COPYRIGHT AND USE OF THIS THESIS

This thesis must be used in accordance with the provisions of the Copyright Act 1968.

Reproduction of material protected by copyright may be an infringement of copyright and copyright owners may be entitled to take legal action against persons who infringe their copyright.

Section 51 (2) of the Copyright Act permits an authorized officer of a university library or archives to provide a copy (by communication or otherwise) of an unpublished thesis kept in the library or archives, to a person who satisfies the authorized officer that he or she requires the reproduction for the purposes of research or study.

The Copyright Act grants the creator of a work a number of moral rights, specifically the right of attribution, the right against false attribution and the right of integrity.

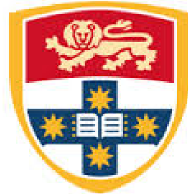
You may infringe the author's moral rights if you:

- fail to acknowledge the author of this thesis if you quote sections from the work
- attribute this thesis to another author
- subject this thesis to derogatory treatment which may prejudice the author's reputation

For further information contact the University's Director of Copyright Services

sydney.edu.au/copyright

CURRACURRONG:
A STREAM PROCESSING SYSTEM FOR DISTRIBUTED ENVIRONMENTS



THE UNIVERSITY OF
SYDNEY

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy in the School of Information Technologies at The University of Sydney

Vasvi Kakkad

March 2015

Abstract

Advances in technology have given rise to applications that are deployed on wireless sensor networks (WSNs), the cloud, and the Internet of things. There are many emerging applications, some of which include sensor-based monitoring, web traffic processing, and network monitoring. These applications collect large amount of data as an unbounded sequence of events and process them to generate a new sequences of events. Such applications need an adequate programming model that can process large amount of data with minimal latency; for this purpose, stream programming, among other paradigms, is ideal. However, stream programming needs to be adapted to meet the challenges inherent in running it in distributed environments. These challenges include the need for modern domain specific language (DSL), the placement of computations in the network to minimise energy costs, and timeliness in real-time applications.

To overcome these challenges we developed a stream programming model that achieves easy-to-use programming interface, energy-efficient actor placement, and timeliness. This thesis presents Curracurrong, a stream data processing system for distributed environments. In Curracurrong, a query is represented as a stream graph of stream operators and communication channels. Curracurrong provides an extensible stream operator library and adapts to a wide range of applications. It uses an energy-efficient placement algorithm that optimises communication and computation. We extend the placement problem to support dynamically changing networks, and develop a dynamic program with polynomially bounded runtime to solve the placement problem. In many stream-based applications, real-time data processing is essential. We propose an approach that measures time delays in stream query processing; this model measures the total computational time from input to output of a query, i.e., end-to-end delay.

Acknowledgement

It would not have been possible to write this doctoral thesis without the help and support of the kind people around me, to only some of whom it is possible to give particular mention here.

Above all, I would like to thank my husband Jignesh for his love, emotional support and encouragement. My parents, grandparents and brother have given me their unequivocal support throughout and have believed in me, as always, for which my mere expression of thanks does not suffice.

I want to express my deep gratitude to Dr Bernhard Scholz, my advisor, for his excellent support, his advice and guidance during this research. Through my journey of the PhD, I was able to gain a lot of valuable experience. I cannot thank him enough for his direction. I would also like to thank Dr Andrew Santosa for helping and giving pace to this research. The good advice and support from Andrew, has been invaluable on my technical writing skill, for which I am extremely grateful. I would like to acknowledge Prof. Alan Fekete for his advice during this research.

My sincere thanks to Cristina Cifuentes, research director at Oracle Labs in Brisbane, Australia, for giving me an opportunity to work with brilliant people and supervising my work on an exciting project during my internship.

I would like to acknowledge the academic, technical and administrative staff of the University of Sydney for providing necessary support for this research. I would like to thank my fellow postgraduate research students in the School of Information Technologies.

I thank my friends in India, Australia and elsewhere for their support and encouragement throughout.

Contents

Abstract ii

Acknowledgement iii

List of Tables vii

List of Figures x

1 Introduction 1

1.1 Stream Data Processing 3
1.2 Wireless Sensor Network 6
1.3 Cloud Network 10
1.4 Problem Definition 12
1.5 Contribution 14
1.6 Publications 15
1.7 Organisation of the Thesis 16

2 Curracurrong for WSN 17

2.1 Curracurrong Query Language 19
2.1.1 Language Syntax 21
2.1.2 Case Study: Seismic Event Detection 23
2.2 Query Processing System 27
2.2.1 Server Module 30
2.2.2 Runtime Environment 31
2.3 Operator Placement and Deployment 33
2.3.1 Data Type and Bandwidth Computation 33
2.3.2 Placement of Operators 35

2.4 Experiments 39
2.4.1 In-Network vs. Forwarding 40
2.4.2 Analytical Model 42
2.4.3 Heuristic Efficiency 45
2.4.3.1 AMPL Program 45
2.4.4 Computation Resource Usage 46
2.4.5 Cost Distribution 48
2.5 Chapter Summary 49

3 Curracurrong Cloud **50**
3.1 Curracurrong Cloud 52
3.1.1 ZigBee vs. UDP 52
3.1.2 Broadcast and Multicast 53
3.1.3 Scheduler 54
3.1.4 Operators 55
3.2 Evaluation 56
3.2.1 Operator Placement Algorithm Efficiency 56
3.2.2 Measuring Message Latency 58
3.2.3 Time-triggered vs. Event-triggered 59
3.3 Chapter Summary 62

4 Migrating Operator Placement **63**
4.1 Migrating Operator Placement Problem 66
4.2 Graph Composites 69
4.3 Dynamic Programming for MOPP 71
4.3.1 Dynamic Program 71
4.3.2 Memo Table 74
4.3.3 Complexity Bounds 76
4.3.4 Mitigating Complexity by Locality 79
4.4 Experimental Evaluation 80
4.4.1 ILP vs. Dynamic Programming 81
4.4.2 In-network vs. Data Forwarding 82
4.4.3 Locality of Placement 82
4.5 Chapter Summary 84

5 Timeliness in Curracurrong **85**
5.1 System Model and Problem Definition . . . 87
5.1.1 Problem Definition . . . 89
5.2 Semantics for Stream Graph Composites . . . 89
5.2.1 Semantics . . . 90
5.3 Algorithm . . . 96
5.3.1 Finding Time Steady State . . . 96
5.3.1.1 Computation of Repetition Vector . . . 97
5.3.1.2 Computation of Periodicity Vector . . . 99
5.3.2 Simulation of Time Steady State . . . 102
5.4 Experiments . . . 107
5.4.1 Computing Time Steady State . . . 108
5.4.2 Periodicity vs. Time Steady Sate . . . 110
5.4.3 Filter Data Rate vs. Time Steady State . . . 110
5.4.4 Measuring Delay in the Stream Graph . . . 111
5.4.5 Effect of Periodicity Scaling . . . 113
5.5 Chapter Summary . . . 115

6 Related Work **116**
6.1 Stream Processing . . . 116
6.1.1 Stream Processing Languages . . . 118
6.1.2 Stream Processing Engines . . . 133
6.2 Stream Data Processing in WSN . . . 137
6.2.1 Productivity . . . 137
6.2.2 Flexibility . . . 138
6.2.3 Efficiency . . . 139
6.3 Stream Data Processing in Cloud . . . 140
6.4 Real-time Stream processing . . . 142

7 Conclusion and Future Work **145**
7.1 Opportunities for Future Work . . . 148

Bibliography **150**

List of Tables

1.1 Comparison between selected commercial motes 7
1.2 Benefits of cloud services 11

2.1 Local cost for operators on sensor nodes (fig. 2.13), here ∞ indicates very high cost; the cost for each operator on the base station is same except for sense and sink operators, which have fixed placement 38
2.2 GLPK running time on `Average` instances 44

4.1 GLPK and dynamic program running times 81
4.2 Optimal in-network vs. data forwarding 82

5.1 Repetition and periodicity vectors for the example stream graph in Figure 5.1 (a). Here, sync is the case when all sensors are synchronised, whereas unsync represents unsynchronised sensors. 102
5.2 Delays with synchronised sensors (cf. Figure 5.1(b)) 107
5.3 Delays with unsynchronised sensors (cf. Figure 5.6) 107
5.4 Comparison of time steady state for the stream graphs with sensors having same periodicity (Tss-uniform), random periodicity (Tss-random), and periodicity prime (Tss-prime). 108
5.5 Time steady state for the real-world stream processing examples 109

6.1 Comparison of Curracurrong with related systems 143

List of Figures

1.1 History of programming models and languages 3

1.2 (a) Synchronous block (b) Synchronous data flow graph 5

2.1 Curracurrong stream operators 20

2.2 `Average` query 22

2.3 Curracurrong query language BNF 24

2.4 Seismic event detection – dataflow graph 24

2.5 `Seismic event` – Curracurrong query 25

2.6 Seismic event detection – Curracurrong stream graph 25

2.7 Borealis query definition – Event.xml (Part-1) 28

2.8 Borealis query definition – Event.xml (Part-2) 29

2.9 Curracurrong system components. Here, S indicates the server module and RE is the runtime environment. The broken lines are administrative channels and bold lines are data channels. 30

2.10 `Average` stream graph. The graph is generated by the server module from the query of Figure 2.2, by converting "`->`" into a FIFO channel and "`|`" into a join operator with the corresponding channels. 31

2.11 Runtime environment components 33

2.12 Topological sort-based graph walk. Here $T(c)(I_v)$ and $B(c)(I_v)$ are respectively the functions that computes the output channel c's type and bandwidth given input channels I_v. 34

2.13 `Average` data type, bandwidth, and placement. The number in boxes denote topological ordering, the numbers attached to edges denote bandwidth requirement, DRecord and DInteger32 are channel data types. The record types (DRecord) have three elements of the primitive 32-bit integer types (DInteger32). 35

2.14 Isolating cut heuristic algorithm 38
2.15 Single-hop (a) vs. multi-hop topology (b). For single-hop, a sensor node N_1 communicates directly to the base station, whereas for multi-hop, the data goes through N_2–N_5 before reaching the base station. . . . 42
2.16 Experimental results. The maximum reduction of energy usage for single-hop topology is 5–7%, and for multi-hop topology is 22–24%. . . 42
2.17 Analytical results. The maximum reduction in energy usage is approximately 5% for single-hop, and $45-46\%$ for multi-hop topology. 43
2.18 Scaled analytical results 43
2.19 AMPL program for multiway cut problem 45
2.20 Number of operators placed in the network (on the sensor nodes excluding the base station) 47
2.21 Cost distribution over query branches 48

3.1 Network data transfer with operator placement 57
3.2 Placement algorithm efficiency 58
3.3 Effect on latency while varying the network load 59
3.4 Effect of the sense time interval setting on time-triggered Sensors 60
3.5 Network data with time-triggered sensors 61
3.6 Network data with event-triggered sensors 62

4.1 Running example. (a) Stream graph G^{pre} for queries Q_1 and Q_2 with placement p^{pre}; (b) Stream graph G for query Q_1, newly added query Q_3, deleted query Q_2, and new placement p. 64
4.2 Incurred costs in operator placement 65
4.3 Composites 70
4.4 Memo tables for operators and composites (see Fig. 4.1(b)) 75
4.5 Sub-network two hops away from base station 79
4.6 Locality heuristic vs. optimal in-Network 83
4.7 Locality heuristic vs. data forwarding 84

5.1 Stream graph and event causality with synchronised sensors 86
5.2 Composites 90
5.3 Example of a compositional stream graph using stream and source composites 91

