally the arrival curve at its tip (1 TTS far), leading to the optimal execution time with the cost of larger queues from 0 to 10 TTS. On the other hand, the LQ heuristic leads to smaller queues, with a slightly longer execution time.

By running this experiment extensively, using random input flows, we are able to verify if our model definitions are consistent, besides being able to gain the intuition and qualify the effect of each heuristic.

Even though these results provide an intuition on the trade-offs between the heuristics proposed, they do not depict the results of multi-hop communication, in which case the effects may differ. Therefore, a more complete evaluation is provided in the next section to understand how each heuristic performs through multiple hops.

6.5 Evaluation of Traffic Shaping Heuristics

Application use-case: To evaluate the proposed heuristics, we consider the application scenario briefly described in Introduction. Remember that the execution of this application is divided logically into four consecutive phases ϕ_1, ϕ_2, ϕ_3 and ϕ_4. In the first phase ϕ_1, the unique sink node requests all the cluster-heads to send their data; in phase ϕ_2, the cluster-heads perform another request to all the nodes of their respective cluster; in phase ϕ_3, the nodes reply to the cluster-heads by sending them the sensed data; and in phase ϕ_4, the cluster-heads process the data received and transmit the result back to the sink. Since there is no network congestion in phases ϕ_1 and ϕ_2 – because all the packets sent from the sink to the cluster-heads and then from the cluster-heads to the sensing nodes have their

own private route to their destination – these two phases are neither affected by a modification of the cluster size, nor by changing the number of clusters, nor by altering the burstiness of the flows generated during phases ϕ_3 and ϕ_4. We shall therefore focus *only* on phases ϕ_3 and ϕ_4 in which network congestion does occur and for which a modification of the aforementioned parameters has an impact on the performance.

Network setup: The network is organized as a square grid of $45 \times 45 = 2025$ nodes with an unique sink located at the center of the grid. Figure 6.1 depicts a closeup on the sink. In that figure we can also see the overall cluster organization, the routes taken by the flows in the different phases and the central row and central column of nodes in the middle that are dedicated only to the communication between the cluster-heads and the sink. Based on an integer parameter n_{radius} that we vary in our experiments, we define every cluster as a square grid of $(2n_{\text{radius}} + 1)^2$ nodes with the cluster-head at the center of the grid. As such, n_{radius} defines both the cluster size and the number of clusters (the smaller the clusters, the more clusters in the network, and reversely). Because of our routing algorithms and network symmetry assumptions (position of sink in the center of the network and cluster-head in the center of their cluster), the workload observed in each quadrant around the sink or cluster-heads will be identical. This makes it sufficient to analyze a single quadrant of the network or cluster.

Shaping heuristics: We evaluate the performance of the three proposed heuristics Min-O, Max-S, and LQ against the performance of a cycle-accurate network simulator that we call BE. The simulator does not implement any traffic shaper and thus it delivers a best-effort (BE) performance overall.

Evaluation criteria and methodology: For each of the three heuristics Min-O, Max-S, LQ, we evaluate the maximum queue sizes and the end-to-end execution time of the phases ϕ_3 and ϕ_4. For BE, maximum queue sizes and phases execution times are measured in the simulator. We do so for different cluster sizes, flow burstiness and network load distributions. Because there is no source of non-determinism in our simulation model, each runs gives the exact same results for the same input parameters. Thus, we are only required to run our experiments once for each scenario, for as long as all four phases last.

To understand the impact of varying the network load, we analyze both homogeneous and heterogeneous flow scenarios in phases ϕ_3 and ϕ_4 (that is, the phases when the actual data transmission happens). We define a homogeneous flow scenario as one in which all nodes generate flows with equal burstiness and message size. A scenario with random message sizes and burstiness is defined as a heterogeneous flow scenario.

We analyze the homogeneous scenario by varying the burstiness β of flows from phases ϕ_3 and ϕ_4 from 0.02 to 1 by step of 0.02, for different cluster sizes.

The message size σ differs for each phase. For phases ϕ_1 and ϕ_2, a single packet is generated at the sink and cluster-head ($\sigma = 1$). In phase ϕ_3, each node outputs a flow with message size $\sigma = 4$. At the end of ϕ_3, the cluster-head receives in total four packets per each node on its cluster. Subsequently, each cluster-head outputs a flow with message size σ, as the sum of all these packets plus 4 packets of its own sensed data, times $\lceil 1 - \mathrm{CR} \rceil$. The term CR aims at reproducing the effect of data compression by the cluster-head. In this work we set CR = 80%, which was shown in Section 5.3 to be a reasonable ratio in relevant air flow scenarios. The number of packets originated by each cluster vary with the cluster size, whereas the number of clusters is inversely proportional to the cluster size. This trade-off has a compensatory effect on the overall number of packets transmitted to the sink.

In the heterogeneous flow scenario, flows generated at phases ϕ_3 and ϕ_4 have random message sizes. We use a uniform distribution function with $\sigma = rand(0, 10)$ and burstiness $\beta = rand(0.02, 1)$. A message size of zero means that a node does not have an output flow.

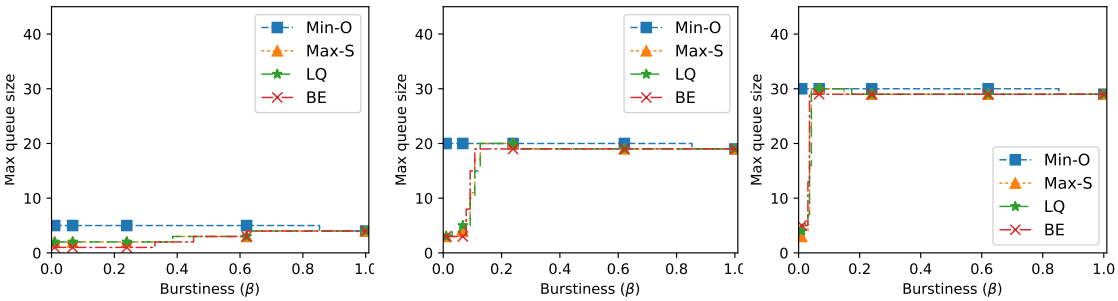
In our analysis, we compare the performance of both heterogeneous and homogeneous scenarios. In order to do this fairly, we guarantee that for both homogeneous (HO) and heterogeneous (HE) network load distributions, the sum of burstiness of all flows, as well as the sum of all message sizes, are equal. That is, $\sum \beta^{HO} = \sum \beta^{HE}$ and $\sum \sigma^{HO} = \sum \sigma^{HE}$. This guarantees that the total network load remains the same for both scenarios, even if individual load distributions vary.

For both scenarios, the offset remains the same; equal to their distance from the sink/cluster-head (since it is meant to model the minimum time required for a node to reply to a request).

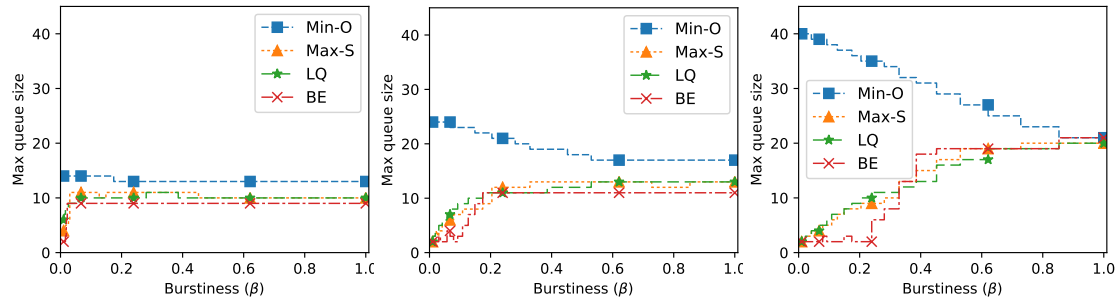
6.5.1 Maximum queue size with homogeneous load distribution

For each of the three heuristics Min-O, Max-S and LQ, we first derive the parameters (O, σ, β) of all the traffic shapers in the network. Then we use Equation 6.7 on every shaper to compute the maximum queue size of the corresponding node and finally, we retain the maximum queue size of all the nodes in the network.

As we can see in Figures 6.5(a) and 6.5(b), in phase ϕ_3 the queues are smaller for smaller clusters (n_{radius}). This is expected since smaller clusters contain fewer nodes and therefore there are less packets exchanged within each cluster, and thus less congestion. The opposite scenario would be expected for phase ϕ_4, since using smaller clusters means



(a) $\phi_3, n_{\text{radius}} = 1$ (b) $\phi_3, n_{\text{radius}} = 3$ (c) $\phi_3, n_{\text{radius}} = 5$

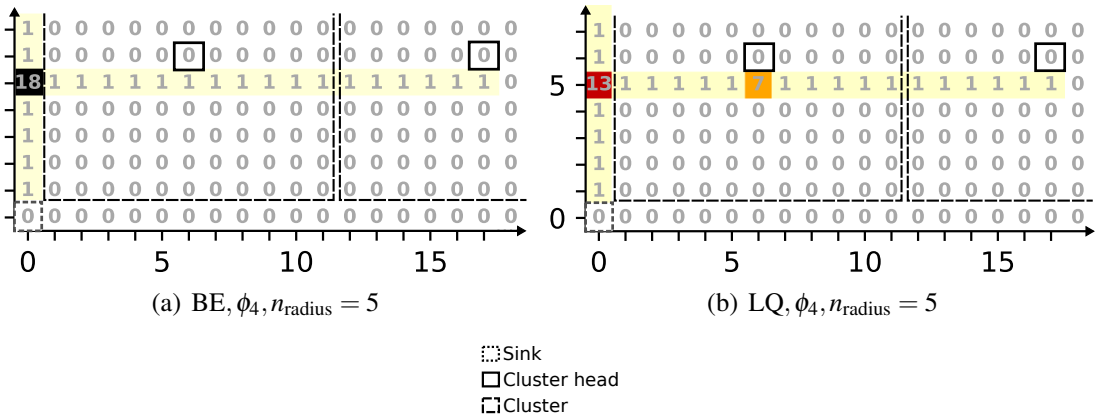


(d) $\phi_4, n_{\text{radius}} = 1$ (e) $\phi_4, n_{\text{radius}} = 3$ (f) $\phi_4, n_{\text{radius}} = 5$

Figure 6.5: Homogeneous flow scenario: Maximum queue size for traffic shaping heuristics against simulation. Results are for phases ϕ_3 and ϕ_4 and n_{radius} set to 1, 3 and 5.

more clusters in the network, and thus more cluster-heads transmitting packets to the sink. Yet, this is not observed in Figure 6.5(d) and 6.5(e). That is, smaller clusters *do not imply larger queues*. The reason for this counter-intuitive result can be unveiled by looking at the *utilization* of the four input links of the sink.

We define the link utilization as the average utilization of a given link of a node during a given phase. It is calculated as the number of packets sent on that link in a given phase (here, phase ϕ_4) divided by the time (number of TTS) it takes for all those packets to traverse it. An utilization of 1 means that the link is never idle during the considered phase whereas an utilization of 0 means that the link is not used. As seen in Figure 6.7(d), smaller clusters yield a better utilization of the input links of the sink. This is because the sum of packets sent to the sink does not depend *only* on the cluster size. With more (and smaller) clusters, there will be more clusters and more cluster-heads transmitting to the sink from shorter distances. Thus, their input links will spend less time idle waiting for the packets to arrive from longer distances. In other words, with fewer (but bigger) clusters, cluster-heads are farther from the sink and thus their input links spend more time idle waiting for the packets to traverse the intermediate hops. Greater utilization, for the same number of packets received by the sink, shows that there is less congestion in the



(a) BE, $\phi_4, n_{\text{radius}} = 5$ (b) LQ, $\phi_4, n_{\text{radius}} = 5$

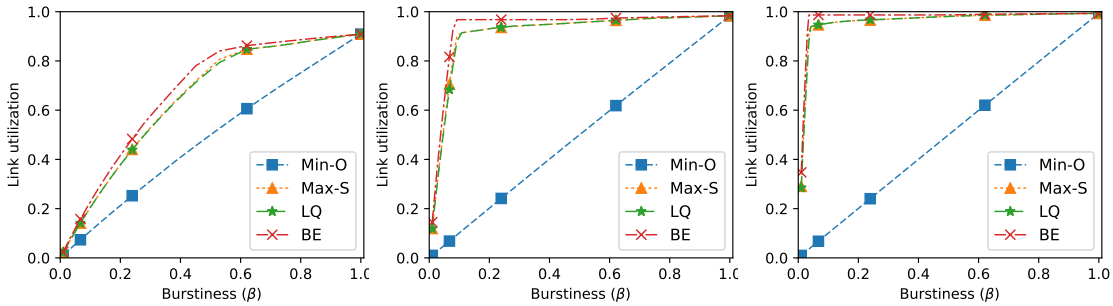
Figure 6.6: Queue size density map of the top-right quadrant of the network (17×7 nodes), for heuristics (a) LQ and (b) BE. X and Y axis are nodes coordinates relative to the sink.

network and thus smaller queue at individual hops.

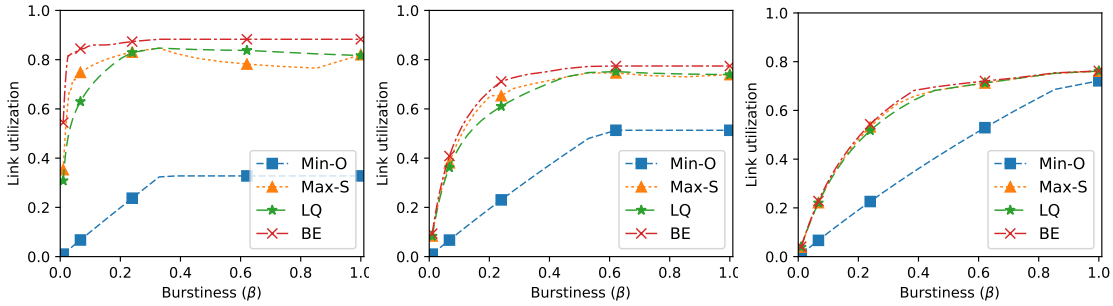
It is worth noticing that in some scenarios, the maximum queue size obtained when using traffic shaping is smaller than the maximum queue size without traffic shaping. This is experienced for example in ϕ_4 for $n_{radius} = 3$ and 5, shown in Figures 6.5(e) and 6.5(f), for $\beta \in [0.4, 0.6]$. In this window, the maximum queue size of Max-S and LQ are smaller than that of BE. This result is due to the offset O imposed by the traffic shapers in the initial hops. In these cases, the offsets act on distributing in time the load on the network, and thus decreasing the maximum congestion.

However, in cases with lower link utilization and burstiness, BE yields shorter queue sizes. In these cases the network is underutilized, such that BE still does not lead to excessive load on the network. On the other hand, performing traffic shaping imputes on unnecessary buffering by nodes, and consequently greater queues.

The aforementioned effect is shown in Figure 6.6. It shows the queue size density map of the top-right quadrant of the network, with $n_{radius} = 5$. The sink is located at the bottom-left corner. Flows are routed using shifted clockwise routing as previously shown in Figure 6.1; hence, right to left in this map. The left-most nodes in the network are usually where the bottleneck happens. Using BE for example, in Figure 6.6(a), the node located at coordinates $(x, y) = (0, 5)$ gets up to 18 packets queued, since it is located at a conjunction of flows coming from cluster-heads on its right and top. Because of the offset O calculated using LQ, we can see (Figure 6.6(b)) that by shaping and queuing the flows originating at the right side of the network (by the node located at $(6, 5)$) has the effect of delaying the reception of those packets by the left most node located at $(0, 5)$; thus, reducing the maximum queue size at the nodes aligned with the sink. Comparatively, the



(a) $\phi_3, n_{\text{radius}} = 1$ (b) $\phi_3, n_{\text{radius}} = 3$ (c) $\phi_3, n_{\text{radius}} = 5$



(d) $\phi_4, n_{\text{radius}} = 1$ (e) $\phi_4, n_{\text{radius}} = 3$ (f) $\phi_4, n_{\text{radius}} = 5$

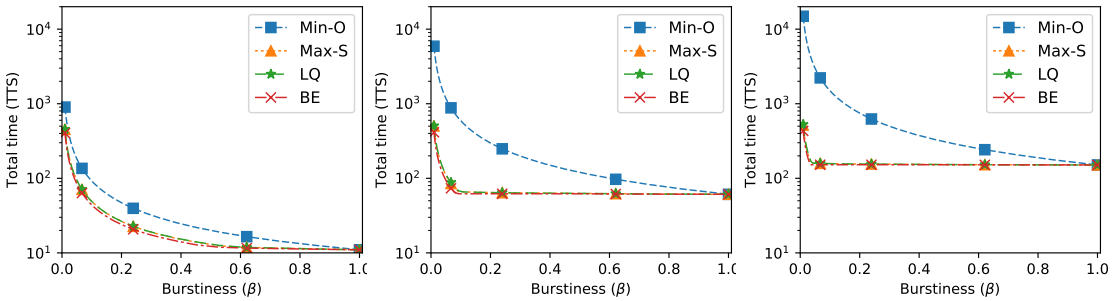
Figure 6.7: Homogeneous flow scenario: Link utilization for traffic shaping heuristics against simulation. Results are for phases ϕ_3 and ϕ_4 and n_{radius} set to 1, 3 and 5.

reduction is from 18 packets using BE, to 13 using LQ.

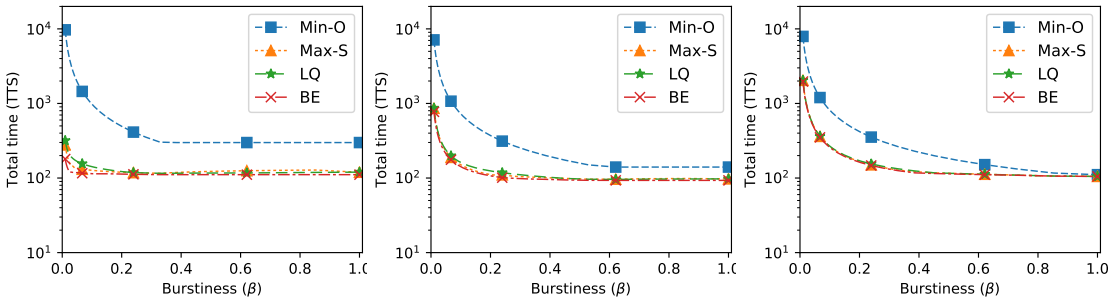
Another interesting observation occurs during ϕ_4 for the method Min-O. The maximum queue size gets smaller with increased burstiness. This is very counter-intuitive since we would expect that by injecting more traffic in the network, the congestion would increase. However, this phenomena can be easily explained mathematically: it is due to the way the method Min-O is defined. Looking at Figure 6.3, the flows duration defined as $\frac{\sigma}{\beta}$ are longer for lower burstiness β and thus for low values of β, the first points $\in \mathscr{T}$ (depicted by p_1, p_2, etc.) are farther from the origin. Since Min-O selects a point close to the origin as "anchor" point, its slope must be small so that the line remains below the function $S(t)$. With a low slope, it is likely that the vertical distance between the function $S(t)$ and the line will be high (in particular if $S(t)$ increases quickly). These phenomena can be observed, to a limited extent, in Figure 6.3.

6.5.2 Phase execution time for homogeneous load distribution

We compare the execution time of the phases ϕ_3 and ϕ_4 in Figure 6.8, again for the cluster sizes defined by $n_{\text{radius}} = 1$, 3 and 5 and varying the burstiness of the initial flows from 0.02 to 1 by step of 0.02. The execution times are computed by using Equation 6.9. As seen



(a) $\phi_3, n_{\text{radius}} = 1$ (b) $\phi_3, n_{\text{radius}} = 3$ (c) $\phi_3, n_{\text{radius}} = 5$



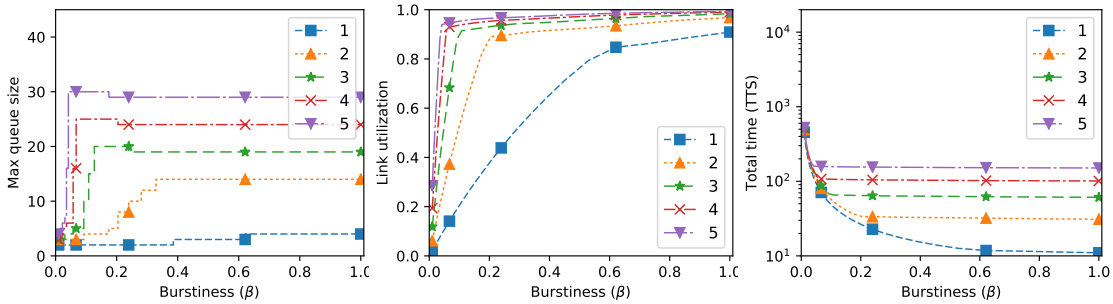
(d) $\phi_4, n_{\text{radius}} = 1$ (e) $\phi_4, n_{\text{radius}} = 3$ (f) $\phi_4, n_{\text{radius}} = 5$

Figure 6.8: Homogeneous flow scenario: Execution time of phases ϕ_3 and ϕ_4 for traffic shaping heuristics against simulation.

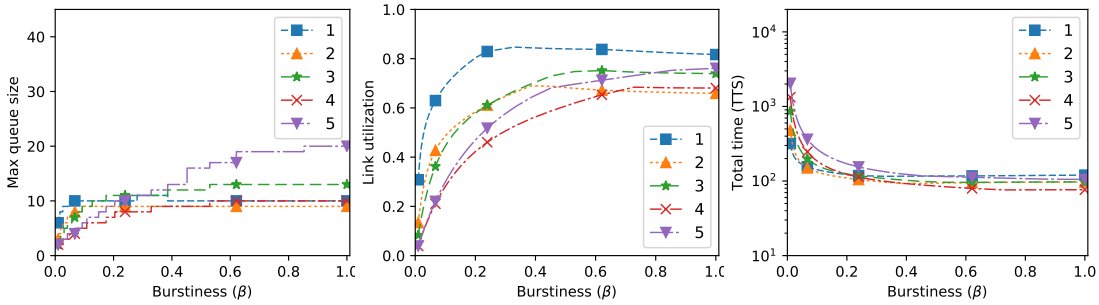
in all graphics of Figure 6.8, increasing the burstiness considerably reduces the execution time of the phases (note that the plots are in logarithmic scale), which remains constant after a point. This point is reached only for high burstiness in Figure 6.8(a) whereas it is reached almost immediately in Figure 6.8(b). This threshold, beyond which the execution time cannot be further reduced, can be explained by looking at the utilization of the input link of cluster-heads (for phase ϕ_3) and the sink (for phase ϕ_4). Those thresholds correspond to specific values of the burstiness for which the links saturate and therefore, any further increase in burstiness only results in larger queues but not in reduced execution time.

From the above results, we observe that the LQ heuristic performs better overall. We vary n_{radius} from 1 to 5, with β varying as before, for both ϕ_3 and ϕ_4. The results are shown in Figure 6.9. Figures 6.9(b) and 6.9(e) show the inverse relationship with n_{radius}. In ϕ_3, the smaller the n_{radius}, the smaller the clusters, and hence reduced traffic. In ϕ_4, there are more clusters transmitting to the sink, and consequently more traffic and link utilization.

It is also worth noticing that the increase/decrease on the link utilization changes non-linearly with n_{radius}. This is because the number of nodes in each cluster grows with the



(a) ϕ_3 (b) ϕ_3 (c) ϕ_3



(d) ϕ_4 (e) ϕ_4 (f) ϕ_4

Figure 6.9: LQ heuristic - (a) link utilization, (b) maximum queue size and (c) total execution time, with varying burstiness and $n_{\text{radius}} = [1,2,3,4,5]$.

square of the n_{radius}.

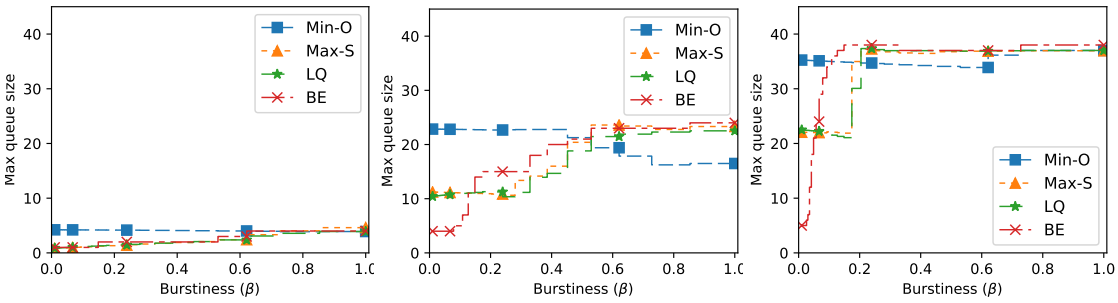
By looking at Figures 6.9(a) and 6.9(c) we can observe a property of the *LQ* heuristic (this also occurs for the other heuristics which are not shown for brevity). For all values of n_{radius}, both maximum queue size and total execution time remain constant (from $\beta >$ 0.4). So even when the link utilization is saturated, an increase in burstiness at flows' sources does not lead to worst queues size and total time. We explained this phenomenon in Section 6.5.1. The same behavior can mostly be observed also during ϕ_4 (Figures 6.9(d) and 6.9(f)).

6.5.3 Maximum queue size with heterogeneous load distribution

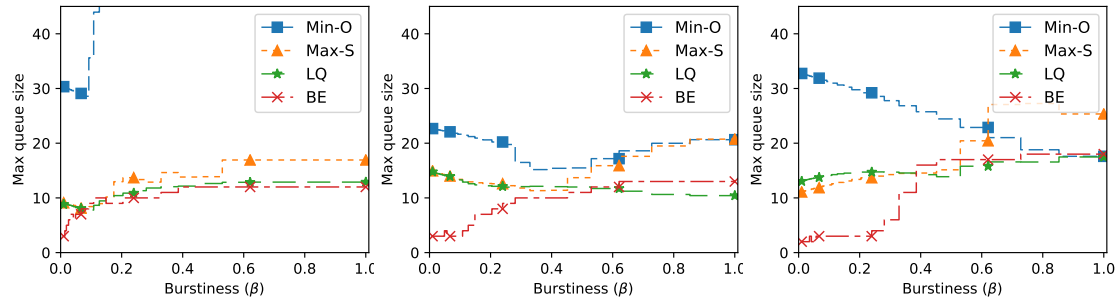
The results for maximum queue sizes for heterogeneous loads (see Figure 6.10) while tending to show the same trends as for homogeneous loads, present more chaotic behavior. BE performance is degraded. This implies that in more scenarios, applying traffic shaping is enough to reduce queue sizes compared to the case with homogeneous loads.

Also, in Figure 6.11, one can see that the link utilization due to heterogeneous load distribution, presents similar overall behavior when compared to the homogeneous scenario. This was expected, since the network load was intentionally designed to be approximately

the same. But apart from that, we can see that the Max-S heuristic performs better than before, specially during phase ϕ_4 (Figures 6.11(d) to 6.11(f)), in which it provides higher link utilization when compared to LQ (in most cases for $\beta < 0.6$), while keeping queue size and total execution time approximately the same.