5.4 Denotational semantics for stream composites. Given an operator u, c_u is the value of $c(v,u)$, p_u is the value of $p(u,v)$, and r_u is the repetition for the operator u (Section 5.3). Given a filter f, $\hat{f}$ is the function implemented by the filter, which takes input data tokens as arguments and produces a sequence of output data tokens, and w_f is the computation time. Operator $\oplus$ is as for Definition 5. 93
5.5 Relationship between tokens consumed and produced 94
5.6 Event causality with the unsynchronised sensors from Figure 5.1(a) . . 100
5.7 Abstract semantics of stream composites 103
5.8 Sensor periodicity vs. time steady state 110
5.9 Filter data-rate vs. time steady state 111
5.10 Sensor periodicity vs. time delay 112
5.11 Filter data-rate vs. time delays 112
5.12 Percentage periodicity scaling vs. time steady state 113
5.13 Percentage periodicity scaling vs. time delay 114

6.1 A Lucid example – `prime` 119
6.2 A VAL example – `stats` 120
6.3 A SISAL example – `Integrate` 121
6.4 A Lustre example – `ABRO` 122
6.5 A Signal example – ADD 122
6.6 A Brook example – `add` 124
6.7 A OpenCL example – `VectorAdd` 126
6.8 A OpenCL example – `VectorAdd` continued 127
6.9 A CUDA example – `gpuVecAdd` 129
6.10 Stream graph for VectorAdd StreamIt program 130
6.11 A StreamIt example – `VectorAdd` 131
6.12 Stream graph for NumberedCat SPL program 131
6.13 A SPL example – `NumberedCat` 132

Chapter 1

Introduction

Emerging applications for contemporary distributed computing technologies, including wireless sensor networks (WSN) and cloud computing, have renewed interest in stream programming. The applications range from disaster area monitoring [1], [2], health care monitoring [3], financial data tracking [4], to network traffic monitoring [5]. These applications process continuous unbounded streams of information. WSN applications comprise a multitude of sensors that generate streams of data. Sensors are tiny devices deployed at distant locations that monitor and generate stream comprising data like temperature, pressure, and gas readings. Cloud computing, also known as utility computing, is another distributed environment. Cloud computing provides various services like storage, database, and software tools; cloud application developers can subscribe to these services and pay for their usage. Applications such as financial data tracking and network data analysis are built in the cloud environment, where the data stream is collected from the cloud nodes and over the cloud network, respectively. The stream applications for WSN and cloud have a broad spectrum. We briefly described some of them below.

One of the important streaming applications include *early bushfire detection* [2]. Bushfire is a frequent event in Australia. Each year, they impact extensive areas causing property damage and loss of human life. One of the most intense and deadly bushfire occurred in 2009, in which 173 people lost their lives [6]. Researchers from the Bushfire Cooperative Research Centre (Bushfire CRC) [7], University of Technology, Sydney, and the University of Western Australia [8] have developed an early detection WSN applications that continuously measure temperature, moisture in the air, wind speed and direction, and the presence of certain chemical particles. These applications collect

data streams and generate an alert when a bushfire threat is encountered.

Health monitoring is another WSN streaming application [9, 10]. With advances in intelligent wireless sensors and embedded computing technologies, miniaturised pervasive health monitoring devices have become practically feasible. These sensors continuously monitor and analyse the stream of physiological parameters like heart rate, body position, and snoring. A supervisory medical worker accesses the database to monitor patients and the system alerts other personnel in case of a health emergency. These types of WSN applications can provide new rehabilitation therapies while patients remain in their home environment [10].

Another WSN streaming application includes *volcanic event detection* [1, 11]. To prevent loss of life and property due to such disastrous events, WSN applications are now possible as early warning event detection systems. Researchers from Harvard University deployed 16 Tmote Sky sensors in the vicinity of a volcano. The sensors continuously captured data including acoustic, seismic events, and discharged gases over a specific time interval. The data are evaluated and if the measurements exceed a threshold then an alert is transmitted via a long distance FreeWave radio modem.

Another streaming example is *network traffic monitoring* application for intrusion detection [5]. An internet service provider can sustain traffic volume from some connections in the range of tens of gigabytes per second. For an application that analyses network traffic to detect possible intrusion threats requires high processing capacity for two reasons. First, the application needs to analyse huge traffic volumes that must be processed in real time to block possible harmful events. Second, the type of computations used to process the data usually require non-trivial aggregation and comparison of online and historic data.

Monitoring the cloud is a task of paramount importance for both cloud providers and consumers. This streaming application continuously captures data such as CPU and network bandwidth usage, and number of reads and writes, across all nodes in a cloud cluster and use them to generate alerts when particular system parameters reach a threshold. There are widely spread commercial and open-source platforms for cloud monitoring as well as services that can help cloud user to assess the performance and reliability of cloud services. Some of such monitoring systems include CloudWatch [12], AzureWatch [13], and Nagios [14].

Another scenario involving online credit card and debit card transactions analysis can be drawn from *financial data analysis* and *fraud detection* applications that demand

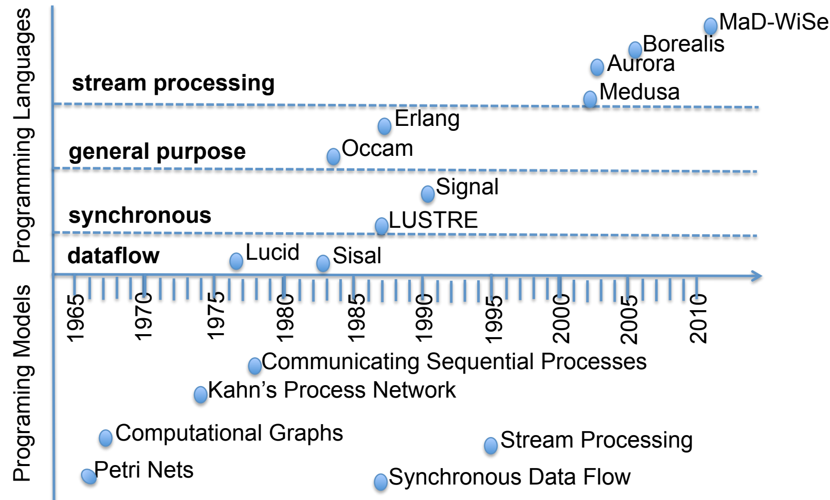


Figure 1.1: History of programming models and languages

high processing capacity [4]. It is important for such applications to provide low latency guarantee, since they must comply with strict time limits.

The above applications emphasise the need of stream programming in real-world applications, where a continuous stream of data is collected over an interval of time, processed, and forwarded to a sink. Understanding these data streams is crucial for managing and troubleshooting a large network. Apart from a complex real-time streaming applications, distributed environments like WSN and the cloud have their own set of challenges and limitations. A detailed explanation of two distributed technologies and the challenges associated with them are given in the following section.

1.1 Stream Data Processing

The concept of a stream programming has a long history in computer science [15]. Figure 1.1 presents the timeline of various programming paradigms over the timeline. The idea of stream processing stems from fundamental data flow models like the *Kahn process model* (KPN) [16] and *synchronous data flow* (SDF) [17].

The Kahn process model is a programming paradigm suitable for streaming-based multimedia and signal-processing applications. KPN is a simple model where an application is modelled as a collection of concurrent autonomous processes communicating

streams of data through FIFO channels. In a network, a process reads from input channels and queues items in their output channels. The network blocks the operation until data has arrived in the input channel. If each process in the network works deterministically, the entire network is deterministic; but it is not possible to statically determine the amount of buffering required for each channel to avoid deadlock.

Synchronous data flow is a restricted form of KPN, where processes execute atomically. Unlike KPN, in SDF the number of data items read and produced by process is known at compile time. Because of deterministic processing it is possible to statically check the network processing for deadlocks. The amount of buffering required in an application can also be determined statically, which helps in determining potential scheduling and optimisation for computation. In data flow, a program is divided into pieces, known as nodes or blocks, which can execute (fire) whenever input data are available. An application in SDF is described as a SDF graph – a directed graph with nodes indicating computational block and edges representing communicating data paths between nodes. Figure 1.2 (a) shows a synchronous data block, including numbers (referred as a, b, and c) associated with consumed inputs and produced output data tokens. A SDF is a network of synchronous blocks (Figure 1.2 (b)). SDF graphs are closely related to computation graphs [18], where each input to a block has two numbers associated with it, a threshold and the number of data tokens consumed. The threshold indicates the number of data tokens required to invoke the block, and could be different from the number of tokens consumed by the block. Drawing on all of the fundamental models, the main features of stream programming model are listed as follows.

- continuous unbounded incoming data,
- volatile data streams,
- sequential data access,
- real-time requirement, and
- limited main memory to buffer data.

As shown in Figure 1.1, the notion of streams has produced several programming paradigms. These include dataflow languages such as Lucid [19] and Sisal [20], which use a demand-driven model for data computation. In this model, each statement can be

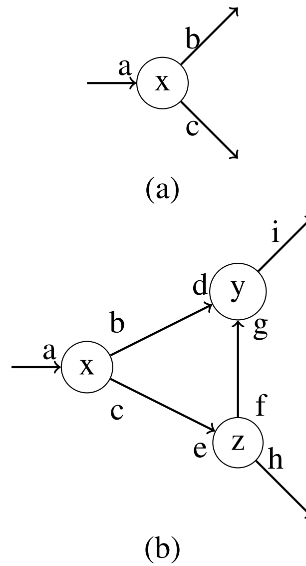


Figure 1.2: (a) Synchronous block (b) Synchronous data flow graph

understood as an equation defining a network of processors and communication lines between them, through which data flows. Each variable is an infinite stream of values and every function is a filter or a transformer. Other paradigms include synchronous languages such as LUSTRE [21], Esterel [22], and Signal [23]. LUSTRE and Esterel were designed for programming reactive systems such as automatic control and monitoring systems. Their synchronous nature is similar to temporal logics, and therefore makes it well suited for handling time in programs. Likewise Signal is another synchronous data flow language designed for domain-specific signal-processing applications. The formal model of Signal describes a system with several clocks. Among general-purpose languages are Occam [24] and Erlang [25]. The domain-specific languages (DSLs) for distributed environments require the notion of communication between nodes, with which users could build an entire system. Existing generalised stream processing systems include Medusa [26], Aurora [27], Borealis [28], and STREAM [29]. The recently-developed MaD-WiSe [30] is another distributed stream management for WSN. However, these approaches have several restrictions. Generalised stream-processing engines like Borealis require users to write an XML file for a query as well as for deployment of operators in the network. MaD-WiSe uses SQL-based queries, which limits stream management to aggregation and does not support flexible application design.

Likewise, a number of recent systems have been developed for the cloud environment. Twitter's Storm system [31] and LinkedIn's Samza [32] are both intended to co-exist with a Hadoop ecosystem. Yahoo! Research has proposed S4 [33], and UC Berkeley offers Spark Streaming [34]. These existing streaming platforms have many powerful features. In the cloud, some of them require the developer to write an application with lots of code in Java or a similar language. Systems like Storm require explicit deployment; the developer needs to identify which node will perform each aspect of the computation. To develop a domain-specific stream processing system, it is important to understand the domain and the challenges associated with it. This thesis has developed stream programming model for distributed environment, after detailed analysis of the features and challenges associated with it.