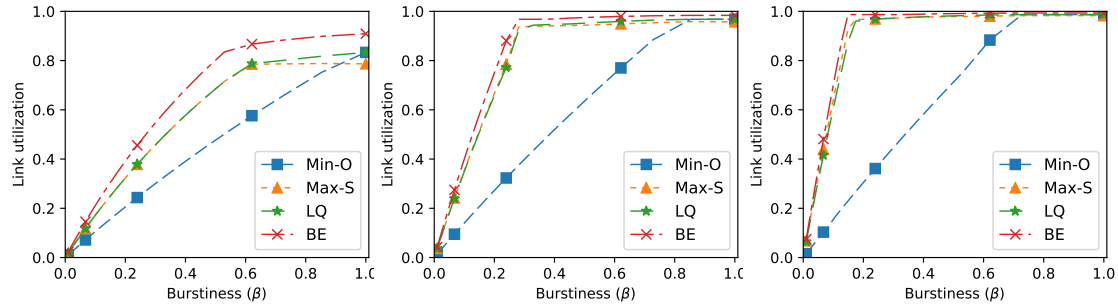


(a) $\phi_3, n_{\text{radius}} = 1$ (b) $\phi_3, n_{\text{radius}} = 3$ (c) $\phi_3, n_{\text{radius}} = 5$

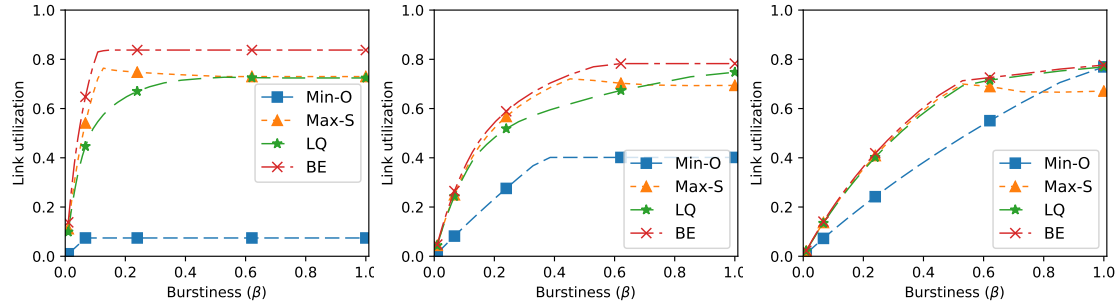


(d) $\phi_4, n_{\text{radius}} = 1$ (e) $\phi_4, n_{\text{radius}} = 3$ (f) $\phi_4, n_{\text{radius}} = 5$

Figure 6.10: Heterogeneous flow scenario: Maximum queue size for traffic shaping heuristics against simulation. Results are for phases ϕ_3 and ϕ_4 and n_{radius} set to 1, 3 and 5.



(a) $\phi_3, n_{\text{radius}} = 1$ (b) $\phi_3, n_{\text{radius}} = 3$ (c) $\phi_3, n_{\text{radius}} = 5$



(d) $\phi_4, n_{\text{radius}} = 1$ (e) $\phi_4, n_{\text{radius}} = 3$ (f) $\phi_4, n_{\text{radius}} = 5$

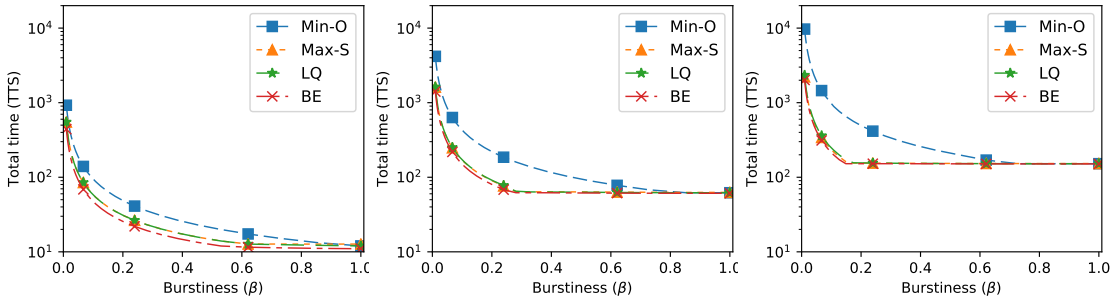
Figure 6.11: Heterogeneous flow scenario: Link utilization for traffic shaping heuristics against simulation. Results are for phases ϕ_3 and ϕ_4 and n_{radius} set to 1, 2 and 5).

6.5.4 Phase execution time for heterogeneous load distribution

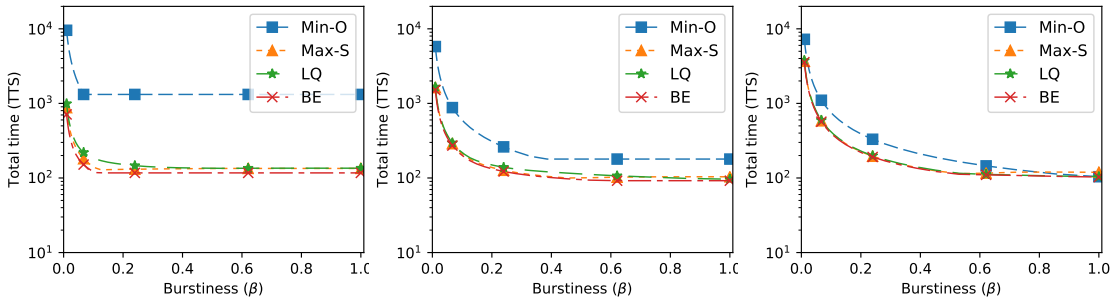
The total execution time gives similar results, since the total number of packets and the average burstiness among the nodes is the same for both scenarios. This is shown in Figure 6.12.

Once again, we take a closer look at the LQ heuristic alone, to understand the impact of n_{radius}. The results are shown in Figure 6.13 for phases ϕ_3 and ϕ_4. There is a clear drop in performance in all metrics for this specific heuristic. The same drop is not observed for the Max-S heuristic, that outperforms BE for heterogeneous flow sources.

To summarize, the heuristics Max-S and LQ, in both homogeneous and heterogeneous flow sources, perform close to that of BE that does not use traffic shaping. This means that by applying our heuristics for traffic shaping we are able to provide timing and resource usage determinism, and yet impose very little loss in terms of performance as compared to the best-effort approach.

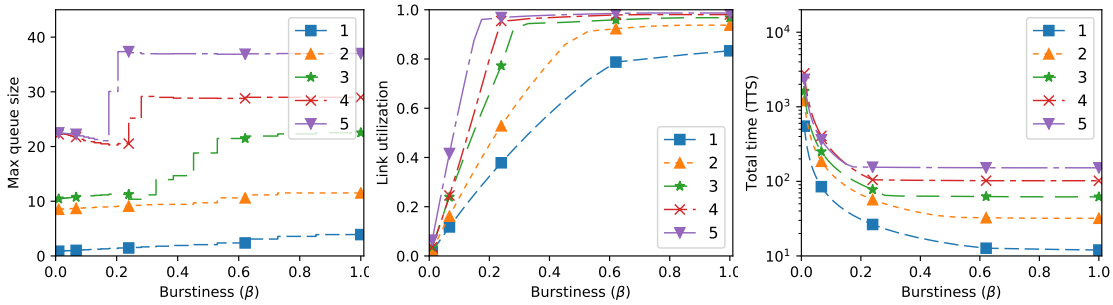


(a) $\phi_3, n_{\text{radius}} = 1$ (b) $\phi_3, n_{\text{radius}} = 3$ (c) $\phi_3, n_{\text{radius}} = 5$

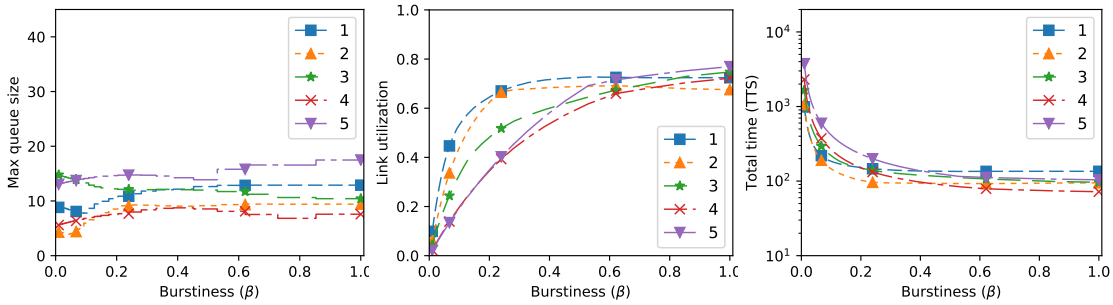


(d) $\phi_4, n_{\text{radius}} = 1$ (e) $\phi_4, n_{\text{radius}} = 3$ (f) $\phi_4, n_{\text{radius}} = 5$

Figure 6.12: Heterogeneous flow scenario: Execution time of phases ϕ_3 and ϕ_4 computed for traffic shaping heuristics against simulation.



(a) ϕ_3 (b) ϕ_3 (c) ϕ_3



(d) ϕ_4 (e) ϕ_4 (f) ϕ_4

Figure 6.13: Link utilization (a), maximum queue size (b) and total execution time (c) with varying burstiness, for the LQ heuristic only, for $n_{\text{radius}} = [1,2,3,4,5]$.

6.6 Concluding Remarks

The proposed traffic shaping heuristics enable us to endow XDense networks with real-time capabilities. We showed that on average, the performance of XDense is very similar with and without traffic shaping. This means that the proposed traffic shaping techniques allow determining timing and memory requirements while imposing minor performance overheads. The performance of XDense, in both homogeneous and heterogeneous scenarios also showcases its stable performance. However, further experiments exploiting distributed processing algorithms on CFD input data for AFC need to be performed (as discussed in Chapter 5).

We believe there are improvements that can be done to the model in many dimensions, but specially towards making it more accurate to the hardware platform we target. The model could also be expanded, in order to cover different application scenarios. Other concluding remarks are further discussed in Chapter 8.

Chapter 7

Hardware Implementation and Performance Evaluation

7.1 Introduction

We target low-cost and low-complexity hardware, COTS components, coped with communication protocols that should exhibit low overhead. This should enable reasonably high performance and scalability (in terms of node count and cost) with low cost and low resource utilization.

Cost is mainly determined by the complexity of the node's hardware and network interconnection. This includes the cost of the chosen microcontroller (μC) and the additional components required, as well as the density and length of interconnections among the various nodes (for on-board, over cable or wireless communication). Performance is usually measured in terms of node's computational power, maximum communication bandwidth and latency. These parameters are defined both by the microcontroller frequency of operation, and by the network topology, reliability and power constraints [124].

Considering the above, in the next sections we detail the hardware implementation details and its performance evaluation.

7.2 Hardware Design

We co-designed the simulation model and hardware and embedded firmware, meaning that for each of the implemented features on the simulator we considered its feasibility on the targeted hardware. This was done keeping in mind the implementation of XDense as per specification, but also taking into account the realistic performance achievable with COTS hardware.

This is an iterative co-design problem involving software and hardware specifications, along with simulation. The objective is to converge to a solution that provides the best fit for the requirements, with the correct balance between performance, processing overhead and cost of implementation. In the end, we should have a best fit hardware solution that can be simulated faithfully; that is, performance measurements from the network implementation and simulation should *not* differ substantially among each other.

In this chapter we discuss the co-design considerations during the development of the XDense hardware. We then develop and measure the hardware performance. These measurements can then become a parameterized input to XDenseSim. This allows XDense hardware performance to then be compared with its simulation results. We also identify the impact and constraints of alternate implementation approaches.

We specify the internal hardware architecture based on the XDense model defined in Chapter 4. We first present its implementation using a field programmable gate array (FPGA) with custom designed circuitry, and later using a COTS solution based on a μC. The intention is to first measure the performance with the FPGA implementation, and then use its performance results as a reference when evaluating the μC COTS implementation.

7.3 System Requirements

We establish the three basic requirements that we use in designing the system that will drive performance and cost requirements.

1) Simplicity of hardware: Nodes (and other components) should be simple in terms of hardware and software. This requirement is driven by the need for developing building blocks for extremely dense deployments with miniaturized nodes. Simple nodes with modular elements allows cost effectiveness and scaling;

2) Application defined performance: The controller should have *fast enough* communication links and processing capability, with respect to the application requirements.

3) Communication ports: XDense nodes require at least four serial ports (considering its architecture). An extra port is practically is useful for debugging purposes or for providing a link to external supervisory systems;

The hardware may also impose other limitations regarding memory capacity (which may limit the maximum packet queue sizes in nodes) and internal delays due to communication overhead. The internal delays are usually small as compared to communication delay, but they can still influence the overall performance and have an impact on the predictability of the system.

We believe a custom designed integrated circuit (IC) would provide the best-fit solution. However, the design becomes less flexibile and oriented towards a single-application. Also, the overhead of development and initial costs of production of a custom design IC is prohibitive for this stage of the development.

Our goal is to have a hardware platform and software framework that is flexible, scalable and that allows customization of applications to a variety of scenarios.

7.4 Design Decisions

We take a bottom up approach towards designing the platform; that is, from the physical and link layers up to the application layer.

We looked into other mesh-grid networks which are seen on many-core general purpose processing units (CPUs) [41], and graphical processing units (GPU) [134]. These cores are connected point-to-point with four or more of their neighbors. The connection links are 64 bits (or more) wide full-duplex parallel links, through which transmissions happen in a single clock cycle [135].

However, it is known that on-chip short-range interconnections costs less than long-range off-chip connections. To communicate off-chip, one possibility is to use general purpose input output pins (GPIOs). However, the cost of GPIOs on μCs and FPGAs is high. Moreover, it usually requires additional logic and circuitry for communications purposes. This is the main reason why inter-devices parallel communication is increasingly less prevalent nowadays [136].

For off-chip interconnection, the most common design approach is to serialize data at very high bitrates, using the minimum number of wires possible. There are many examples that show this trend inside the industry, such as I2C and SPI buses for on-board master-(multiple) slave communication, and Ethernet, SATA, USB and USART for point-to-point inter-device communication [137].

As stated in our design model, our decision is to use full-duplex, point-to-point links for inter-node communication. This avoids (i) the restrictions imposed by master-slave based communication (as in SPI and I2C), and (ii) the limitations imposed by half-duplex links (such as in I2C and USB).

This now limits our choices down to Ethernet and UART ports. In the context of XDense, UART ports have many advantages.: i) they are commonly found on numerous COTS μCs; ii) they do not require external components and circuitry to establish electrical interface between nodes; iii) at short distances they present very small bit error rates; and

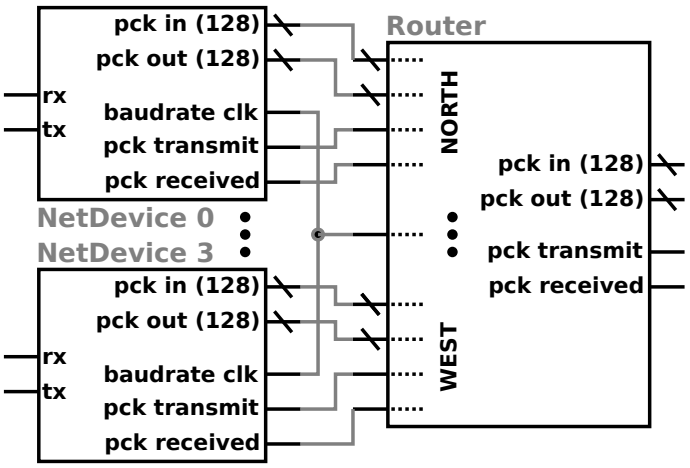


Figure 7.1: Simplified schematic of the XDense node's Switch and Net-Device implementation using on a FPGA.

iv) they consume fewer resources in terms of power, chip-area, and protocols overhead. The authors in [138] comment on the benefits of the UART communication ports.

UART ports also present other advantages for both FPGA and micontroller implementations. Firstly, UART logic-blocks for FPGAs are open source and widely available. Secondly, UART ports can be easily accommodated on low-end FPGA chips due to their small circuit footprint; Finally, worth to mention that low/mid-range low pin count (< 32 pins) μCs can have up to five serial ports, making them a good fit as a low-cost μC COTS hardware solution.

The main limitation of UART ports is that they can only operate at a fraction of the main clock frequency, usually 1/16 of the main clock. This leads to maximum bitrates way below the ones on USB and Ethernet ports. For example, in a μC running at 100 Mhz, communication can only be done at most at 6,2 Mbps.

7.4.1 Composing an FPGA XDense node

As a reference of maximum performance, and to demonstrate the potentials of a custom IC, we designed a XDense node using FPGA. We compare its performance with the ones achieved with the COTS implementation.

We model the router and network devices in hardware, while having the application layer running on a software core. The networking devices are instances of a customized UART port with controllable input/output queues sizes. The router uses a custom design circuit that interconnects each UART instance with a dedicated parallel connection. Dedicated connections allow minimum internal delays. It also has signaling bits that allow the router to monitor and control each networking device individually. The router is packet

Table 7.1: FPGA implementation: Resource utilization due to XDense's communication logic.

	Available	Utilized	Utilized (%)
Registers	5,720	419	3%
Lookup tables (LUTs)	11,440	415	7%
Memory	1,440	16	1%

switched, and therefore can forward packets between ports without the intervention of the processor.

The width of the bus between the router and networking devices defines how many bits can be transferred in parallel, and consequently in how many parts a packet has to be split to be transferred. In order to maximize performance, we make this bus wide enough to be able to transmit a single packet in one transaction (16 bytes wide). This is so that a packet can be transferred between networking devices and router in parallel (in both directions), in a single clock cycle. Figure 7.1 shows the internal architecture of networking devices and router on a node.

Each router is connected to four networking devices and provides to each a clock that defines the baudrate. On the right side of the router (in Figure 7.1), there is a fifth dedicated bus to connect the router to the software core. The software core was not implemented in this case, as we are interested in measuring only communication delays. Therefore, it is sufficient to implement the router and the networking devices in order to generate and/or forward packets. The performance of the software is tested separately using the software core on a μC implementation. This is discussed in the next section.

We use a Xilinx Spartan 6 FPGA [139] for our implementation. This is a low-end and low-cost FPGA, which is a programmable alternative to custom ASICs. The resource utilization is shown in Table 7.1. Although we use a relatively small FPGA (in terms of resource availability), the logic required for the networking devices and router still consume very little of the available resources. Even having 16 bytes wide internal buses.

The implementation floor plan is shown in Figure 7.2 with the utilized logic blocks highlighted. It is an implementation optimised to performance, and therefore it consumes more logic and space inside the FPGA than necessary. The current implementation is inefficient in terms of resource utlization, since we only focus on achieving maximum performance with the available resources.

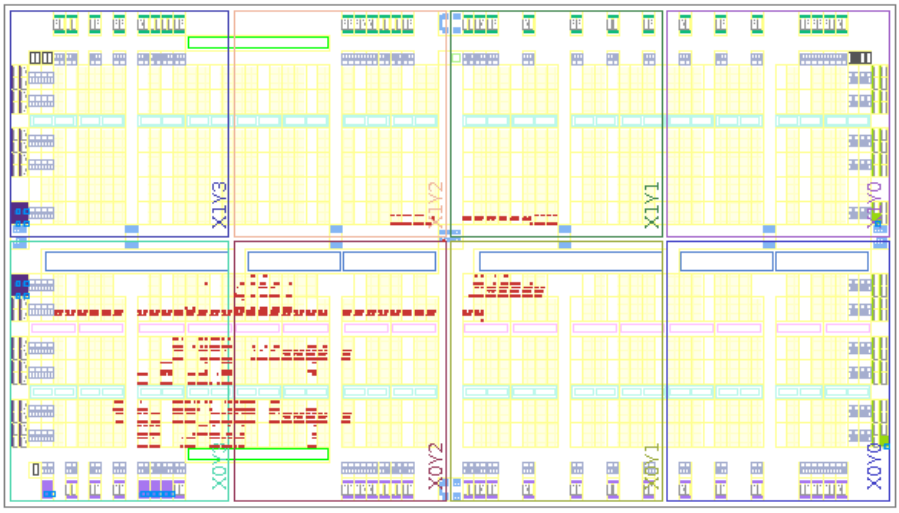


Figure 7.2: FPGA implementation: floor plan of the router and four networking devices.

7.4.2 Microcontroller Based XDense Node

As a lower cost alternative to the FPGA custom design, we have also developed a COTS prototype of XDense.

Given the requirements, we are restricted to a limited number of candidate μCs. Especially concerning our requirement of number of available high-speed serial ports. Few μCs have five or more serial ports and exhibit, at the same time, a low pin count, low size, and low cost. Availability of direct memory access (DMA) peripherals is also desirable to reduce the processing time spent on communication.

We chose the Atmel ATSAM4N8A μC, a 32-bit ARM Cortex-M4 RISC [140] processor implementation, which is a mid-range general purpose μC. It runs at 100 MHz and provides a good balance between power consumption and processing power. With digital signal processing (DSP) extensions and a floating point unit (FPU) co-processor, it has enough computational power for a relatively high power efficiency. It has a small 48-pin footprint, with five high speed UART ports, each with a dedicated DMA channel that allows transmission of packets in parallel, without consuming CPU time.

To implement temporal predictability, we use the FreeRTOS [141] real-time operational system (RTOS) in our nodes. It is a free open source real-time RTOS for 32 bits μCs. It provides part of the required device drivers, and additional high level abstractions, such as context switching and multi-tasking.

The schematics and prototype node are shown in Figure 7.3. We placed four different sensors on the top of the board for sensing. These include a motion sensor with 9 degrees-of-freedom, a pressure sensor, a temperature sensor, and a visual-range and light color sensor. Each of the sensors are related to sensing for AFC airflow monitoring. The motion

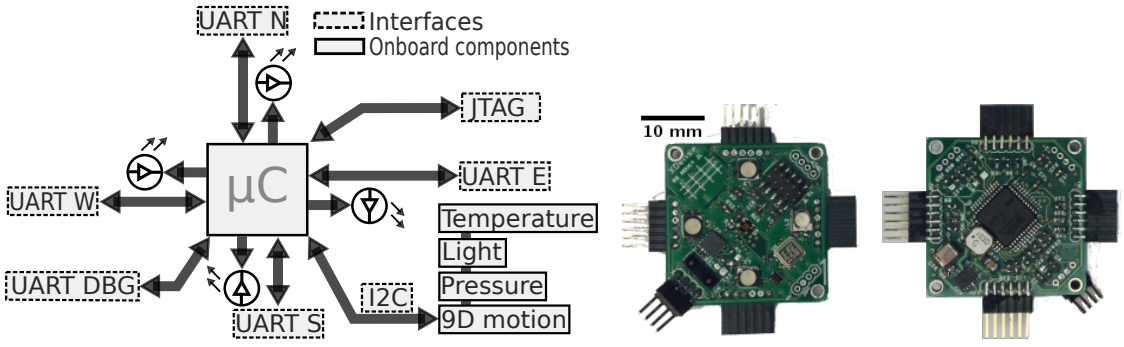


Figure 7.3: (a) Node's schematic showing each of the major components of the system. (b) and (c) show the top and the bottom sides of the PCB, respectively.

sensor can potentially provide information about vibrations on the wing, which may be related to the airflow characteristics [142].

Pressure and temperature sensors provide direct and indirect measurements about the characteristics of the airflow. Data from both sensors can be used complementary. The visual-range light sensor, while having other uses, is meant for testing and debugging during development, by using a beamer to project visual data from the phenomena over the network, to allow studying the behavior of the network when exposed to the (emulated) expected data. Figure 7.4 shows a 3×3 deployment connected to a computer that shows the acquired sensed values using color scale. A lantern is pointed to the lower part of the network, what justifies the measurements.

Four RGB LEDs are used for debugging purposes, to either provide a direct visual feedback about the sensed data, or to transduce any kind of output values to colors. This can be used during development to represent actuation actions.

Table 7.2: Resource utilization of the Atmel ATSAM4N8 ARM Cortex M4 running XDense.

	Available	Utilized	Utilized (%)
RAM	65536 bytes (64 kb)	57792 bytes	88.2%
Flash	524288 bytes (512 kb)	57684 bytes	11.0%

The Table 7.2 shows the memory utilization of the software implementation. Even though we use an RTOS, the code still consumes few flash memory. In contrast, the RAM utilization is high. This is caused mainly by the RTOS, that statically allocates memory for tasks stack for multitasking, for the deepest possible nesting of function calls.

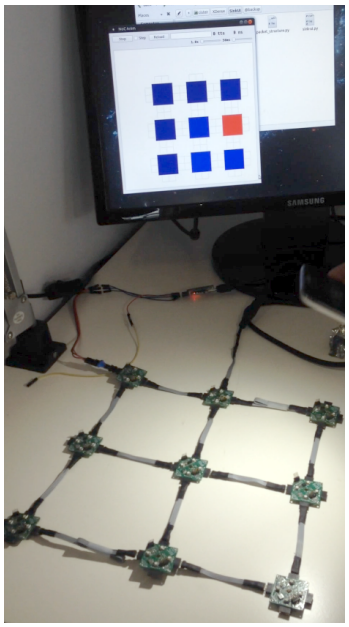


Figure 7.4: Deployment of a 3×3 network connected to a computer that shows the acquired sensed values using color scale.

7.4.2.1 Software Framework

To bring up the XDense abstractions to a functional node hardware we started by porting the application, communication and routing protocols code from XDenseSim to our μC firmware. Additionally, we developed the router abstraction to coordinate all the serial ports using DMA. We also developed device drivers to support the external sensors, and a middleware that abstracts the hardware specific code to interface with the application layer ported from XDenseSim.

We implemented multiple independent tasks to realize XDense. These tasks are scheduled by the RTOS according to our specifications. There are four different mutual exclusive tasks, each one to read one of the external sensors in the I2C bus, according to their maximum sampling rate. The router has its own task that handles transmissions from/to each serial port. It uses each port dedicated DMA channel to receive and/or transmit in parallel from/to any of the serial ports.

Any node, but at least one, should be connected to a host machine running a supervisory system through any of its communication ports. But notice that the debugging port is the only extra port available on nodes which are not in the edges of the network, since they are connected to four neighbors. That is, only the nodes in the edges have unconnected ports in addition to the debugging port. This allows the host machine to interact with the node using our protocols, for debugging, firmware writing, performance measurements and for setting configurations, such as neighborhood size, sampling rate, baudrate, sinks

election, sensors thresholds, and other specific configurations of the routing, applications and communication protocols used. The host machine utilizes the same protocols and behaves as any other node on the network, that can unicast/multicast/broadcast configurations or any other data once it is connected to the network.

Another important implementation feature we provide it the possibility of performing in-network programming of firmware. The action is started by the host, which puts the node it is connected to, into programming mode and programs it. After programmed, the node starts to propagate its newly received firmware to its neighbors, one by one, by putting each at a time into programming node, and copying its own FLASH content to it through one of the communication ports. This process continues node by node, until the entire network is re-programmed. We use specific routing algorithms to propagate the firmware through rows and columns, in a way that nodes are programmed only once.