1.2 Wireless Sensor Network

Advances in wireless communication technologies have enabled the development of small, low-cost and low-power multi-functional sensor nodes that can sense the environment, process the data, and communicate with each other over a short range. A sensor node consists of five basic units, for processing, memory, sensing, power, and communication. A processing unit is responsible for executing the set of routines that form a sensor's task. A memory unit consists of three parts. First, the program flash memory used by the processing unit as a temporary storage area to execute routines. Second, measurement flash memory to store sensory measurements obtained by the sensing unit. Third, an EEPROM configuration where all configuration data for a sensor node are kept.

A sensing unit includes different kinds of sensors to measure such factors as temperature, light, and humidity, depending on the application. The most important component is the power unit, which supplies all other units with the required power to operate. A communication unit connects a sensor node with its neighbours via radio communication. The power consumption, available memory, processing ability, and programability of sensors has been rapidly improving. Table 1.1 lists out some of the available commercial sensor nodes also known as motes.

WSN consists of a large number of those tiny sensors deployed randomly in an area of interest. Nodes in a sensor network communicate in multi-hop fashion to deliver the

Table 1.1: Comparison between selected commercial motes

Sensor Node Name	Microcontroller	Tranceiver	Memory (Program + data)	Programm-ing
BTnode [35]	ATmega 128L	Chipcon CC1000 Bluetooth (2.4 GHz)	64 KB + 180 KB RAM	C and nesC
EPIC mote [36]	MSP430	250 kbit/s 2.4 GHz IEEE 802.15.4 Chipcon	10 KB RAM	C
Iris Mote [37]	ATmega 128L	Atmel AT86RF230 802.15.4/ZigBee	8 KB RAM	nesC
Mica2 [38]	ATmega 128L	Chipcon 868/916 MHz	4 KB RAM	nesC
Shimmer [39]	MSP430F1611	802.15.4 Shimmer SR7	48 KB flash 10 KB RAM	C and nesC
SunSPOT [40]	ARM 920T	802.15.4 ZigBee	512 KB RAM	Java
T-Mote Sky [41]	MSP430	802.15.4 Chipcon	10 KB RAM	C
FireFly [42]	ATmega 32L	Chipcon CC2420	2 KB RAM	C
Waspmote [43]	ATmega 128L	ZigBee/802.15.4/ DigiMesh/RF	8 KB SRAM	C
WISense [44]	TI MSP430G2955	TI CC2520 (2.4 GHz)	4 KB RAM	C

collected data to a central processing unit called the base station or the sink node. The networking of sensor nodes has been a challenging area of research due to its special characteristics. The main challenges of a WSN include:

- *Ability to withstand harsh environmental conditions* – For WSN, the challenges lie in sectors where conditions are harsh or devices are inaccessible once deployed. Requirements in such operating conditions are stringent enough to make it impossible to even consider wired sensors or battery operated devices. One such application is volcanic activity monitoring [45], where researchers deploy sensor nodes in more than three square kilometres around the volcano.

  The Berkeley Micromechanical Analysis and Design (BMAD) group is working on sensors for harsh environments [46], some of them using silicon carbide or even aluminium nitride, which retains its mechanical stability and piezoelectric properties up to temperatures over 1000 ^{o}C. These qualities enable the sensors to structurally sense and actuate material in harsh environment conditions.

- *Power consumption constraints for battery-powered nodes* – As per Moore's law, sensor devices are getting smaller in size, and as physical size decreases, so does energy capacity. Power consumption is the most important factor to determine the life of a sensor network, because usually sensor nodes are driven by battery and have very limited energy. After deploying sensor nodes to an observation site, it is difficult to charge the device batteries with sufficient frequency. This constraint challenges the user to design power-efficient systems.

  Existing power-aware solutions include systems like TinyDB [47], Corona [48, 49], and task mapping [50]. TinyDB automatically reorders operators such that operators with higher data transfer rate are executed earlier and on the same node, reducing data transfer over the network. Corona allows the user to specify a *freshness* constraint on queries, which state the time limit of old sensor data to be reusable for the current query. This improves cache hit and minimises energy costs. In [50], the authors provide a model for the task-mapping problem for both energy balance and total energy spent, and suggest mixed integer programming (MIP) formulations that gives optimal results with long runtime, coupled with a greedy heuristic.

- *Ability to cope with node and communication failures* – As a network ages, nodes

will fail. Periodically the network will have to be reconfigured to handle node/link failure or to redistribute network load. As researchers learn more about the environment they are studying, they may decide to insert additional sensing points. A simple solution to this problem is to increase the transmission range of all nodes; but this has the side-effect of creating undue congestion in other parts of the network. Two existing solutions for robustness in WSN are described in [51], [52].

- *Heterogeneity of nodes* – Early studies of WSNs focused on technologies based on homogeneous WSNs in which all nodes have the same system resource. However, heterogeneous wireless sensor networks have recently become more popular: the results of research [53] show that heterogeneous nodes can prolong network lifetime and improve network reliability without significantly increasing the cost. An example like volcano activity detection requires different types of sensors that can sense temperature, emitted gases, and seismic activities.

- *Scalability to large scale of deployment* – The number of sensor nodes may be in order of hundreds or thousands, and may reach millions. The system is required to incorporate and support a scaled network. In [54], the authors present a scalability analysis for WSN routing protocols.

In general, WSN programming abstractions fall in two major categories: node- and network-centric. The basic approach to developing a WSN application is to program sensor nodes using a low-level language like nesC [55], which makes program analysis and optimisation simpler and more accurate. An alternative approach is the network-centric, or macro-programming, approach [56, 30]. The network-centric approach provide high-level programming abstraction that includes a suitable programming model, compiler and runtime support for WSN. In this concept the programmer does not deal with low-level interfaces of WSN. The node-centric approach for stream data processing requires major code changes to be adapted for various applications. The higher-level abstraction provided by the user-friendly programming model in network-centric approach helps in developing complex real-time streaming applications.

1.3 Cloud Network

In the past most companies stored their data within their own system on a single server that was backed up by several replication servers. These servers were very expensive, and the company itself was in charge of maintaining the infrastructure. Microsoft's report on 'Economics of the Cloud' [57] shows that apart from storage and maintenance costs, electricity cost is rapidly rising to become the largest element of total burden of ownership, representing around 15–20% of total cost. Due to these financial reasons, companies have started using cloud services to store data reliably and to use the higher processing power provided by cloud services. With more people using the cloud, it has become one of the most notable trends in the past few years. Cloud computing uses virtualised processing and storage resources in conjunction with modern web-technologies to deliver abstract, scalable platforms and applications as on-demand services. The billing of these services directly depends on cloud service usage. In general, cloud service providers tend to offer services that can be grouped into three categories:

The first category is *Infrastructure as a Service (IaaS)*, where the cloud provides access to computing resources like server space, network connections, bandwidth, IP addresses, and load balancers in a virtualised environment across a public connection, usually the internet. One of the examples of IaaS is Amazon Elastic Compute Cloud (EC2) [58]. Amazon EC2 is a web service that provides resizable compute capacity in the cloud. The simple EC2 web service interface allows user to obtain and configure capacity with minimal effort; it provides complete control of computing resources and lets user run services on Amazon's reliable, quickly scalable, and secure computing environment.

The second category is *Platform as a Service (PaaS)*, where the cloud provides a platform and environment to allow developers to build applications and services over the internet. PaaS services are hosted in the cloud and accessed by users simply via their web browser. It allows users to create software applications using tools supplied by the provider. PaaS services can consist of set of features that customers can subscribe to; they can choose to include the features that meet their requirements while discarding others. Like most cloud offerings, PaaS services are generally paid on a subscription basis. PaaS features include operating systems, database management systems, storage, network access, and tools for design and development. Examples of PaaS includes Microsoft Azure [59] and Google AppEngine [60].

Table 1.2: Benefits of cloud services

IaaS	PaaS	SaaS
Scalability	No need to invest in physical infrastructure	Cross device compatibility
Data Security	Development is possible for non-experts	Accessible from anywhere
No single point of failure	Flexibility in choosing platform and software tools	Automated updates
Only pay for the resources that are actually used	Adaptability if circumstances change	
	Data security, backup and recovery	

The last category is *Software as a Service (SaaS)*, where the consumers access online software applications hosted in the cloud and useable for a wide range of tasks. Google, Twitter, Facebook and Flickr are all examples of SaaS, with users able to access the services via any internet-enabled device.

Table 1.2 lists the benefits of the each category of cloud services. The challenges in building cloud applications includes:

- *Scalability on demand* – Cloud computing workloads must scale rapidly, seamlessly and automatically to meet dynamic user demand, and the applications have to be developed with this in mind.
- *Minimise data transfer cost* – Most people think about computing and storage costs in running an application in the cloud. But, delivering data out of a datacentre costs money too and can be a hidden design decision. Minimising data transfer costs are critical in designing cloud systems.
- *Getting skilled programmers* – Mobile development has to be combined with cloud computing platforms to build efficient cloud-based applications. But finding a skilled programmers with this expertise is very difficult. To overcome this challenge, there should be a simple and easy-to-use high-level language with which a programmer can build any light-weight application.

- *Heterogenity* – Cloud clusters consist of heterogeneous computing nodes that differ in processing power, storage and main memory. The data processing system needs to support this heterogeneity of the cloud environment to add flexibility.

One special type of cloud service is concerned with streaming. Streaming refers to content that is processed to generate new stream. Examples are incoming RSS feeds (streams) that are combined to produce new RSS feeds. Other streaming applications watch the payments in a web-shop or analyse server loads and inform the system administrator as soon as a load peak is detected. A number of stream data management systems are available, among them Aurora [27], Borealis [28], and TelegraphCQ [61].

1.4 Problem Definition

The distributed nature of sensor and cloud applications, the scarce resources of sensor nodes, and the poor debugging facilities of distributed networks make the development of these applications a challenging task. To overcome or at least alleviate these problems, distributed programming environments seek an optimal trade-off between productivity, flexibility, and energy efficiency. Apart from these three major requirements, the programming model needs to support dynamic nature of the domain and timeliness for real-time streaming applications.

1. *Productivity* – Productivity is a key issue in designing distributed programming environments. Since most application developers are domain experts with limited formal training in programming, the programming model needs to be simple and easy to use. Existing programming models such as TinyDB [47], Mad-WiSe [30] provide SQL-based query language, while other models like Borealis [28], Regiment [56], and Storm [31] require XML files or Java to write an application query.

2. *Flexibility* – The complexity of applications require adequate flexibility in writing a wide range of applications such as data mining, monitoring, and signal processing. The SQL-like languages does not support complex arithmetic, analytical, and application-specific operations. We need a flexible model that supports a wide spectrum of operations and platforms without specific restrictions while developing an application. Wireless motes are becoming more intelligent and have

better resources, and new and better platforms are continually being introduced. The programming model need to cope with these technological changes without major design modifications.

3. *Energy-efficiency* – An important challenge in distributed environments is to deploy the computation units in the network to minimise communication and local computation costs and provide an efficient use of available resources like battery and communication bandwidth. The new programming model should provide an energy-efficient solution for the placement of computation over the available nodes.

4. *Dynamic redeployment of computation units* – In WSN, some nodes disconnect over time due to battery or network failure, and therefore topology may dynamically change over time. Similarly, cloud services may require scaling up or down of a cluster based on user traffic, which leads to topology change. The new data processing system should be robust enough to dynamically redeploy the computation units on the available nodes.

5. *Real-time Data Processing* – Some real-time streaming applications like bushfire warning, financial data analysis and server monitoring require timely and reliable processing. To determine the reliability and timeliness of the data, the model should be capable of performing time analysis and determining delay caused during computation.