7.5 Performance Evaluation and Comparison

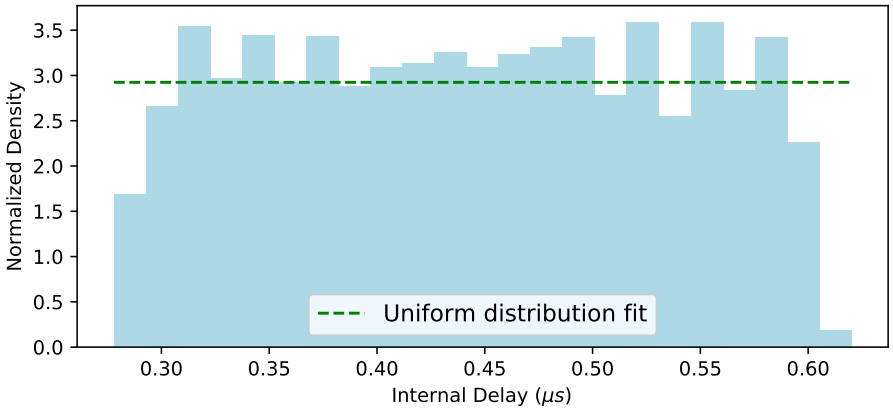
We now setup a controlled experimental scenario to measure the internal delays that affect the total communication time, and its impact on the network performance on both the FPGA and the μC implementations. We compare the measurements to the values obtained with simulation.

We perform two sets of experiments to measure the internal delay: (i) we measure the time a packet takes to travel through a single hop, on both the FPGA and the μC implementations; (ii) we measure the time a packet takes to travel through multiple hops on the μC implementation. For this metric, we vary the hop distances and compare the delays with the results obtained using XDenseSim. We also measure the packet drop ratio (PDR) on the μC implementation. Table 7.3 shows the experiment configuration.

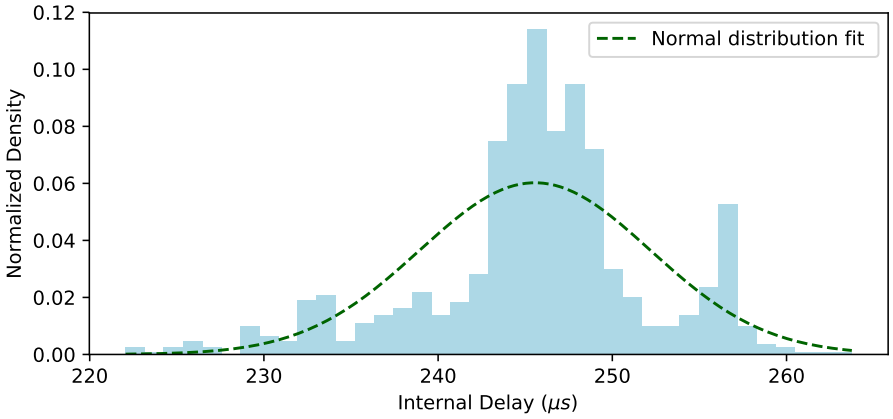
Table 7.3: XDense testbed configuration.

Parameter	**Value**
System clock	100 MHz
Baudrate	3 Mbps
Packet size	10 bytes $\times$ (8 + 2) = 160 bits
Packet duration (t_{pck})	53.3 μs

In order to enhance XDenseSim we perform numerous measurements, for a statistical survey, so we can provide a probabilistic model to allow simulating internal delays.



(a)



(b)

Figure 7.5: Packet forwarding internal delay on a single hop, using (a) a FPGA-based node and (b) a μC-based node.

7.5.1 Experiment 1: Single Hop Delay

In this experiment we measure a node's internal delay, and the variability of it throughout several measurements.

We connect a single node to a host machine that has a random packet generator. These packets are received and forwarded by the node, in which we measure the time elapsed from the time instant the packet is completely received by the node (coming from the host machine), until the time the packet starts to leave the node. With the aid of an oscilloscope and an automation script, we performed ten thousand measurements on the same node. We run this experiment for both the FPGA and the μC implementations.

The histograms and Probability Density Functions (PDF) measurements are shown in Figure 7.5. The number of bins is selected according to the Freedman Diaconis estimator, which is a robust estimator (resilient to outliers), that takes into account data variability

and data size [143].

Figure 7.5(a) shows the results for the FPGA implementation. It is is an approximate uniform distribution, centered at $0.44\mu s$. The minimum delay measured is $0.28\mu s$, which is the intrinsic delay due to signal propagation imposed by the FPGA circuitry. The maximum measured delay is $0.62\mu s$, which happens to be $0.34\ \mu s$ greater than the minimum intrinsic delays. This value is due to an offset between the host machine and the node's clock. Because packets are only processed by the FPGA on the rising edge of the communication ports clock, there is a uniform random delay between the time the starting bit arrives, until the FPGA actually detects it (since it depends on the next observed rising edge). The difference between the maximum and minimum delay ($0.34\ \mu s$) is the period of the communication clock. Its inverse gives us the approximate baudrate, which is $2.94 \approx 3Mbps$.

Notice that the maximum internal delay represents only $0.62 \div 53.3\ \mu s = 1.16\%$ of the packet duration. This is close to the optimal delay achievable on a node running at 100 MHz, with UART at 3 Mbps communication rate (as shown in Table 7.3).

For the μC implementation, Figure 7.5(b) shows the histogram and PDF of the measurements taken. In this case the distribution is more random. It ranges from 222 to 263 μs, with 246 μs average delay.

The minimum time required by the RTOS to consume, process and serve the packet is $222\mu s$. This delay is strictly related to the RTOS implementation and configuration specifics, and how tasks are defined. Even though we configured the RTOS for maximum performance, without concurrent transmissions, with interrupt based receptions (for better responsiveness), the processing overhead added by the RTOS is still significant.

The peaks at 233, 246 and 258 μs in the histogram are caused by a concurrent RTOS task with higher priority that frequently interferes with the reception.

The internal delays obtained with the μC node are much greater when compared to the FPGA implementation ($500\times$). The maximum internal delay of the μC implementation is 263μs and therefore $= 493\%$ of the packet duration.

Although the PDF is a rough approximation to the histogram, we believe it may still be a good approximation to model a node's internal delay. With that, a simple model with low overhead could be implemented for XDenseSim in order to simulate the internal delays.

7.5.2 Experiment 2: Multi-hop Delay

With PDFs to model internal delays, we can now implement them in XDenseSim, and compare with the hardware implementation in a multi-hop scenario. We start by measur-

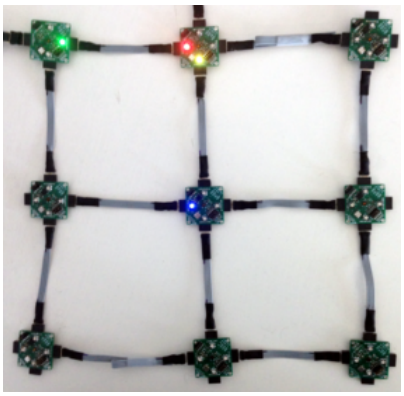


Figure 7.6: 3×3 testbed deployment.

ing a single packet traversal time on both implementations. This is the time taken for a single packet to travel through n hops without any other concurrent workload. We vary n from 2 to 100 hops and measure the end-to-end delay at each scenario. We perform this experiment ten thousand times. We use our 9 nodes test-bed to emulate a much larger network. We do this by connecting nodes from one extreme to the other extreme of the network so that the packet continues on a virtually bigger network. Figure 7.6 shows the μC deployment with 9 nodes.

Figure 7.7 shows the comparison between the average end-to-end delay on different multi-hop scenarios for the μC and the FPGA implementations. We compare the measure values with the XDenseSim results, with and without internal delay model. The FPGA implementation represents a very small increase on the theoretical minimum delay, given by the XDenseSim without the internal delay model.

There is a linear growth on the trip delay as the trip distance increases, for all four scenarios.

Internal delays represent an approximate five-fold increase in the delay.

When XDenseSim is used with internal delay model, it approaches considerably the performance of the μC implementation, with a linear increasing error (meaning that the PDF captured is not an accurate fit to the real values).

Figure 7.8 shows the distribution of the measured delays from our μC implementation and XDenseSim, on multi-hop transmissions with 2, 20, 60 and 100 hops. For short trip distances (2 hops), both scenarios show approximate same average value. whereas simulation present more confined normal distribution (expected according to the PDF), compared to the more random and wide distribution observed in hardware. As we increase the trip distance (Figure 7.8(d)), the averages start to diverge, while the distribution for the hardware tends to get more confined around the average when compared to the simulation.

In Figure 7.9, we compare the packet drop ratio measured for the μC implementation

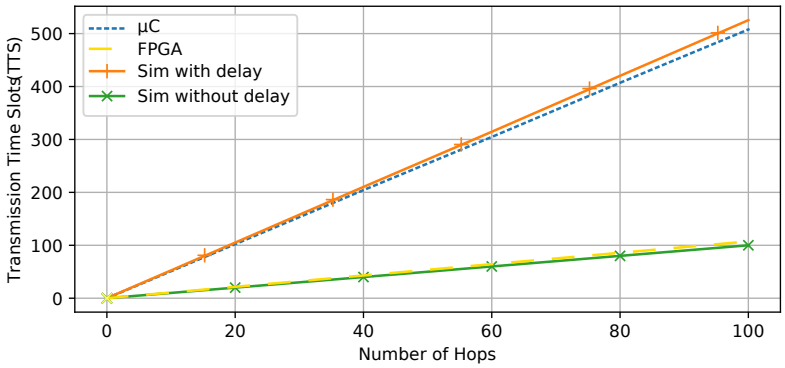
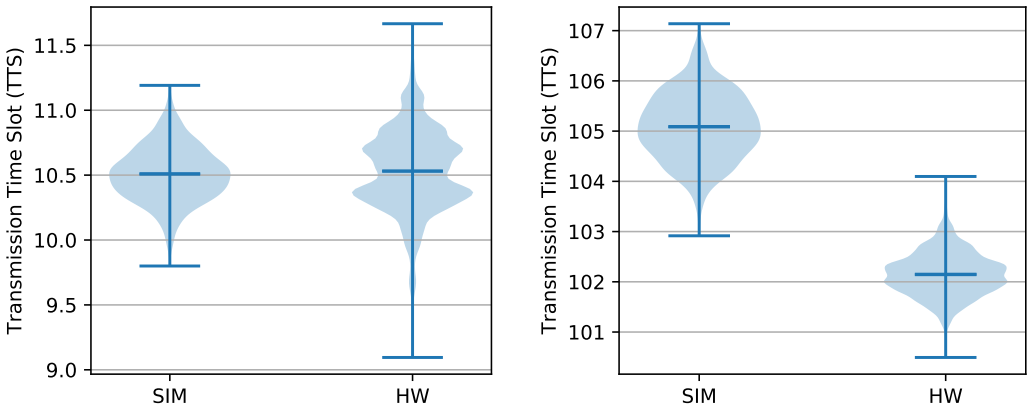


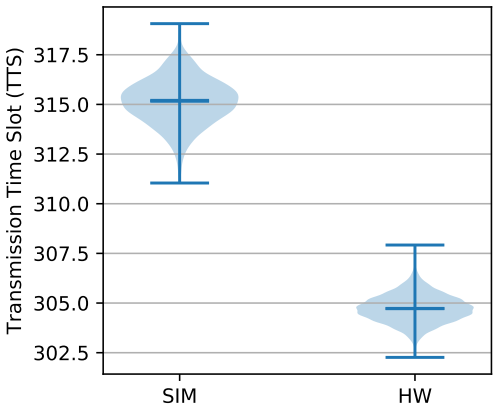
Figure 7.7: Average trip delay in a multi-hop scenario for varying trip distances.

and XDenseSim. Since we do not implement error models, no packets are dropped in the simulation. While in hardware, we observe a linear growth on packet drops as the trip distance grows. At 100 hops trip distance, around 30% of the packets are dropped. Such high drop rate was due to the overhead having the router implemented in software, and the overhead of queuing/dequeuing packets at high communication rates. The poor implementation of the queuing algorithms and operational system overhead also play a role in this results. We believe this number could be drastically reduced at lower communication rates and with a more efficient implementation of the router and queuing algorithms, even though we refrained from doing it in this work.

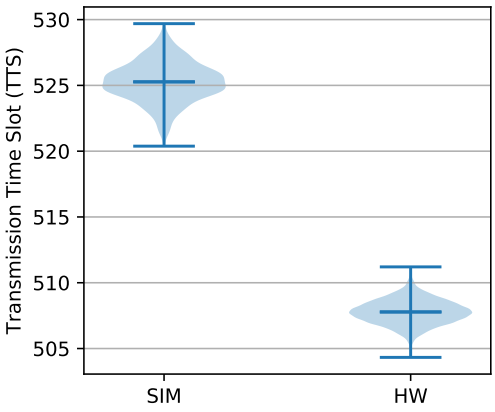


(a) 2 hops

(b) 20 hops



(c) 60 hops



(d) 100 hops

Figure 7.8: Comparison between simulation and hardware of the packet trip delay distribution, for different number of hops.

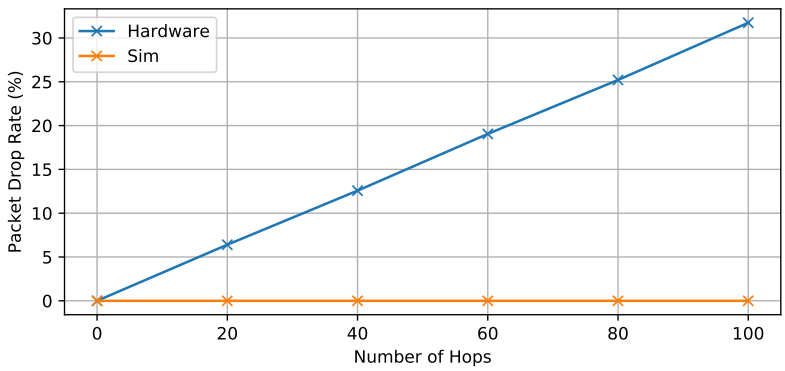


Figure 7.9: Packet drop ratio in a multi-hop scenario for varying trip distances.

7.5.3 Current Limitations

Even though we provide a functional μC implementation, the internal delays measured exceeded our initial expectations. This could be prohibitive for applications with stringent temporal requirements.

We have tried a few methods to reduce these internal delays. We first tried to configure the RTOS for increased responsiveness. We did this by increasing the frequency in which it performs context switching between the tasks responsible for receiving the packets. We were able to reduce the internal delays by 20% on average. However, we also made the system more unpredictable due the increased overhead due to the excessive context switches.

In addition, we built a second firmware implementation without the RTOS. This implementation uses a interrupt-based reception to immediately process incoming packets. By doing so, we have been able to drastically reduce the internal delays.

Figure 7.10 shows the histogram and PDF of ten thousand measurements of the internal delay for the non-RTOS implementation. The internal delay range from around 3.7 to 4.7 μs, with a 4.25μs average, which represents only 9% of the packet duration. This is a drastic decrease that allows for much more performant nodes. The drawback of not using the RTOS, lies mostly on lack of certain abstractions and functionalities, such as periodic, preemptive tasks with guaranteed temporal behavior.

It is also important to notice that, depending on the implementation, there may be a secondary source of delay due to concurrent transmissions. To investigate that, we setup a new scenario in which a node receives a single packet, and immediately forwards it to all its communication ports (including the port it received the packet from).

Figure 7.11(a) shows the waveform of the input and output packets on the FPGA, and the measured forward delay. We observe that as soon as the topmost waveform ends, the

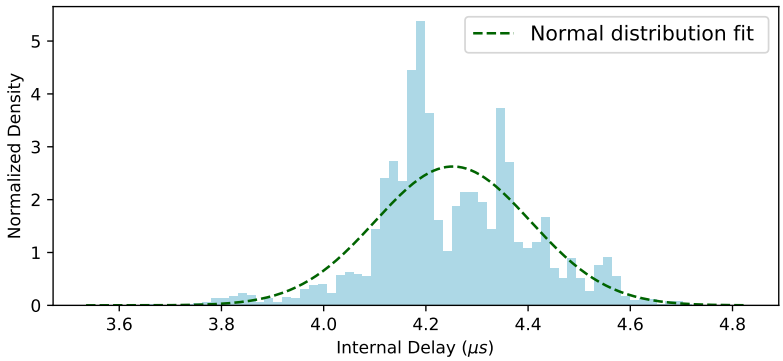
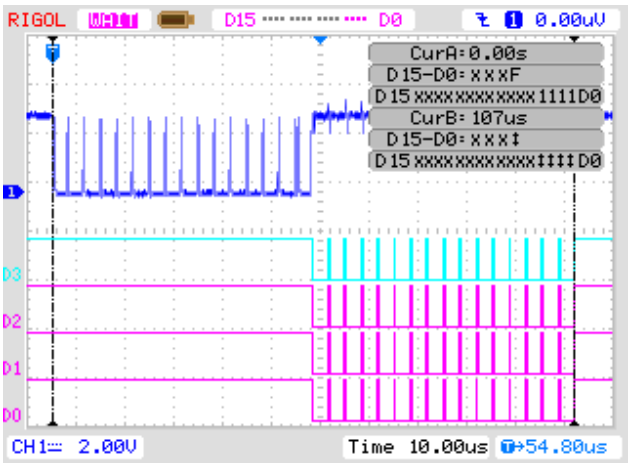
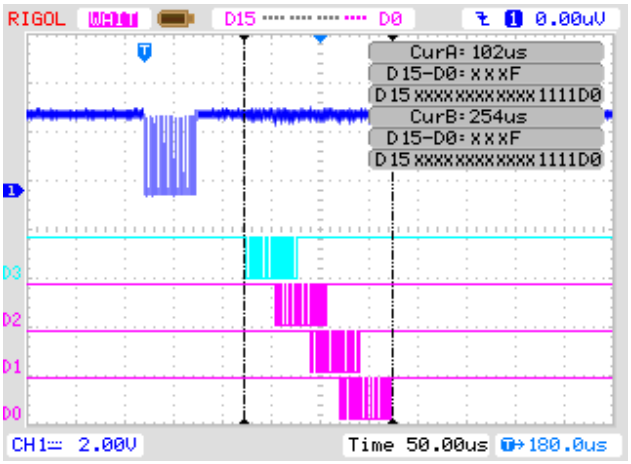


Figure 7.10: Packet forwarding internal delay on a single hop without RTOS.



(a)



(b)

Figure 7.11: Waveform showing the internal delay due to concurrent transmissions on (a) FPGA implementation and (b) μC implementation. Note the difference in time scales.

four other ones start and finish simultaneously. In other words, as soon as the incoming packet is received, it is immediately forwarded in parallel through the four ports.

The same can not be observed in the μC implementation (Figure 7.11(b)). Here there is an intrinsic delay between the reception and trasnmission of the packet, as well as a delay between the transmission at each output port. The delays happen due to the time required by the CPU to configure and activate the DMA transfers from the input port to the RAM, and from the RAM to each output port individually.

The delays measured for the μC implementation show the challenges related to the predictability of the software implementations, which also impacts on the accuracy of the XDenseSim delay model, since we base it on an statistical distribution. These results also reflect the advantages of having dedicated hardware, capable of handling parallel transmissions with much superior performance.

7.6 Concluding Remarks

We provide a complete implementation of our network model, that approaches satisfactorily the performance observed on real implementation platform. The implementation proved to be robust, with stable operation, providing access to a diversity of performance metrics for XDense.

This internal delay model for XDenseSim is still too simplistic. To make it more accurate, we should also account for potential packet drops, and any other source of internal delays and uncertainty.

Both hardware implementations shown to be feasible, with potentially low internal delays (very implementation specific), and with the potential to perform various of the desired XDense functionalities.

Chapter 8

Conclusions and Future Directions

In this chapter we revisit the research objectives and the results obtained with this work as a whole. We comment on how our contributions fulfilled the original research objectives. Some guidelines for future research work in the area are also provided.

8.1 Summary of Contributions

We divide our contrbution in mainly three topics:

(a) Efficient data extraction:

In Chapter 5, we evaluate various performance metrics on accuracy of acquired data, delays and utilization of the network while varying parameters such as the input data, neighborhood size and feature detection algorithms. We show the resulting trade-offs in performance. Both the nature of the input data and the feature detection algorithm utilized strongly affect the performance.

Most importantly, performing in-network data compression, and feature detection and extraction leads to drastic reductions on the time required to sample the data.

(b) Real-time behavior: We provide traffic shaping heuristics that endow XDense with real-time capabilities. The proposed traffic shaping techniques allow determining timing and memory requirements by shaping nodes transmission, while imposing minor performance overheads.

We believe the proposed framework can be used in other kinds of synchronous multi-hop networks. The requirements are that the flow sources (the nodes from which the packets are generated) are known, and can be modeled according to our flow model. This should allow us to shape every output flows on the network, and provide real-time guarantees based on calculations. We believe that there are other approaches on

designing new traffic shaping heuristics that would provide reduced queue sizes and delays compared to the actual ones.

(c) Scalable infrastructure: It is important to note that the simplicity of the architecture allowed for practical construction of such networks using currently available technology. A prototype test-bed was developed and is operational. The simulations in this work establishes the competence of XDense and our next step is developing a prototype for experimental evaluation. A low-cost solution is being developed with COTS microcontrollers (see Chapter 7).

We believe to have enough evidences that the XDense sensor network have the potential to enable real-time dense sensing. Combined with its novel architecture and feature detection techniques, it is possible to sense the phenomena and extract high level information using in-network processing in real-time.

XDense enables dense deployments that fulfill the spatial and temporal granularity required by AFC applications, by keeping low spatial and temporal complexity (minimally influenced by the large number of nodes on the network).

8.2 Future Directions

The simulator developed provided the infrastructure required for evaluating XDense network accurately. However, some temporal aspects still have to be more carefully addressed in the simulation model, such as processing delay, clock asynchronism, concurrency issues and other sources of randomness.

Also, the implemented function modules affect performance of the system and depend closely on the application scenario. In the same direction, the data input should represent more accurately the real phenomena, both spatially and temporally, from accurate computational models or from real data logs.

Improvements to the model can be made along many dimensions. One would be to bring in computational fluid dynamics data to analyze the performance of the model vs synthetic data. The model can also be improved with more accurate portrayal of hardware (for example, to consider the internal delays). One way to do this is to measure delays on real hardware and incorporate it. We have already made some progress along these lines [144].

More experiments on a large test-bed are required for better characterization of the real system. Next step should include the development of an application oriented node which meets as close as possible the specific requirements of aviation applications. A real deployment with several nodes in a wind tunnel should allow validating the real system.

Furthermore, the proposed analysis framework can also serve as a basis to reason on the dimensioning of the system; in the choice of network size, the clusters size, and other configuration that may impact on the performance. A framework provided for this purpose would also be of a valuable future work.

Bibliography

[1] N. Kasagi, Y. Suzuki, and K. Fukagata, "Microelectromechanical systems–based feedback control of turbulence for skin friction reduction," *Annual Review of Fluid Mechanics*, vol. 41, no. 1, pp. 231–251, 2009. [Online]. Available: https://doi.org/10.1146/annurev.fluid.010908.165221

[2] U. Buder, L. Henning, A. Neumann, and E. Obermeier, "Aeromems wall hot-wire sensor arrays on polyimide with through foil vias and bottom side electrical contacts," in *Solid-State Sensors, Actuators and Microsystems Conference, 2007. TRANSDUCERS 2007. International*. IEEE, 2007, pp. 2333–2336.

[3] F. Kopsaftopoulos, R. Nardari, Y.-H. Li, and F. Chang, "Experimental identification of structural dynamics and aeroelastic properties of a self-sensing smart composite wing," in *Proceedings of the 10th International Workshop on Structural Health Monitoring, Stanford, CA*, 2015.

[4] J. Ohta, T. Tokuda, K. Sasagawa, and T. Noda, "Implantable CMOS Biomedical Devices," *Sensors (Basel, Switzerland)*, vol. 9, no. 11, pp. 9073–9093, Nov. 2009, 00028 PMID: 22291554. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3260631/

[5] J. Loureiro, M. Rangarajan, Albano, T. Cerqueira, Raghuraman, and E. Tovar, "A module for the xdense architecture in ns-3," *Workshop on ns-3 (WNS3)*, 2015.

[6] C. B. da Silva and J. C. Pereira, "Invariants of the velocity-gradient, rate-of-strain, and rate-of-rotation tensors across the turbulent/nonturbulent interface in jets," *Physics of Fluids (1994-present)*, vol. 20, no. 5, p. 055101, 2008.

[7] J. Westerweel, C. Fukushima, J. M. Pedersen, and J. Hunt, "Momentum and scalar transport at the turbulent/non-turbulent interface of a jet," *Journal of Fluid Mechanics*, vol. 631, pp. 199–230, 2009.

[8] R. Que and R. Zhu, "Aircraft aerodynamic parameter detection using micro hot-film flow sensor array and bp neural network identification," *Sensors*, vol. 12, no. 8, pp. 10 920–10 929, 2012. [Online]. Available: http://www.mdpi.com/1424-8220/12/8/10920

[9] S. L. Ceccio, "Friction drag reduction of external flows with bubble and gas injection," *Annual Review of Fluid Mechanics*, vol. 42, pp. 183–203, 2010.

[10] C. Rodrigues, C. Félix, A. Lage, and J. Figueiras, "Development of a long-term monitoring system based on fbg sensors applied to concrete bridges," *Engineering Structures*, vol. 32, no. 8, pp. 1993 – 2002, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141029610000866

[11] L. Troy M, G. Joseph T, and F. Daniel P, "How many electrodes are really needed for eeg-based mobile brain imaging?" *Journal of Behavioral and Brain Science*, vol. 2012, 2012.

[12] A. Saudabayev and H. A. Varol, "Sensors for robotic hands: A survey of state of the art," *IEEE Access*, vol. 3, pp. 1765–1782, 2015.

[13] A. Pantelopoulos and N. G. Bourbakis, "A survey on wearable sensor-based systems for health monitoring and prognosis," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 40, no. 1, pp. 1–12, 2010.

[14] D. Miorandi, S. Sicari, F. D. Pellegrini, and I. Chlamtac, "Internet of things: Vision, applications and research challenges," *Ad Hoc Networks*, vol. 10, no. 7, pp. 1497 – 1516, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1570870512000674

[15] J. Lifton, D. Seetharam, M. Broxton, and J. Paradiso, "Pushpin computing system overview: A platform for distributed, embedded, ubiquitous sensor networks," in *Pervasive Computing*. Springer, 2002, pp. 139–151.

[16] Z. Çelik-Butler, D. Butler, A. Yardanakul, and A. Yildiz, "Infrared sensors on flexible substrates for smart skin," in *SPIE Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Controls (AEROSENSE): Infrared Detectors and Focal Plane Arrays VII, Dereniak EL and Sampson RE (Eds), Orlando, FL, SPIE*, vol. 4721, 2002, pp. 260–268.

[17] J. Engel, J. Chen, C. Liu, B. R. Flachsbart, J. C. Selby, and M. A. Shannon, "Development of polyimide-based flexible tactile sensing skin," in *MRS Proceedings*, vol. 736. Cambridge Univ Press, 2002, pp. D4–5.

[18] S. Airbus, "Global market forecast 2016-2035," Technical report, Tech. Rep., 2016.