Programming environments for distributed environments can be classified by their extent of productivity, flexibility, and efficiency, which can have opposing effects on each other. For example, high productivity often reduces the energy efficiency because node- or system-level details are abstracted from the user if they use abstracted language. Apart from these three challenges, the programming model needs to meet mobility and timeliness requirements in distributed applications.

1.5 Contribution

This thesis introduces Curracurrong, a distributed programming environment for WSNs that achieves a trade-off between the productivity, flexibility, and efficiency. We obtain productivity by introducing a new domain-specific language for stream programming. The model represents computations as a stream graph [15]. Any node in the stream graph represents a computational element called a stream operator, and any edge represents a communication channel. To succinctly represent high-level queries as stream graphs, we hierarchically compose stream graphs with from filters, pipelines, and splitter-joiners. We support data-rate annotations for stream operators that permit static computations of communication bandwidths. We provide an extensible library of predefined stream operators that offers flexibility in designing complex applications. Curracurrong facilitates in-network computation that lowers the communication overhead, in contrast to the standard approach of forwarding the raw data of sensor nodes directly to the base station. To improve energy efficiency, Curracurrong employs an algorithm that places stream operators optimally by minimising local computation and communication costs. We extend the placement problem to migrating operators and introduce what we refer to as the *migrating operator placement problem* (*MOPP*); this places operators of stream graphs on network nodes so that energy costs are minimised. The placement takes changes of queries and migration of operators into account that targets the problem of the dynamic nature of distributed environments. We propose an approach that measures time delays in stream query processing that aims to achieve timeliness in the system. Our model establishes a causality relationship between consumed and produced data tokens at each operator and their corresponding occurrence times. The total time taken for computation from the input to the output of a query (end-to-end delay) is computed by the causality relationships and periodic schedules for stream queries. A summary of the contributions of our work is as follows:

- The design and implementation of a stream programming environment for WSNs based on an easy-to-use query language and a runtime environment that improves development productivity.

- A predefined library of stream operators that includes signal processing and light data mining tasks. Curracurrong facilitates the user to write domain- specific stream operators for achieving flexibility.

- Curracurrong Cloud, a light-weight distributed stream processing system designed for deployment in large distributed clusters hosted using cloud computing infrastructure.
- The modelling of energy-efficient operator placement problem as multiway cut [62], and the implementation of an approximation heuristic that bounds the quality of the placement.
- The modelling of the NP-hard *migrating operator placement problem* (*MOPP*) that computes a placement and minimises computation, communication, and transition costs. We devise a dynamic program that computes an optimal operator placement for *compositional* streams which is a sub-class of general stream graphs in polynomial time.
- Denotational semantics for stream data processing that explains time information propagation and an algorithm to measure the end-to-end delays in a stream graph.

1.6 Publications

1. Vasvi Kakkad, Saeed Attar, Andrew E. Santosa, Alan Fekete, and Bernhard Scholz. **Curracurrong: a Stream Programming Environment for Wireless Sensor Networks**. Software: Practice and Experience, Vol. 14, issue 2, pp 175-199. John Wiley & Sons, Ltd., 2014.
2. Vasvi Kakkad, Andrew Santosa, Bernhard Scholz. **Migrating Operator Placement for Compositional Stream Graphs**. In Proc. of ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWIM'12), pp 124-134. Paphos, Cypress Island, October 2012.
3. Vasvi Kakkad, Akon Dey, Alan Fekete, Bernhard Scholz. **Curracurrong Cloud: Stream Processing in Cloud**. CloudDB'14, Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on , vol., no., pp.207,214, 2014.
4. Vasvi Kakkad, Andrew Santosa, Bernhard Scholz. **Computing End-to-End Delays in Stream Query Processing**, submitted to SCP, Elsevier.

1.7 Organisation of the Thesis

The thesis is organised as follows. Chapter 2 gives the details about the new programming model along with query language syntax. We discuss Curracurrong's novel approach to operator placement and deployment on the sensor nodes with the goal of energy efficiency. The operator placement problem is modelled as multiway cut problem and find an approximation using heuristic. In Chapter 3 we describe Curracurrong Cloud, a light-weight, distributed stream processing system designed to be deployed in large distributed clusters hosted on cloud computing infrastructure. We describe the changes we made to the existing Curracurrong system so that it runs in a cloud-hosted cluster where the computational resources are much less constrained than WSNs. In Chapter 4 we introduce the NP-hard *migrating operator placement problem* (*MOPP*), which computes a placement that minimises computation, communication, and transition costs, where transition cost is the cost incurred while moving an operator to a new node. We devise a dynamic program that computes an optimal operator placement for *compositional* streams – a sub-class of general stream graphs in polynomial time. Chapter 5 discusses the timeliness in Curracurrong system, where we present denotational semantics for stream data processing that explains time information propagation. We design an algorithm to measure the end-to-end delays using event causality in a stream graph, and show the effectiveness of the approach. Chapter 6, a literature review, shows the detailed related work in the area of stream data processing. Conclusions and directions for future work are presented in Chapter 7.

Chapter 2

Curracurrong for WSN

Wireless sensor networks (WSNs) are distributed networks composed of a set of small sensor devices comprising a processor, memory, a set of transducers, and a radio transceiver. Typical applications of sensor networks include habitat monitoring [63], environmental sampling [64], disaster area monitoring [11], and surveillance [65]. The development of sensor applications is a challenging task because of the distributed nature of sensor applications, the scarce resources of sensor nodes, and the poor debugging facilities of WSNs. To overcome or at least alleviate these problems, a WSN programming model seeks an optimal trade-off between *productivity*, *flexibility*, and *energy efficiency*.

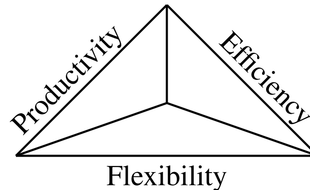
Productivity is one of the key issues in designing WSN programming environments. Since most users will be domain experts with limited formal training in programming, the model needs to be simple and easy to use. Models for WSN applications are broadly classified into *node-centric* [55, 66] and *network-centric* [67, 56, 68]. In the node-centric model, apart from application logic, users specify nodes' local behaviour with low-level details such as communication between nodes and efficient resource usage. This requires users to have a thorough understanding of embedded and distributed systems programming. In contrast, the network-centric programming model expresses the behaviour of a sensor application in a declarative fashion and provides high development productivity [47, 67, 56, 68].

Flexibility is another key issue for WSN programming environments. The complexity of WSN applications requires flexibility in writing a wide range of applications such as data mining, monitoring, and signal processing. The low-level details in node-centric programming models offer good flexibility in allowing users to write diverse

applications. However, network-centric models such as database models [47, 69] only support SQL queries; this can limit a range of applications, and therefore the approach has lesser flexibility.

Energy efficiency is critical in the design of WSN programming environments. Sensors are powered by on-board batteries, which have limited lifetime. In node-centric programming models, users have the capability and the responsibility to configure system-level settings to reduce the energy consumption. However, the higher levels of abstraction in network-centric models means that the language or deployment environment needs to optimise the task partitioning to achieve efficiency.

WSN programming environments can be classified by their extent of productivity, flexibility, and efficiency, which can have opposing effects on each other. For example, high productivity often reduces the energy efficiency because node-level details are abstracted from the user. The trade-offs between productivity, flexibility, and efficiency are illustrated in the triangle below: finding a "sweet spot" is a challenge.



We develop Curracurrong [70], a stream programming system for distributed environments like WSN and cloud. The model achieves an optimal trade-off between productivity, flexibility, and efficiency. We obtain **productivity** by introducing a new domain-specific language for stream programming. The model represents computations as a stream graph [15]. Any node in the stream graph represents a computational element, called a stream *operator*, and any edge represents a communication *channel*. To succinctly represent high-level queries as stream graphs, we hierarchically compose stream graphs built from filters, pipelines, and splitter-joiners. This is similar to other structured stream programming languages such as StreamIt [71]. An implicit type system ensures the type compatibility of stream operators. We support data rate annotations for stream operators that permit static computations of communication bandwidths. An extensible library of predefined stream operators is provided to offer **flexibility** in designing complex applications. Curracurrong facilitates *in-network* computations that lower the communication overhead; this is in contrast to the standard approach of data

forwarding, which propagates the raw data of sensor nodes directly to the base station. To improve **energy efficiency**, Curracurrong employs an algorithm that places stream operators optimally by minimising local computation and communication costs. This chapter explains the Curracurrong system for WSN and in the next chapter we discuss the same for cloud environment.

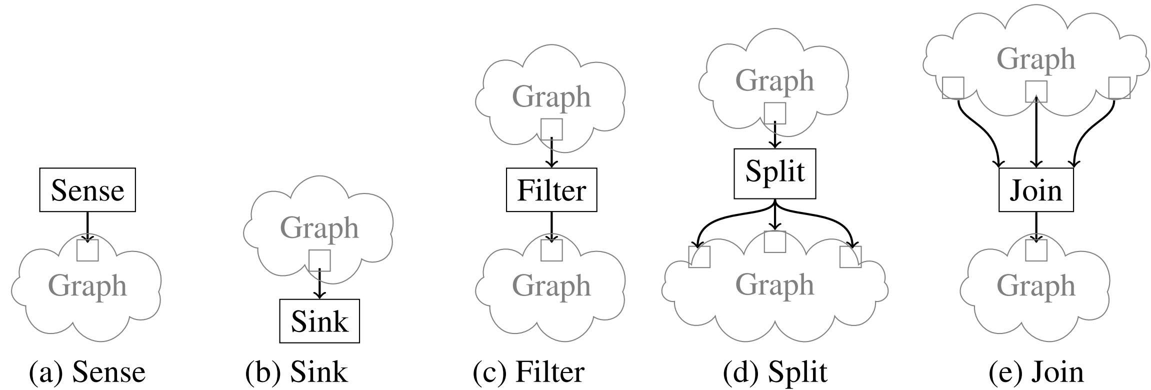
The contributions of our work in this chapter are as follows:

- The design and implementation of a stream programming environment based on an easy-to-use *query language* and a runtime environment that improves development productivity.
- A predefined library of stream operators that includes signal processing and light data mining tasks. In addition, Curracurrong facilitates the user to write domain-specific stream operators for achieving flexibility.
- The modelling of energy-efficient operator placement problem as *multiway cut* [62], and the implementation of an approximation heuristic that bounds the quality of the placement.

The chapter is organised as follows. Section 2.1 provides a detailed description of Curracurrong's query language and data model. Section 2.2 presents the Curracurrong query processing system that is responsible for handling activities at sensor node level. Section 2.3 discusses a novel operator placement algorithm to optimise energy consumption. Section 2.4 provides an evaluation of the algorithm.

2.1 Curracurrong Query Language

The model of computation in Curracurrong is based on the notion of a *stream* – a sequence of values that are seen successively over time. A program in Curracurrong is a *stream graph*, consisting of independent nodes representing stream operators (also called *actors*) that communicate over edges representing FIFO data channels. Each channel carries a stream of data values. As Figure 2.1 shows, each stream graph in Curracurrong is a composition of five types of stream operators, namely *sense*, *sink*, *filter*, *split*, and *join*. Each operator has two stages of execution: *initialization* and *steady state*. During initialisation, input and output channels are created and static parameters are initialised, whereas during the steady state there is an endless repetition of a single



(a) Sense (b) Sink (c) Filter (d) Split (e) Join

Figure 2.1: Curracurrong stream operators

step, driven by the stream of values on the input channel(s) and placing a stream of values onto the output channel(s). One step of a stream operator pops or peeks data items from input channels, processes or transforms them, and pushes the transformed items into output channels. The number of data items peeked, popped, and pushed by each step of a given stream operator is constant throughout invocations. This allows us to compute communication bandwidths statically. The stream operators in Figure 2.1 are explained below:

(a) The *sense* operators in Figure 2.1(a) are the initial nodes of the graph. Sensor nodes sample the environmental data and through sense operators inject sampled data streams into the network. A sense operator has only one output channel and is located on predetermined sensor nodes.