[19] S. Anders, W. Sellers, and A. Washburn, "Active flow control activities at nasa langley," in *2nd AIAA Flow Control Conference*, 2004, p. 2623.

[20] T. Washburn, "Airframe drag/weight reduction technologies," *Green Aviation Summit-Fuel Burn Reduction, NASA Ames Research Centre*, 2010.

[21] J. Reneaux *et al.*, "Overview on drag reduction technologies for civil transport aircraft," *ONERA: Tire a Part*, vol. 153, pp. 1–18, 2004.

[22] S. K. Robinson, "Coherent motions in the turbulent boundary layer," *Annual Review of Fluid Mechanics*, vol. 23, no. 1, pp. 601–639, 1991.

[23] W. K. Blake, *Mechanics of Flow-Induced Sound and Vibration V2: Complex Flow-Structure Interactions*. Elsevier, 2012, vol. 2.

[24] S. F. Hoerner, *Fluid-dynamic drag: practical information on aerodynamic drag and hydrodynamic resistance*. Hoerner Fluid Dynamics, 1965.

[25] C. B. da Silva and R. J. N. dos Reis, "The role of coherent vortices near the turbulent/non-turbulent interface in a planar jet," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 369, no. 1937, pp. 738–753, 2011.

[26] "Fly-by-feel systems represent the next revolution in aircraft controls," 2011.

[27] A. Berns and E. Obermeier, "Aeromems sensor arrays for time resolved wall pressure and wall shear stress measurements," in *Imaging Measurement Methods for Flow Analysis*. Springer, 2009, pp. 227–236.

[28] C. Bruinink, R. Jaganatharaja, M. De Boer, E. Berenschot, M. Kolster, T. Lammerink, R. Wiegerink, and G. Krijnen, "Advancements in technology and design of biomimetic flow-sensor arrays," in *Micro Electro Mechanical Systems, 2009. MEMS 2009. IEEE 22nd International Conference on*. IEEE, 2009, pp. 152–155.

[29] U. Buder, R. Petz, M. Kittel, W. Nitsche, and E. Obermeier, "Aeromems polyimide based wall double hot-wire sensors for flow separation detection," *Sensors and Actuators A: Physical*, vol. 142, no. 1, pp. 130–137, 2008.

[30] M. Watson, A. Jaworski, and N. Wood, "Application of synthetic jet actuators for the modification of the characteristics of separated shear layers on slender wings," *The Aeronautical Journal*, vol. 111, no. 1122, pp. 519–529, 2007.

[31] A. Sofla, S. Meguid, K. Tan, and W. Yeo, "Shape morphing of aircraft wing: Status and challenges," *Materials & Design*, vol. 31, no. 3, pp. 1284–1292, 2010.

[32] L. N. Cattafesta III and M. Sheplak, "Actuators for active flow control," *Annual Review of Fluid Mechanics*, vol. 43, pp. 247–272, 2011.

[33] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, "A network on chip architecture and design methodology," in *VLSI, 2002. IEEE Computer Society Annual Symposium on*. IEEE, 2002, pp. 105–112.

[34] S. Y. Kung, "Vlsi array processors," *Englewood Cliffs, NJ, Prentice Hall, 1988, 685 p. Research supported by the Semiconductor Research Corp., SDIO, NSF, and US Navy.*, vol. 1, 1988.

[35] H. T. Kung and P. L. Lehman, "Systolic (vlsi) arrays for relational database operations," in *Proceedings of the 1980 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '80. New York, NY, USA: ACM, 1980, pp. 105–116. [Online]. Available: http://doi.acm.org/10.1145/582250.582267

[36] E. Cloud, "The geometric arithmetic parallel processor," in *Frontiers of Massively Parallel Computation, 1988. Proceedings., 2nd Symposium on the Frontiers of*, Oct 1988, pp. 373–381.

[37] B. AMBA, "Arm announces amba 5 chi specification to enable high performance, highly scalable system on chip technology, http://www.arm.com/about/newsroom/arm-announces-amba-5-chi-specification-to-enable-high-performance-highly-scalable-system-on-chip.php."

[38] L. Benini and G. D. Micheli, "Networks on chips: a new soc paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, Jan 2002.

[39] G. D. Micheli and L. Benini, "On-chip communication architectures: System on chip interconnect," 2008.

[40] A. Olofsson, T. Nordström, and Z. Ul-Abdin, "Kickstarting high-performance energy-efficient manycore architectures with epiphany," in *2014 48th Asilomar Conference on Signals, Systems and Computers*, Nov 2014, pp. 1719–1726.

[41] A. Olofsson, "Epiphany-v: A 1024 processor 64-bit RISC system-on-chip," *CoRR*, vol. abs/1610.01832, 2016. [Online]. Available: http://arxiv.org/abs/1610.01832

[42] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. C. Miao, J. F. B. III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, no. 5, p. 15–31, Sep 2007.

[43] S.-C. C. C. Intel, "www.intel.com/content/www/us/en/research/intel-labs-single-chip-cloud-computer.html."

[44] M. Dehyadgari, M. Nickray, A. Afzali-Kusha, and Z. Navabi, "Evaluation of pseudo adaptive xy routing using an object oriented model for noc," in *Microelectronics, 2005. ICM 2005. The 17th International Conference on*. IEEE, 2005, pp. 5–pp.

[45] C. Bobda, A. Ahmadinia, M. Majer, J. Teich, S. Fekete, and J. van der Veen, "Dynoc: A dynamic infrastructure for communication in dynamically reconfugurable devices," in *Field Programmable Logic and Applications, 2005. International Conference on*. IEEE, 2005, pp. 153–158.

[46] H. Kariniemi and J. Nurmi, "Arbitration and routing schemes for on-chip packet networks," in *Interconnect-centric design for advanced SoC and NoC*. Springer, 2005, pp. 253–282.

[47] K. Kim, S.-J. Lee, K. Lee, and H.-J. Yoo, "An arbitration look-ahead scheme for reducing end-to-end latency in networks on chip," in *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*. IEEE, 2005, pp. 2357–2360.

[48] T. Bartic, J.-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins, "Topology adaptive network-on-chip design and implementation,"

IEE Proceedings-Computers and Digital Techniques, vol. 152, no. 4, pp. 467–472, 2005.

[49] S. Pasricha and N. Dutt, *On-chip communication architectures: system on chip interconnect*. Elsevier / Morgan Kaufmann Publishers.

[50] T. E. Anderson, D. E. Culler, and D. Patterson, "A case for now (networks of workstations)," *IEEE Micro*, vol. 15, no. 1, pp. 54–64, Feb 1995.

[51] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, "Myrinet: a gigabit-per-second local area network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, Feb 1995.

[52] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The illiac iv computer," *IEEE Transactions on Computers*, vol. C-17, no. 8, pp. 746–757, Aug 1968.

[53] K. T. H. Torleiv, *Algebraic Coding Theory*. American Cancer Society, 1999. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/047134608X.W4205

[54] K. Bolding, M. Fulgham, and L. Snyder, "The case for chaotic adaptive routing," *IEEE Transactions on Computers*, vol. 46, no. 12, pp. 1281–1292, Dec 1997.

[55] A. Borodin and J. Hopcroft, "Routing, merging, and sorting on parallel models of computation," *Journal of Computer and System Sciences*, vol. 30, no. 1, pp. 130 – 145, 1985. [Online]. Available: http://www.sciencedirect.com/science/article/pii/002200008590008X

[56] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[57] D. Wiklund and D. Liu, "Socbus: switched network on chip for hard real time embedded systems," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE, 2003, pp. 8–pp.

[58] L. M. Ni and P. K. McKinley, "A survey of wormhole routing techniques in direct networks," *Computer*, vol. 26, no. 2, pp. 62–76, 1993. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=191995

[59] K. Goossens, J. Dielissen, and A. Radulescu, "Aethereal network on chip: concepts, architectures, and implementations," *IEEE Design Test of Computers*, vol. 22, pp. 414–421.

[60] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Computing Surveys (CSUR)*, vol. 38, no. 1, p. 1, 2006.

[61] L. Löfdahl and M. Gad-el Hak, "Mems applications in turbulence and flow control," *Progress in Aerospace Sciences*, vol. 35, no. 2, pp. 101–203, 1999.

[62] A. Berns, U. Buder, E. Obermeier, A. Wolter, and A. Leder, "Aeromems sensor array for high-resolution wall pressure measurements," *Sensors and Actuators A: Physical*, vol. 132, no. 1, pp. 104–111, 2006.

[63] A. Berns, H.-D. Ngo, U. Buder, and E. Obermeier, "Aeromems pressure sensor array featuring through-wafer vias for high-resolution wall pressure measurements," in *Micro Electro Mechanical Systems, 2008. MEMS 2008. IEEE 21st International Conference on*. IEEE, 2008, pp. 896–899.

[64] A. Berns, U. Buder, E. Obermeier, X. Wang, J. Domhardt, J. Leuckert, and W. Nitsche, "Aeromems pressure sensor with integrated wall hot-wire," in *Sensors, 2008 IEEE*. IEEE, 2008, pp. 1560–1563.

[65] M. Schober, E. Obermeier, S. Pirskawetz, and H.-H. Fernholz, "A mems skin-friction sensor for time resolved measurements in separated flows," *Experiments in Fluids*, vol. 36, no. 4, pp. 593–599, 2004. [Online]. Available: http://dx.doi.org/10.1007/s00348-003-0728-4

[66] J. Engel, J. Chen, C. Liu, and D. Bullen, "Polyurethane rubber all-polymer artificial hair cell sensor," *Microelectromechanical Systems, Journal of*, vol. 15, no. 4, pp. 729–736, Aug 2006.

[67] N. Chen, C. Tucker, J. Engel, Y. Yang, S. Pandya, and C. Liu, "Design and characterization of artificial haircell sensor for flow sensing with ultrahigh velocity and angular sensitivity," *Microelectromechanical Systems, Journal of*, vol. 16, no. 5, pp. 999–1014, Oct 2007.

[68] X. CHEN, F. KOPSAFTOPOULOS, H. CAO, and F. CHANG, "Intelligent flight state identification of a self-sensing wing through neural network modelling," *Structural Health Monitoring 2017*, 2017.

[69] S. Lee, W. Buxton, and K. C. Smith, "A multi-touch three dimensional touch-sensitive tablet," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '85. New York, NY, USA: ACM, 1985, pp. 21–25. [Online]. Available: http://doi.acm.org/10.1145/317456.317461

[70] N. Salowitz, Z. Guo, S. J. Kim, Y. H. Li, G. Lanzara, and F. K. Chang, "Bio-inspired intelligent sensing materials for fly-by-feel autonomous vehicles," in *2012 IEEE Sensors*, Oct 2012, pp. 1–3.

[71] N. Salowitz, Z. Guo, Y.-H. Li, K. Kim, G. Lanzara, and F.-K. Chang, "Bio-inspired stretchable network-based intelligent composites," *Journal of Composite Materials*, vol. 47, no. 1, pp. 97–105, 2013.

[72] F. Gen-Kuong and B. Swanson, "Network sensor systems- the pressure belt application," in *19 th Aerospace Testing Seminar, Manhattan Beach, CA*, 2000, pp. 379–390.

[73] J. Lifton, M. Broxton, and J. A. Paradiso, "Experiences and directions in pushpin computing," in *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, ser. IPSN '05. Piscataway, NJ, USA: IEEE Press, 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1147685.1147753

[74] J. A. Paradiso, J. Lifton, and M. Broxton, "Sensate media—multimodal electronic skins as dense sensor networks," *BT Technology Journal*, vol. 22, no. 4, pp. 32–44, 2004.

[75] B. F. Mistree and J. A. Paradiso, "Chainmail: A configurable multimodal lining to enable sensate surfaces and interactive objects," in *Proceedings of the Fourth International Conference on Tangible, Embedded, and Embodied Interaction*, ser. TEI '10. New York, NY, USA: ACM, 2010, pp. 65–72. [Online]. Available: http://doi.acm.org/10.1145/1709886.1709899

[76] A. Dementyev, H.-L. C. Kao, and J. A. Paradiso, "Sensortape: Modular and programmable 3d-aware dense sensor network on a tape," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, ser. UIST '15. New York, NY, USA: ACM, 2015, pp. 649–658. [Online]. Available: http://doi.acm.org/10.1145/2807442.2807507

[77] F. Leens, "An introduction to i2c and spi protocols," *IEEE Instrumentation Measurement Magazine*, vol. 12, no. 1, pp. 8–13, February 2009.

[78] S. Rosset and H. R. Shea, "Flexible and stretchable electrodes for dielectric elastomer actuators," *Applied Physics A*, vol. 110, no. 2, pp. 281–307, Nov. 2012. [Online]. Available: http://link.springer.com/article/10.1007/s00339-012-7402-8

[79] S. Wagner, S. P. Lacour, J. Jones, P.-h. I. Hsu, J. C. Sturm, T. Li, and Z. Suo, "Electronic skin: architecture and components," *Physica E: Low-dimensional Systems and Nanostructures*, vol. 25, no. 2, pp. 326–334, 2004.

[80] M. L. Hammock, A. Chortos, B. C.-K. Tee, J. B.-H. Tok, and Z. Bao, "25th Anniversary Article: The Evolution of Electronic Skin (E-Skin): A Brief History, Design Considerations, and Recent Progress," *Advanced Materials*, vol. 25, no. 42, pp. 5997–6038, Nov. 2013, 00091. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/adma.201302240/abstract

[81] Y. Ohmura, Y. Kuniyoshi, and A. Nagakubo, "Conformable and scalable tactile sensor skin for curved surfaces," in *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*. IEEE, 2006, pp. 1348–1353.

[82] T. Sekitani, Y. Noguchi, K. Hata, T. Fukushima, T. Aida, and T. Someya, "A Rubberlike Stretchable Active Matrix Using Elastic Conductors," *Science*, vol. 321, no. 5895, pp. 1468–1472, Sep. 2008, 00499. [Online]. Available: http://www.sciencemag.org/content/321/5895/1468

[83] R. Dahiya, G. Metta, M. Valle, and G. Sandini, "Tactile Sensing - From Humans to Humanoids," *IEEE Transactions on Robotics*, vol. 26, no. 1, pp. 1–20, Feb. 2010, 00377.

[84] J. Ohta, T. Tokuda, K. Kagawa, S. Sugitani, M. Taniyama, A. Uehara, Y. Terasawa, K. Nakauchi, T. Fujikado, and Y. Tano, "Laboratory investigation of microelectronics-based stimulators for large-scale suprachoroidal transretinal stimulation (sts)," *Journal of neural engineering*, vol. 4, no. 1, p. S85, 2007.

[85] M. Monge, M. Raj, M. Honarvar-Nazari, H.-C. Chang, Y. Zhao, J. Weiland, M. Humayun, Y.-C. Tai, and A. Emami-Neyestanak, "A fully intraocular 0.0169mm2/pixel 512-channel self-calibrating epiretinal prosthesis in 65nm CMOS," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2013 IEEE International*, Feb. 2013, pp. 296–297, 00000.

[86] A. Rabbi, K. Ivanca, A. Putnam, A. Musa, C. Thaden, and R. Fazel-Rezai, "Human performance evaluation based on EEG signal analysis: A prospective review," in *Annual International Conference of the IEEE Engineering in Medicine and Biology Society, 2009. EMBC 2009*, Sep. 2009, pp. 1879–1882.

[87] M. Hassaballah, A. Ali, and H. Alshazly, *Image Features Detection, Description and Matching*, 02 2016, vol. 630, pp. 11–45.

[88] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc, "Feature tracking and matching in video using programmable graphics hardware," *Machine Vision and Applications*, vol. 22, no. 1, p. 207–217, Jan 2011. [Online]. Available: https://link.springer.com/article/10.1007/s00138-007-0105-z

[89] C. Tomasi and T. Kanade, "Detection and tracking of point features," 1991.

[90] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, Nov 2004. [Online]. Available: https://doi.org/10.1023/B:VISI.0000029664.99615.94

[91] J. A. Ross, D. A. Richie, and S. J. Park, "Implementing image processing algorithms for the epiphany many-core coprocessor with threaded mpi," in *IEEE, September*, 2015.

[92] C. Y. Villalpando, A. E. Johnson, R. Some, J. Oberlin, and S. Goldberg, "Investigation of the tilera processor for real time hazard detection and avoidance on the altair lunar lander," in *Aerospace Conference, 2010 IEEE*. IEEE, 2010, p. 1–9.

[93] S. Srinivasan, S. Dattagupta, P. Kulkarni, and K. Ramamritham, "A survey of sensory data boundary estimation, covering and tracking techniques using collaborating sensors," *Pervasive and Mobile Computing*, vol. 8, no. 3, p. 358–375, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574119212000430

[94] K. Chintalapudi and R. Govindan, *Localized edge detection in sensor fields*. IEEE, May 2003, p. 59–70. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1203357

[95] M. Li and Y. Liu, "Iso-map: Energy-efficient contour mapping in wireless sensor networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 5, p. 699–710, May 2010.

[96] S. Duttagupta, K. Ramamritham, and P. Ramanathan, "Distributed boundary estimation using sensor networks," in *Mobile Adhoc and Sensor Systems (MASS), 2006 IEEE International Conference on*. IEEE, 2006, pp. 316–325.

[97] X. Meng, T. Nandagopal, L. Li, and S. Lu, "Contour maps: Monitoring and diagnosis in sensor networks," *Computer Networks*, vol. 50, no. 15, p. 2820–2838, Oct 2006. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1389128605003841

[98] Y. Xu, W.-C. Lee, and G. Mitchell, *CME: A Contour Mapping Engine in Wireless Sensor Networks*. IEEE, Jun 2008, p. 133–140. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4595877

[99] W. Xue, Q. Luo, L. Chen, and Y. Liu, *Contour Map Matching for Event Detection in Sensor Networks*, 2006.

[100] K. King and S. Nittel, *Efficient Data Collection and Event Boundary Detection in Wireless Sensor Networks Using Tiny Models*, 2010, p. 100–114.

[101] P.-K. Liao, M.-K. Chang, and C.-C. J. Kuo, *A Cross-Layer Approach to Contour Nodes Inference with Data Fusion in Wireless Sensor Networks*. IEEE, 2007, p. 2773–2777. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4224760

[102] N. Pereira, B. Andersson, and E. Tovar, "Widom: A dominance protocol for wireless medium access," *IEEE Transactions on Industrial Informatics*, vol. 3, no. 2, pp. 120–130, May 2007.

[103] B. Andersson, N. Pereira, W. Elmenreich, E. Tovar, F. Pacheco, and N. Cruz, "A scalable and efficient approach for obtaining measurements in can-based control systems," *IEEE Transactions on Industrial Informatics*, vol. 4, no. 2, pp. 80–91, May 2008.

[104] M. Vahabi, V. Gupta, M. Albano, R. Rangarajan, and E. Tovar, "Feature extraction in densely sensed environments: Extensions to multiple broadcast domains," *International Journal of Distributed Sensor Networks*, vol. 11, no. 8, p. 457537, 2015. [Online]. Available: https://doi.org/10.1155/2015/457537

[105] Z. Shi and A. Burns, "Real-time communication analysis for on-chip networks with wormhole switching," in *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, ser. NOCS '08. Washington,

DC, USA: IEEE Computer Society, 2008, pp. 161–170. [Online]. Available: http://dl.acm.org/citation.cfm?id=1397757.1397996

[106] L. S. Indrusiak, "End-to-end schedulability tests for multiprocessor embedded systems based on networks-on-chip with priority-preemptive arbitration," *Journal of Systems Architecture*, vol. 60, no. 7, pp. 553 – 561, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762114000800

[107] D. Rahmati, S. Murali, L. Benini, F. Angiolini, G. D. Micheli, and H. Sarbazi-Azad, "Computing accurate performance bounds for best effort networks-on-chip," *IEEE Transactions on Computers*, vol. 62, no. 3, pp. 452–467, March 2013.

[108] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, "Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 2. IEEE, 2004, pp. 890–895.

[109] T. Bjerregaard and J. Sparso, "A router architecture for connection-oriented service guarantees in the mango clockless network-on-chip," in *Design, Automation and Test in Europe, 2005. Proceedings*, pp. 1226–1231 Vol. 2.

[110] W.-D. Weber, J. Chou, I. Swarbrick, and D. Wingard, "A quality-of-service mechanism for interconnection networks in system-on-chips," in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, ser. DATE '05. IEEE Computer Society, 2005, p. 1232–1237. [Online]. Available: http://dx.doi.org/10.1109/DATE.2005.33

[111] J. Liang, A. Laffely, S. Srinivasan, and R. Tessier, "An architecture and compiler for scalable on-chip communication," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 7, p. 711–726, Jul 2004.

[112] J. Liu, L.-R. Zheng, and H. Tenhunen, "Interconnect intellectual property for network-on-chip (noc)," *Journal of Systems Architecture*, vol. 50, no. 2, p. 65–79, Feb 2004.

[113] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, and et al., "The raw microprocessor: a computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, p. 25–35, Mar 2002.

[114] A. Leroy, P. Marchal, A. Shickova, F. Catthoor, F. Robert, and D. Verkest, "Spatial division multiplexing: A novel approach for guaranteed throughput on nocs," in *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '05. ACM, 2005, p. 81–86. [Online]. Available: http://doi.acm.org/10.1145/1084834.1084858

[115] R. L. Cruz, "A calculus for network delay. i. network elements in isolation," *IEEE Transactions on Information Theory*, vol. 37, no. 1, pp. 114–131, Jan 1991.

[116] M. Fidler, "Survey of deterministic and stochastic service curve models in the network calculus," *IEEE Communications Surveys Tutorials*, vol. 12, no. 1, pp. 59–86, First 2010.

[117] X. Fan, M. Jonsson, and J. Jonsson, "Guaranteed real-time communication in packet-switched networks with fcfs queuing," *Computer Networks*, vol. 53, no. 3, pp. 400 – 417, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128608003538

[118] V. Sivaraman, F. M. Chiussi, and M. Gerla, "Deterministic end-to-end delay guarantees with rate controlled {EDF} scheduling," *Performance Evaluation*, vol. 63, no. 4–5, pp. 509 – 519, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0166531605000507

[119] S. Manolache, P. Eles, and Z. Peng, "Buffer space optimisation with communication synthesis and traffic shaping for nocs," in *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings*, ser. DATE '06. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 718–723. [Online]. Available: http://dl.acm.org/citation.cfm?id=1131481.1131683

[120] J. Specht and S. Samii, "Urgency-based scheduler for time-sensitive switched ethernet networks," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2016, pp. 75–85.

[121] D. Sigüenza-Tortosa, T. Ahonen, and J. Nurmi, "Issues in the development of a practical noc: the proteo concept," *Integration, the VLSI Journal*, vol. 38, no. 1, p. 95–105, Oct 2004.

[122] D. Andreasson and S. Kumar, "Slack-time aware routing in noc systems," in *2005 IEEE International Symposium on Circuits and Systems*, May 2005, pp. 2353–2356 Vol. 3.

[123] Institute of Electrical and Electronics Engineers, *Computer Architecture, 1997. Conference Proceedings. The 24th Annual International Symposium on.*

[124] S. Pasricha and N. Dutt, *On-chip communication architectures: system on chip interconnect*. Morgan Kaufmann, 2010.

[125] J. Hu and R. Marculescu, "Energy-aware mapping for tile-based noc architectures under performance constraints," in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '03. New York, NY, USA: ACM, 2003, pp. 233–239. [Online]. Available: http://doi.acm.org/10.1145/1119772.1119818

[126] J. Loureiro. [Online]. Available: https://github.com/joaofl/noc

[127] G. Riley and T. Henderson, "The ns-3 network simulator," in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Güneş, and J. Gross,

Eds. Springer Berlin Heidelberg, 2010, pp. 15–34. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12331-3_2

[128] G. Haller and G. Yuan, "Lagrangian coherent structures and mixing in two-dimensional turbulence," *Physica D: Nonlinear Phenomena*, vol. 147, no. 3, pp. 352–370, 2000.

[129] V. Schmitt and F. Charpin, "Pressure distributions on the onera-m6-wing at transonic mach numbers," *Experimental data base for computer program assessment*, vol. 4, 1979.

[130] F. Palacios, J. Alonso, K. Duraisamy, M. Colonno, J. Hicken, A. Aranake, A. Campos, S. Copeland, T. Economon, A. Lonkar *et al.*, "Stanford university unstructured (su 2): An open-source integrated computational environment for multi-physics simulation and design," in *51st AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, 2013, p. 287.

[131] B. Krishnamachari, D. Estrin, and S. Wicker, "The impact of data aggregation in wireless sensor networks," in *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, 2002, pp. 575–578.

[132] B. N. Datta, *Numerical linear algebra and applications*. Siam, 2010.

[133] O. Vincent and O. Folorunso, "A descriptive algorithm for sobel image edge detection," in *Proceedings of Informing Science & IT Education Conference (InSITE)*, 2009, pp. 97–107.

[134] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, "A survey on parallel computing and its applications in data-parallel problems using gpu architectures," *Communications in Computational Physics*, vol. 15, no. 2, p. 285–329, 2014.

[135] A. Agarwal, C. Iskander, and R. Shankar, "Survey of network on chip (noc) architectures & contributions," *Journal of engineering, Computing and Architecture*, vol. 3, no. 1, pp. 21–27, 2009.

[136] W. J. Dally and J. W. Poulton, *Digital systems engineering*. Cambridge university press, 2008. [Online]. Available: https://www.google.com/books?hl=pt-PT&lr=&id=anC_AwAAQBAJ&oi=fnd&pg=PR21&dq=Digital+Systems+Engineering&ots=ED8vA_i6yp&sig=2GFQgh7JSzE40zdyKEdSfaQWhXc

[137] L. E. Frenzel, *Handbook of serial communications interfaces: a comprehensive compendium of serial digital input/output (I/O) standards*. Newnes, 2015.

[138] N. Patel, V. Patel, and V. Patel, "Vhdl implementation of uart with status register," in *Communication Systems and Network Technologies (CSNT), 2012 International Conference on*. IEEE, 2012, pp. 750–754.

[139] Xilinx, "Spartan-6 family overview," October 2011, product Specification. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds160.pdf

[140] Atmel, "Atmel smart arm-based mcu datasheet," March 2015, product Specification. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11158-32-bit%20Cortex-M4-Microcontroller-SAM4N16-SAM4N8_Datasheet.pdf

[141] R. Barry *et al.*, "Freertos," *Internet, Oct*, 2008.

[142] Z. Hameed, Y. Hong, Y. Cho, S. Ahn, and C. Song, "Condition monitoring and fault detection of wind turbines and related algorithms: A review," *Renewable and Sustainable energy reviews*, vol. 13, no. 1, pp. 1–39, 2009.

[143] D. Freedman and P. Diaconis, "On the histogram as a density estimator: L 2 theory," *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, vol. 57, no. 4, pp. 453–476, 1981.

[144] J. Loureiro, P. Santos, R. Rangarajan, and E. Tovar, "Simulation module and tools for xdense sensor network," in *Proceedings of the Workshop on Ns-3*, ser. WNS3 '17. New York, NY, USA: ACM, 2017, pp. 110–117. [Online]. Available: http://doi.acm.org/10.1145/3067665.3067680