(b) The *sink* operator in Figure 2.1(b) is a terminal node of the stream graph. It collects the final results of the computation from the incoming channel, after which they might be passed to other forms of computer system. A sink has only one input channel and is located on the predetermined base station.

(c) The *filter* in Figure 2.1(c) is an operator with a single input and single output channel. It reads the data items from its input channel, applies a computation to transform them, and pushes results into the output channel. The results may have different values, data type, and rate from the input.

(d) The *split* operator in Figure 2.1(d) has a single input stream and distributes data

items between its output streams. There are two types of split operators: *duplication* and *round-robin*. The duplication split copies each input item to all output channels. On the other hand, a round-robin split distributes its input data to the output channels following a particular sequence given to the channels.

(e) The *join* operator in Figure 2.1(e) combines multiple input channels into one output channel. The output has the same values and data types as the input channels, but possibly with different rates.

The stream graphs in Curracurrong are constructed by a hierarchical composition of *pipeline*, split, and join structures [71]. The pipeline is composed of two or more graphs arranged in a sequence, whereas split and join structures are illustrated in Figure 2.1(d) and Figure 2.1(e), respectively. These compositions allow the modular comprehension of a query program, improving the programmer's productivity. Because Curracurrong has a query language with hierarchical structured queries, it is easy to represent queries with graphical notation when desired (Figure 2.6); however, our system expects queries to be defined using the textual representation (Figure 2.5).

Curracurrong has a complete data type system that includes all basic primitive data types, such as integers of various sizes, single- and double-precision floating points, and character strings. The type system includes composite types such as fixed-length arrays and records. The elements of composite types in turn can be either primitive or composite types. The Curracurrong type system is implemented in Java. It is extensible via an interface for data type classes.

2.1.1 Language Syntax

The basic syntax for a stream operator in Curracurrong query language includes a parameter list with zero or more parameters specified in brackets. These parameters are used either for the initialisation of a stream operator or as runtime variables. For example, in a sense operator

```
Sense[node="1.1.1.1", interval=20, starttime=12345]
```

the user passes three parameters to a sense operator, including the location of the operator on a sensor node 1.1.1.1, an execution interval, and a start time of the execution. `Interval` and `starttime` parameters specify the scheduling semantic of the operator, where sampling is performed every 20 ms starting from the wall clock time 12345.

```
query Average =
   (Sense[node="1.1.1.1", rate=100] ->
   Select[field=3] ->
   AverageFilter[window=5] |
   Sense[node="1.1.1.2", rate=50] ->
   Select[field=3] ->
   AverageFilter[window=5] |
   Sense[node="1.1.1.3", rate=200] ->
   Select[field=3] ->
   AverageFilter[window=5])
```

Figure 2.2: `Average` query

Curracurrong has a set of predefined stream operators for data collection and signal processing applications, including: `Sense`, representing sensing operation; `Select`, to select a data field to propagate further, `AverageFilter` to compute the average of a number of data; `VarianceFilter`, to compute the variance of a set of sensor readings; `Threshold`, to specify a threshold on the data value to be propagated; and `LowPassFilter` and `HighPassFilter` to remove low- and high-frequency signal reading, respectively. Users can expand this set of operators by extending a Java abstract class, `StreamOpFilter`, enabling them to define any kind of application-specific computation.

Figure 2.2 shows the `Average` query. In this example, we compute average temperature sensor readings at each node and send the results to the base station. We define the query using `query` command with a query expression. The query is composed of the `Sense`, `Select`, and `AverageFilter` stream operators of the predefined library. The symbol "`->`" between stream operators indicates the pipeline structure, and pipes the output stream into the subsequent operators. The join operator – expressed as "`|`" between query expressions – merges its input streams and sends the data to its single output stream.

In this query, the `Sense` operators sample the data with rates of 100, 50, and 200 ms, given explicitly as `rate` parameters. Curracurrong query language supports different sample rates for the `Sense` operators. The output channel of a `Sense` operator consists of sensor-sampled values in the Curracurrong data record format. Each field in the record represents the reading of one of the sensors. The `Select` operator picks a temperature value indicated by the `field` parameter. The `AverageFilter` is applied

to selected values based on the window size specified by the `window` parameter, which indicates the number of data elements to be averaged.

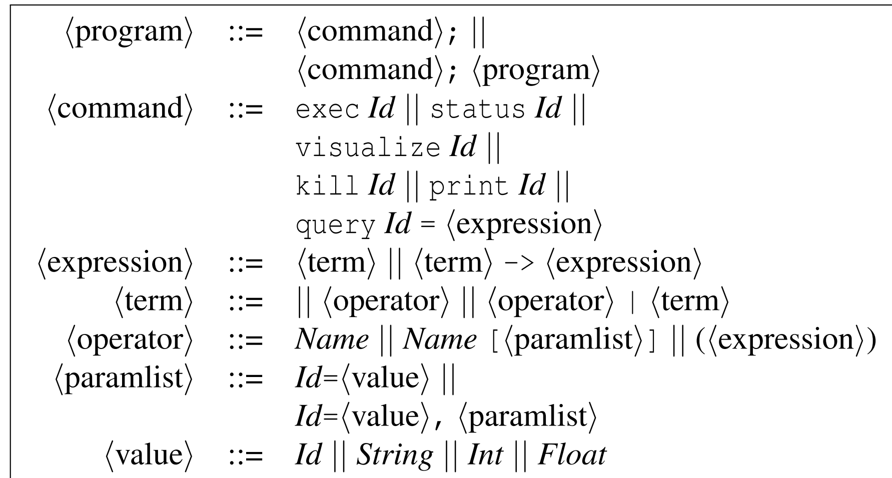
Curracurrong supports selection, join, split, and aggregation queries, and the environment allows commands for defining, executing, stopping, visualising, and checking the status of a query in the system. Queries are defined by the `query` clause, which is associated with an identifier. The identifier can be used by other commands to control the query. Figure 2.3 gives the syntax of the language in *Backus-Naur Form* (*BNF*). The commands to control the queries are listed below:

- `exec` *Id* executes a query with an identifier *Id*. This command activates a query by sending stream graph construction messages to the nodes, and executes the operators.
- `kill` *Id* stops an active query specified by *Id*. This command deactivates the query, destroying all of the stream operators on the nodes.
- `visualize` *Id* shows a graphical presentation of the stream graph of a given query.
- `status` *Id* displays the status of the query.
- `print` *Id* displays the query.

Instead of using an explicit `kill` command, we can limit the execution time of a query via a `repeat` parameter – for example, `repeat=10` stops the query execution after 10 iterations, or we can include a stopping condition triggered by an event.

2.1.2 Case Study: Seismic Event Detection

We demonstrate how Curracurrong is used for a seismic event detection application from literature [11]. Curracurrong programmability is compared with that of TinyDB [47] and Borealis [28]. A seismic event detection dataflow graph is illustrated in Figure 2.4. This application attempts to detect the onset of an earthquake or other interesting seismic events using a simple “ratio of two low-pass filters” approach on a seismometer signal. The program samples the seismic sensor every 10 ms and passes the resulting data through two exponentially weighted moving average (EWMA) operators with different gain settings. If the ratio of these operators exceeds a threshold, this indicates that



⟨program⟩	::=	⟨command⟩; ||
		⟨command⟩; ⟨program⟩
⟨command⟩	::=	`exec` *Id* || `status` *Id* ||
		`visualize` *Id* ||
		`kill` *Id* || `print` *Id* ||
		`query` *Id* = ⟨expression⟩
⟨expression⟩	::=	⟨term⟩ || ⟨term⟩ -> ⟨expression⟩
⟨term⟩	::=	|| ⟨operator⟩ || ⟨operator⟩ | ⟨term⟩
⟨operator⟩	::=	*Name* || *Name* [⟨paramlist⟩] || (⟨expression⟩)
⟨paramlist⟩	::=	*Id*=⟨value⟩ ||
		Id=⟨value⟩, ⟨paramlist⟩
⟨value⟩	::=	*Id* || *String* || *Int* || *Float*

Figure 2.3: Curracurrong query language BNF

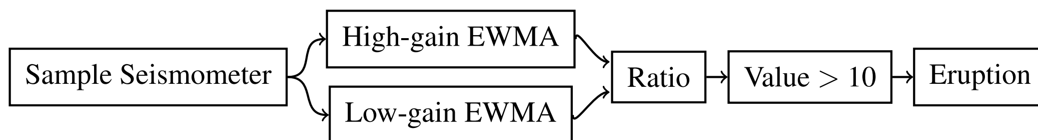


Figure 2.4: Seismic event detection – dataflow graph

the seismic signal is significantly larger than the background noise, then a "detected" message is transmitted to the base station.

Figure 2.5 shows the Curracurrong query `Event` for the seismic event detection application. The query uses the `Sense` stream operator (defined in the operator library) with an interval parameter. The application samples the seismometer reading using a sensor with address "1.1.1.1" every 10 ms. The time-stamped data stream is pushed into a `Split` operator that generates two copies of the input stream. The output stream from the splitter is given to the `EWMA` operators with two different gain values: 0.2 and 0.8. The average values from two `EWMA` operators are passed through the built-in join operator and forwarded to the next user-defined operator `Ratio`. `Ratio` calculates a ratio of the two averages and pushes the resulting value to the next built-in operator `Threshold`. An event detection signal is sent to the sink operator located on the base station if the average ratio is beyond the threshold value – here, 10. We next show the complete code for user-defined operators `EWMA` and `Ratio`. Figure 2.6 demonstrates the corresponding stream graph for the `Event` query.

```
query Event =
   Sense[node="1.1.1.1",interval=10] ->
   ([type="duplicate"] EWMA[gain=0.2] | EWMA[gain=0.8]) ->
   Ratio ->
   Threshold[value=10];
```

Figure 2.5: `Seismic event` – Curracurrong query

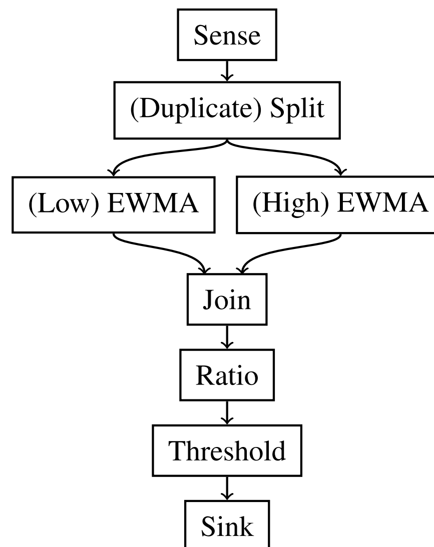


Figure 2.6: Seismic event detection – Curracurrong stream graph

- EWMA

```
 1: public class StreamOpEWMA extends StreamOpFilter {
 2:   private float gain;
 3:   private float state, out;
 4:   public void initialize(int opID, Hashtable propertyMap) {
 5:     super.initialize(opID, propertyMap);
 6:     String value = getPropertyValue("gain").tostring();
 7:     if(value != null) {
 8:       gain = Float.parseFloat(value);
 9:     }
10:     state = 0.0f;
11:   }
12:   public void execute() {
```

```
13:     DRecord rec = (DRecord) receive();
14:     float in = Float.parseFloat(rec.getElement(1));
15:     out = state = gain * in + (1.0 - gain) * state;
16:     rec.setElement(1) = out;
17:     send(rec);
18:   }
19:}
```

`EWMA` operator extracts a parameter 'gain' from the parameter list during initialisation – in *initialize* function and uses it during execution. The *execute* method is called when the scheduler puts StreamOpEWMA operator in a ready queue (cf. 2.2). During execution, the operator receives first data from its input queue using *receive()*, performs computation and sets output data in the record, and sends it to the output queue.

- Ratio

```
 1: public class StreamOpRatio extends StreamOpFilter {
 2:   private float out;
 3:   public void initialize(int opID, Hashtable propertyMap) {
 4:     super.initialize(opID, propertyMap);
 5:   }
 6:   public void execute() {
 7:     DRecord rec = (DRecord) receive();
 8:     float in1 = Float.parseFloat(rec.getElement(1));
 9:     rec = (DRecord) receive();
10:     float in2 = Float.parseFloat(rec.getElement(1));
11:     out = in1/in2;
12:     rec.setElement(1) = out;
13:     send(rec);
14:   }
15:}
```

When `Ratio` operator is scheduled for the execution, it collects first two data from its input queue, computes the ratio by dividing first data into second data, and sets the *out* value in the record. Finally the record is sent to the output queue of `Ratio`.

We compare the expressiveness of our language with that of TinyDB. TinyDB query processing system is a declarative SQL-like query interface that uses clauses, such as

SELECT, FROM, WHERE, HAVING. As mentioned in the literature [47], in the latest version of TinyDB, the WHERE and HAVING clauses contain only simple conjunction over arithmetic comparison operators; arithmetic expressions are limited to the set $\{+, -, *, /\}$. Due to limitations in the query language, this is less expressive than the Curracurrong query language. An advantage is that operators such as `EWMA` can be easily implemented in the Curracurrong system, while this is not the case with the TinyDB system.

We further compare our query with a sample Borealis query for the same application (cf. Figure 2.7). In the Borealis system, a user writes XML files to define a query. As shown in the XML code, a Borealis query contains stream, schema, and boxes definitions. Each box represents a query operator and corresponding input and output streams. For example, the `sense` operator is defined within *box* and requires 10 XML tags. In a Curracurrong query, specifying the `sense` operator is much shorter. Other issues of complexity in writing Borealis queries include choice of predefined box type, definition of input–output streams, manual type inference over I/O streams, manual definition of connection points between boxes, and the need for a separate XML file for query deployment.

The latest version of the Borealis system supports a graphical editor to define queries [72]. However, there are several reports that frequent users of a programming system prefer a traditional textual interface over the graphical [73]. The current case study shows that defining any query using the Curracurrong programming model is simpler and shorter than using the Borealis system's textual interface. The case study also illustrates that Curracurrong offers adequate flexibility in defining domain-specific queries using built-in and user-defined query operators. Unlike Borealis, Curracurrong provides an automatic type inference feature that reduces the burden of application developer. The query deployment in Borealis requires the user to write a separate XML file, while Curracurrong performs deployment automatically, as explained in the next section.

2.2 Query Processing System

Curracurrong's query processing system consists of two subsystems: a *server module* and a *runtime environment*. The server module is responsible for parsing the query, mapping it onto a stream graph, computing the placement of the stream operators on

```
<?xml version="1.0"?>
<!DOCTYPE borealis SYSTEM "../../src/src/borealis.dtd">
<borealis>
    <input  stream="ElementInput" schema="Element" />
    <output stream="SenseOutput" schema="Element"/>
    <output stream="LowEWMAOutput" schema="Element"/>
    <output stream="HighEWMAOutput" schema="Element"/>
    <output stream="JoinOutput"  schema="Element"/>
    <output stream="ThresholdOutput" schema="Element"/>

    <schema name="Element">
        <field name="timestamp" type="int"/>
        <field name="id"        type="int"/>
        <field name="reading"  type="float"/>
    </schema>

   <query name="Event">
      <box name="sense" type="map">
          <in  stream="ElementInput"/>
          <out stream="SenseOutput"/>
          <parameter name="expression.0" value="timestamp"/>
          <parameter name="output-field-name.0" value="timestamp"/>
          <parameter name="expression.1"        value="id"/>
          <parameter name="output-field-name.1" value="id"/>
          <parameter name="expression.2"        value="reading"/>
          <parameter name="output-field-name.2" value="reading"/>
       </box>
       <box name="LowEWMA" type="aggregate">
          <in name="SenseOutput" />
          <out name="LowEWMAOutput" />
          <parameter name="aggregate-function.0" value="ewma(0.2)" />
          <parameter name="aggregate-function-output-name" value="ewma"/>
          <parameter name="order-on-field" value="TUPLENUM" />
          <parameter name="group-by" value="id" />
          <parameter name="window-size" value="" />
          <parameter name="window-size-by" value="TUPLES" />
       </box>
       <box name="HighEWMA" type="aggregate">
          <in name="SenseOutput" />
          <out name="HighEWMAOutput" />
          <parameter name="aggregate-function.0" value="ewma(0.8)" />
          <parameter name="aggregate-function-output-name" value="ewma"/>
```

Figure 2.7: Borealis query definition – Event.xml (Part-1)

```
<parameter name="order-on-field" value="TUPLENUM" />
         <parameter name="group-by" value="id" />
         <parameter name="window-size" value="" />
         <parameter name="window-size-by" value="TUPLES" />
      </box>
      <box name="joinRatio" type="aurorajoin">
         <in  stream="LowEWMAOutput"/>
         <in  stream="HighEWMAOutput"/>
         <out stream="JoinOutput"/>
         <parameter name="predicate" value="left.id==right.id" />
         <parameter name="left-order-by" value="VALUES" />
         <parameter name="right-order-by" value="VALUES" />
         <parameter name="left-order-on-field" value="timestamp" />
         <parameter name="right-order-on-field" value="timestamp" />
         <parameter name="out-field.0" value="left.timestamp" />
         <parameter name="out-field-name.0" value="timestamp" />
         <parameter name="out-field.1" value="left.id" />
         <parameter name="out-field-name.1" value="id" />
         <parameter name="out-field.2" value=
                                 "left.reading/right.reading" />
         <parameter name="out-field-name.2" value="reading" />
      </box>
      <box name="threshold" type="filter">
         <in  stream="JoinOutput"/>
         <out stream="ThresholdOutput"/>
         <parameter name="expression.0" value="reading &lt;= 10"/>
      </box>
    </query>
</borealis>
```

Figure 2.8: Borealis query definition – Event.xml (Part-2)

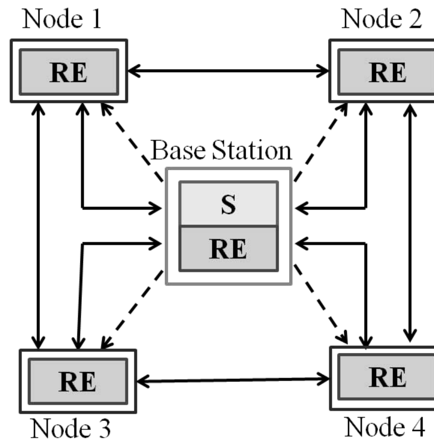


Figure 2.9: Curracurrong system components. Here, S indicates the server module and RE is the runtime environment. The broken lines are administrative channels and bold lines are data channels.

sensor nodes, and computing the channel bandwidths and data types of the channels. The runtime environment, on the other hand, manages all aspects of the in-network execution of the queries, such as communication between the stream operators and their scheduling on sensor nodes. The server module resides on the base station, while each sensor node executes an instance of the runtime environment. Figure 2.9 illustrates the structure of the system including server module and runtime environment, on base station and sensor nodes.

2.2.1 Server Module

The server module parses user queries using a *syntax processor*. The syntax processor is implemented using ANTLR, a parser generator for LL(k) grammars [74]. The parsed user-defined queries are translated to an internal representation of stream graphs, where the vertices are stream operators and the directed edges are FIFO channels. Figure 2.10 shows the stream graph of the `Average` query, where the `->` symbols in Figure 2.2 are translated into channels, and the outputs of streams separated by `|` are joined by a joiner operator.

The server module performs specific tasks on the stream graph to prepare for deployment, including the computation of bandwidth and data types of channels followed by the computation of the optimal stream operator placement on the sensor nodes. On

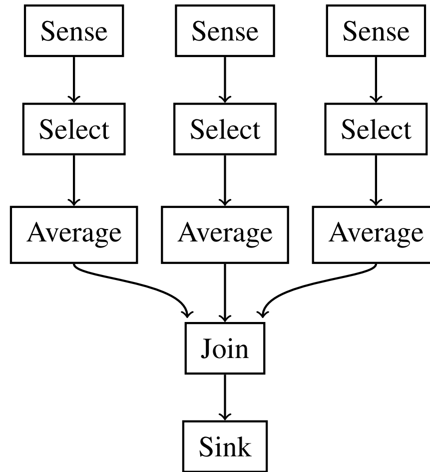


Figure 2.10: `Average` stream graph. The graph is generated by the server module from the query of Figure 2.2, by converting "`->`" into a FIFO channel and "`|`" into a join operator with the corresponding channels.

receiving the deployment commands from the server module, the runtime environment module in the base station sends commands to the runtime environment module on each sensor node to initialise the stream operators. Once all stream operators are initialised, each sensor node sends an acknowledgment message to the server, which in turn sends a command to start the query execution on each sensor node.

2.2.2 Runtime Environment

The Curracurrong runtime environment is implemented in Java. To run WSN applications the system runtime was targeted for the SunSPOT virtual machine Squawk [75], an open-source Java ME virtual machine for embedded systems and small devices that supports dynamic loading of Java bytecode. Thus, Curracurrong runtime environment can be dynamically reprogrammed. The runtime environment manages the communication, execution and administration of stream operators on each sensor node. It has three main components: the *administrator*, *communicator*, and *scheduler*. Figure 2.11 illustrates the components of runtime environment.

- The *administrator* runs as a single thread and receives administrative commands (`exec` and `kill`) from the base station. It executes the construction and destruction of stream operators on the sensor node and other commands.

- The *communicator* is implemented as a parallel running thread. It is responsible for handling the communications of input and output channels of the stream operators placed on sensor nodes. When the communicator received a data message, it places it in the queue of recipients based on queue id. At the end of each execution of the stream operators, the communicator checks the output channels. If the receiver operator of the new data is located on the same node, it puts the data in its input channels; otherwise, it sends the data to the communicator of the sensor node where the receiver operator is located. The communicator has a routing table that indicates the placement of stream operators on the nodes.

  One of the design characteristics of the runtime environment is the separation of data and administration channels so that the communicator is responsible for data communication between the stream operators only; whereas the administrator handles administrative tasks such as operator creation, query deletion.

- The *scheduler* manages and controls the execution of stream operators *on the sensor node* based on their scheduling semantics. During the initialisation, each stream operator registers itself to the scheduler. The scheduler has two queues: *run* and *wait*. The run queue is a time-triggered priority-based blocking queue that orders tasks (operators) based on their execution time. The wait queue maintains the stream operators that do not have sufficient data items for execution. The communicator notifies the scheduler when it stores data items in the input channels of an operator in the wait queue. The scheduler moves the stream operator from the wait queue to the run queue if the queue has sufficient data items in its input channels.

To optimise energy consumption by a sensor node during runtime, we provide a *curfew model* in the runtime environment, to delay the data communication between sensor nodes. The model collects the data and sends a data packet to other sensor as a relatively big chunk rather sending small-sized data packets individually. The model exploits the possibility for an energy drop at sensor radio while communication is idle – that is, between two consecutive communications. When the curfew is set, the sensor radio is turned off until the output buffer is full. During this time, the sensor goes into shallow sleep mode, reducing energy consumption. The curfew model is ideal for event log applications, but not for real-time event-detection applications. It is a decision of

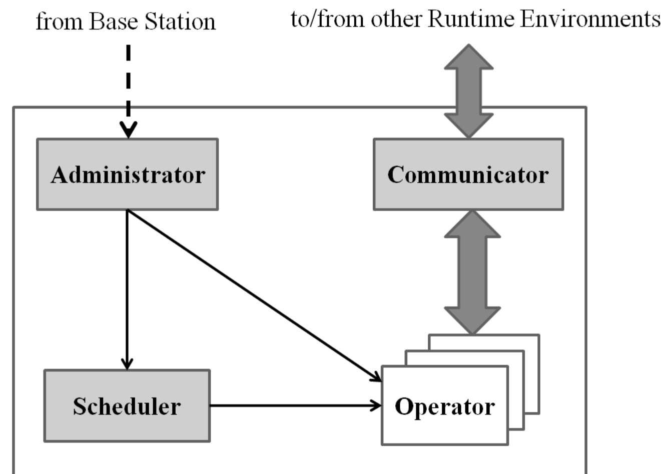


Figure 2.11: Runtime environment components

the application developer to apply the curfew model to get an added energy optimisation on top of the optimised operator placement; this is discussed in the next section.

2.3 Operator Placement and Deployment

We discuss Curracurrong's novel approach to operator placement and deployment on the sensor nodes in terms of energy efficiency. Curracurrong takes advantage of a stream graph to compute data types and bandwidth requirements of the channels, and uses the information to compute an optimised placement of operators on sensor nodes. Our placement approach could be applied in other WSN query processors, provided that the query can be converted to a graph of operators and each source has bounded bandwidth on its data stream. The former condition holds in most systems, but the latter is not a common assumption; instead most stream processing systems target widely varying data rates.

2.3.1 Data Type and Bandwidth Computation

Query construction of the Curracurrong system involves channel type and bandwidth inferencing. The data types and bandwidth of the channels are deduced ahead of time for the construction of the input and the output data channels by the server module. A type propagation algorithm computes the data type of all stream edges in the structured

Assume:
$G = (V, E)$ a stream graph with V ordered by a mapping $\alpha : V \to \mathbb{N}$ such that $\alpha(v_1) < \alpha(v_2)$ if $(v_1, v_2) \in E$, and $1 \leq \alpha(v) \leq |V|$ for any $v \in V$
for $i = 1$ **to** $|V|$ **do**
  Assume v such that $\alpha(v) = i$
    $I_v, O_v \subseteq E$ input/output channels of v
  for each $c \in O_v$ **do**
    $\tau(c), \beta(c) = T(c)(I_v), B(c)(I_v)$ **end for**
end for
Output τ, β

Figure 2.12: Topological sort-based graph walk. Here $T(c)(I_v)$ and $B(c)(I_v)$ are respectively the functions that computes the output channel c's type and bandwidth given input channels I_v.

stream graph by traversing the graph based on a topological ordering [76] of the operators, where an operator that produces data into a channel precedes another operator that consumes the data produced.

Figure 2.12 shows the pseudocode of the data type and bandwidth assignment algorithm. Starting with operators without input channels (`Sense` operators), the algorithm assigns the data type of the output channels. By following the topological sort, whenever the algorithm visits an operator during the walk, their input channels data types from which output channel types are computed are known. The algorithm continues to traverse the graph following the topological sort until data types have been assigned to all channels. It exhibits a linear running time in the number of nodes and edges: when V and E are respectively the sets of operators and channels, the running time is in $O(|V| + |E|)$.

The stream graph does not change throughout execution, and the output bandwidth of sense operators are statically known, therefore the bandwidth of data channels are computed statically. Each stream operator has a specific method, which returns the bandwidth of the output channels based on input bandwidth and operator type. For filter operators input and output bandwidth remains same; for join operator, output bandwidth is addition of all its input bandwidths; for duplicate splitter, each output channel has bandwidth same as its input and round-robin splitter divides the input bandwidth into the number of output channels. As shown in the pseudocode Figure 2.12, the bandwidth

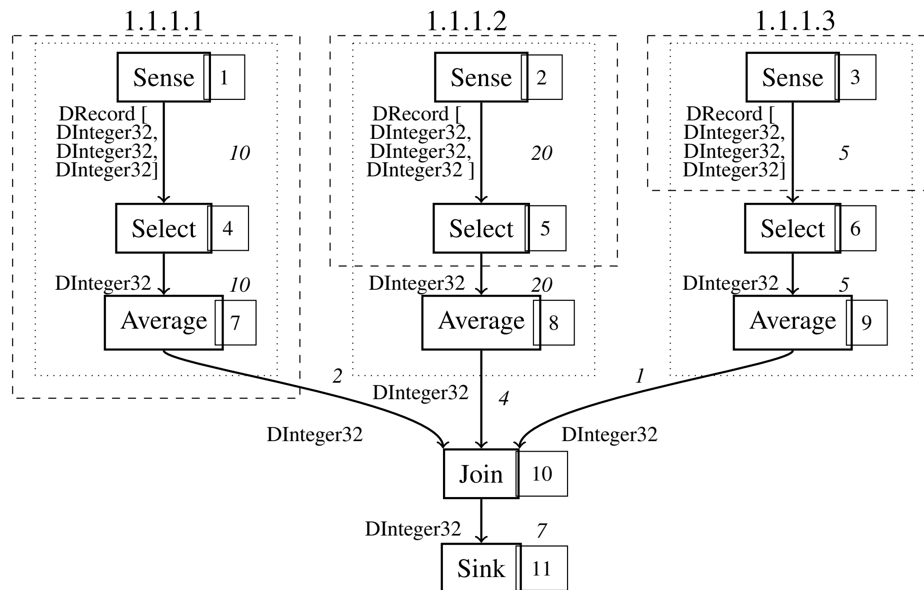


Figure 2.13: `Average` data type, bandwidth, and placement. The number in boxes denote topological ordering, the numbers attached to edges denote bandwidth requirement, DRecord and DInteger32 are channel data types. The record types (DRecord) have three elements of the primitive 32-bit integer types (DInteger32).

information is computed along with data types during the walk over the graph following the topological sort.

In the `Average` query example from Section 2.1, the query processor parses the query upon submission and constructs its stream graph (cf. Figure 2.10). During construction, the parser assigns a unique identifier to each operator in the stream graph based on topological ordering. Figure 2.13 illustrates the data type and bandwidth inference in the `Average` query stream graph based on the topological ordering of operators.

2.3.2 Placement of Operators

By minimising the local cost of executing operators on sensor node and communication between the nodes, we expect to effectively reduce the power consumption of the whole network. Communication between sensor nodes depends on the allocation of stream operators in the network, since it is costlier for operators located on different sensor nodes to communicate in comparison to operators located on the same node. However, it is unreasonable to place all operators on the same sensor node, because sensor nodes are tiny devices with low power resource. If operators requiring heavy computations

are placed on the same node will have potentially higher energy consumption and this may lead to the failure of the computation.

Formally, we model Curracurrong queries as a graph $G = (V,E)$. We are also given a set $N \subseteq V$ of *sensors*, which includes the sink operator placed on the base station. The operator placement problem is to find a mapping $p : V \to N$ that allocates the operators to the sensors and base stations, with the constraint that for all $n \in N, p(n) = n$. The aim is to find a mapping p that optimise the following two costs:

Local cost: The local cost represents the execution cost incurred while placing an operator on a given sensor node. The cost depends on the available energy level of the sensor node. For an operator $u \in V$ and the placement p, the placement cost is $w_L(u, p(u)) \in \mathbb{Q}^+$ which is given as $w_e(u) \cdot e(p(u))$, where $w_e : V \to \mathbb{N}$ is, for example, the *worst-case execution time* (*WCET*) in clock cycles and $e : N \to \mathbb{Q}^+$ the energy consumed per clock cycle.

Communication cost: The communication cost represents the cost when two communicating operators are placed on different sensor nodes. For channel $(u, v) \in E$, the communication cost is $w_C(u, v, p(u), p(v)) \in \mathbb{Q}^+$ defined as $w_s(u, v) \cdot I(p(u) \neq p(v))$, where $w_s : (V \times V) \to \mathbb{Q}^+$ is, for instance, the bandwidth requirement in bytes for communication over the channel. I is an *indicator* function, such that $I(\varphi) = 0$ if the condition φ did not hold, and $I(\varphi) = 1$ otherwise. Therefore, $I(p(u) \neq p(v)) = 1$ when operators u and v are placed on the different nodes, and 0 otherwise.

The total cost of operator placement to be minimised for stream graph $G = (V,E)$ is defined as:

$$f(p) = \sum_{u \in V} w_L(u, p(u)) + \sum_{(u,v) \in E} w_C(u, v, p(u), p(v)). \tag{2.1}$$

One simple approach to processing a query is to transmit all collected samples to the base station, and to perform all computations at the sink. This approach, which we refer to as *forwarding*, causes considerable burden on the network because of the high communication overhead. An alternative common approach is that of in-network computation [77], where raw data is read from sensor nodes, and after computations is aggregated and sent to the sink operator. The Curracurrong system further improves in-network operator placement by computing an optimal placement, using an algorithm

in which the problem is viewed as an instance of the *multiway cut* problem, also known as *k-way cut* (e.g., [78]). The placement in effect pushes some or all of the query execution tasks and stream operators to the sensor nodes, which in turn reduces the overall communication cost.

We now present a modelling of our placement problem as the *multiway cut* problem. Multiway cut is defined on an undirected graph $G^{MC} = (V^{MC}, E^{MC})$, where V^{MC} is the set of vertices, and E^{MC} is the set of edges. We are also given a set $T \subseteq V^{MC}$ of $k = |T|$ *terminal* vertices. A *multiway* cut, is a set $C \subseteq E^{MC}$ of edges such that in $G' = (V^{MC}, E^{MC} - C)$, no path exists between any two nodes of T, – that is, the terminal vertices become disconnected from each other. The multiway cut problem seeks a cut such that $|C|$ becomes minimal; the weighted multiway cut problem seeks a cut C such that $\sum_{e \in C} w(e)$ becomes minimal where $w(e)$ is the weight of edge e.

The placement problem is that of assigning labels to the stream operators, where each label corresponds to a sensor node (some specific operators in the stream graph have a fixed labelling, where the sense and sink operators are located on specific sensor nodes or the base station, respectively). First, we equate the set N of sensor nodes and base station to the set T of the terminal vertices of the multiway cut: $T = N$. A stream graph can then be viewed as a graph of a multiway cut problem. Due to the fixed assignments of sense operators to sensor nodes and sink operator to base station, we equate sensor nodes with sense operators, and base station with sink operator, such that $T = N \subseteq V$. We view the set of vertices V^{MC} as the set of all operators V and edges E^{MC} as communication channels along with edges from vertices to the sensor node: $E^{MC} = E \cup ((V \setminus N) \times N)$, with weights on the edges determined by the local and communication costs as follows:

$$w(u,v) = \begin{cases} (\sum_{q \in N} w_L(u,q)) - w_L(u,v) & \text{if } v \in N, u \notin N \\ w_C(u,v) & \text{otherwise.} \end{cases}$$

The formula $(\sum_{q \in N} w_L(u,q)) - w_L(u,v)$ represents the cost that is paid for *not* placing u on node v (see [79, 80]). Obtaining the minimum cut, then, corresponds to obtaining a set of edges with the least weights. Hence, the placement problem has a natural mapping to the multiway cut.

For $k = 2$, the multiway cut problem reduces to the $s-t$ *min-cut* problem [81], which can be solved via its dual problem: the *maximum flow* problem with a complexity bound

Assume $T = \{t_1, \ldots, t_k\} \subseteq V$
for $i = 1$ **to** k **do**
Connect terminals in $T - \{t_i\}$ with a new terminal $u \notin V$ via edges of weight $+\infty$.
$C_i =$ minimum $s - t$ cut between t_i and u.
end for
Assume for some $1 \leq l \leq k, w(C_l) \geq w(C_i)$ for all $1 \leq i \leq k$.
Output $C_1 \cup \ldots \cup C_{l-1} \cup C_{l+1} \cup \ldots \cup C_k$.

Figure 2.14: Isolating cut heuristic algorithm

Op\ Node	1.1.1.1	1.1.1.2	1.1.1.3	Base Station
1	0	∞	∞	∞
2	∞	0	∞	∞
3	∞	∞	0	∞
4	2	30	40	5
5	25	3	25	5
6	10	25	35	5
7	3	15	20	5
8	20	40	45	5
9	30	45	40	5
10	35	30	30	5
11	∞	∞	∞	0

Table 2.1: Local cost for operators on sensor nodes (fig. 2.13), here ∞ indicates very high cost; the cost for each operator on the base station is same except for sense and sink operators, which have fixed placement

polynomial in the size of the graph. However, the multiway cut problem is known to be NP-hard for $k \geq 3$. We therefore employ the algorithm of Dahlhaus et al. [82] (Figure 2.14), which provides a simple combinatorial isolation heuristic with an approximate solution bounded by $2 - \frac{2}{k}$ to the optimal solution. In this algorithm, we select a terminal and solve the $s - t$ min-cut between the terminal and the remaining $k - 1$ terminals, which minimally disconnects the two sets. The union of these *isolating* cuts *excluding* the cut with the heaviest weight gives the approximation to the optimal solution, because the union of $k - 1$ isolating cuts is a cut. Although this algorithm has a worse approximation bound than the algorithms of [83, 84], it has the lower runtime complexity. Therefore we have adopted it for Curracurrong.

Figure 2.13 shows the outcome of the placement algorithm for the `Average` query. In the first case, for the ease of explanation we assume a **uniform** local cost w_L , i.e., $w_L(u, p(u)) = c$, for each $u \in V$, where c is a constant. Placement with this assumption gives a trivial cut that can be easily seen from the stream graph. In the figure, larger dotted boxes depict sensor nodes enclosing a number of operators, with operators 10 and 11 placed on the base station. However, our placement algorithm is capable of computing non-trivial cuts. This is represented as the second case, where we consider the **non-uniform** local costs (Table 2.1) in addition to the communication costs; this placement is shown as dashed boxes. The remaining operators numbered $6, 8, 9, 10$, and 11 are placed on the base station.

In practice, the communication cost between two sensor nodes is affected by the number of hops required to send information on the physical network. Hence, depending on the *connectivity* of the nodes, communication might be cheaper or more expensive. Such consideration can be incorporated into the isolating cut heuristics by introducing a connectivity scaling factor $\Phi_{step} : N \to \mathbb{Q}^+$, defined as follows for any $x \in N$.

$$\Phi_{step}(x) = \frac{\sum_{y \in N} \text{shortest_path}(x, y)}{|N|},$$

where shortest_path(x, y) denotes the minimum scaling for communication between nodes x and y, which is proportional to the least number of hops between them. The scaling factor $\Phi_{step}(t_i)$ is applied to the communication edges in the solving of $s - t$ min-cut problem between t_i and u in Figure 2.14.

2.4 Experiments

We implemented the Curracurrong runtime environment in Java. For WSN the runtime is targeted for the SunSPOT virtual machine Squawk [75], an open-source Java ME virtual machine for embedded systems and small devices, which supports dynamic loading of Java bytecode. The implementation contained $2,000$ lines of Java code for the administration, communication, and scheduler components as explained in Section 2.2.2. Along with Curracurrong's runtime environment, we implemented the server module with query parser, the operator placement algorithm, and a library of commonly used operators, such as `Sense`, `Select`, and `AverageFilter`.

We evaluated the Curracurrong runtime environment and the operator placement

algorithm with a network of SunSPOT sensor nodes, measuring the energy consumption by the SunSPOT nodes to run particular Curracurrong queries. We assume that every node uses its local time and all nodes are synchronised. If clocks drift, we need to employ global time synchronisation protocols [85, 86],which is beyond the scope of this work. For precise measurements, we used a stabilised external power supply and resistors connected in series to the SunSPOTs. To measure the energy consumption, we used the voltage drop on the resistors and the voltage of the stabilised power supply with National Instruments' *Data Acquisition* (*DAQ*) devices. A small application collected the voltage readings from the DAQ device and computed the energy consumed by the node for a predefined time.

We performed experiments for up to five nodes, where we confirmed that the optimisation technique improves energy efficiency. In all the experiments, we run queries indefinitely and stop them explicitly. The results were validated by comparing them with those of an analytical model. We confirmed the execution efficiency of the heuristics by a comparison with an ILP solver. More detailed analysis of our system is illustrated by the experiments measuring the computation resource usage and cost distribution over query (cross-reference section number).

2.4.1 In-Network vs. Forwarding

Pushing computations to sensor nodes based on local and communication costs reduces energy consumption by nodes. Our experiments compared the forwarding (non-optimised) approach in which the raw data was sent to the base station for computation with our in-network (optimised) approach, where our operator placement approach pushed computations to the sensor nodes. We ran the following two queries:

Average Traffic: A query to log the vehicle traffic measurements from sensors placed at various locations. We took readings from the sensors in intervals of 10 s and log the average traffic from each sensor at every 60 s.

```
query Average = Sense[node="1.1.1.1",interval=10000] ->
    Select[field=2] ->
    Average[window=6];
```

Temperature Variance: A query to detect the temperature variance in the laboratory. Query samples the temperature reading at every 5 s and compute the variance

with past five readings. If the temperature variance exceeds 10 ^{o}C, the signal is sent to the base station.

```
query NetVariance = Sense[node="1.1.1.1",interval=5000] ->
      Select[field=2] ->
      Variance[window=5] ->
      Threshold[value=10];
```

The `Average` query induces a simple in-network computation, where the optimisation algorithm allocates most operators on the sensor nodes instead of the base station. With optimal placement, the number of packets sent per second was reduced by a factor of six, thus considerably reducing the communication overhead and hence the energy required by the SunSPOT motes. Whereas `Average` query sends an average of all sensed data to the base station, the `NetVariance` query sends the data only if the variance of samples exceeds a specific threshold. In this query, the optimised in-network computation significantly decreases the number of sent packets.

As explained before, we ran these queries connecting the external power supply to the SunSPOTs. Due to the spatial limitation of experimental setup, we used light-intensity sensor readings for both the queries. We ran these experiments for two network topologies:

1. *single-hop*, a scenario where sensor nodes were close to the base station and nodes communicated with the base station in single hop without routing through intermediate nodes; and

2. *multi-hop*, where each sensor node communicated with base station via all other remaining nodes.

Figure 2.15 shows the difference of the two topologies. Single hop is the 'best' case because it is the least costly topology for communication, whereas multi-hop is the 'worst'. Figure 2.16 shows the percentage of energy reduction when using in-network computation instead of forwarding for single- and multi-hop topologies. We observed that energy consumption can be reduced up to 5–7% for single-hop, and 22–24% for multi-hop. Potentially our approach becomes more effective the further the topology is from the 'best' and the closer it is to the 'worst.'

The results show that optimised in-network computation significantly reduces the communication overhead for the aggregating queries like `Average` and `NetVariance`.

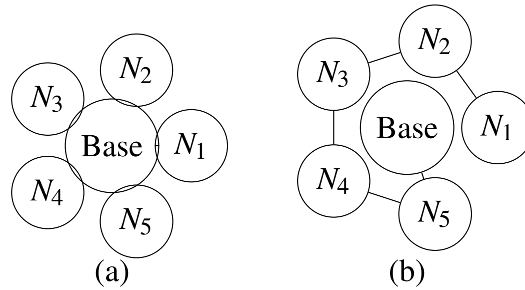


Figure 2.15: Single-hop (a) vs. multi-hop topology (b). For single-hop, a sensor node N_1 communicates directly to the base station, whereas for multi-hop, the data goes through N_2–N_5 before reaching the base station.

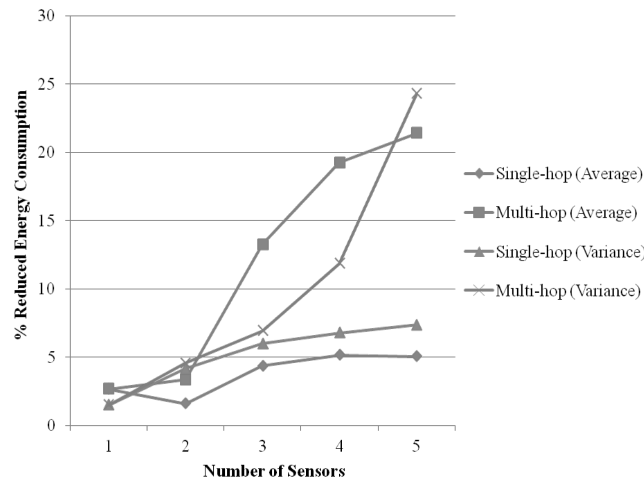


Figure 2.16: Experimental results. The maximum reduction of energy usage for single-hop topology is 5–7%, and for multi-hop topology is 22–24%.

2.4.2 Analytical Model

Due to the constraints in the availability of sensors and physical arrangements, we only ran our experiments on a small number of nodes. The validity of the results rests on the fact that the experimental results coincide with an analytical model, where energy consumption is given as:

$$\begin{aligned} Energy &= (U \times NormalCurrent \times RadioOnTime) + \\ &\quad (U \times IdleCurrent \times RadioOffTime). \end{aligned}$$

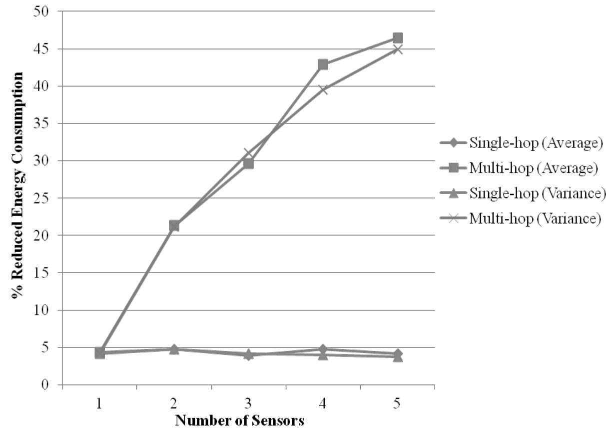


Figure 2.17: Analytical results. The maximum reduction in energy usage is approximately 5% for single-hop, and $45-46\%$ for multi-hop topology.

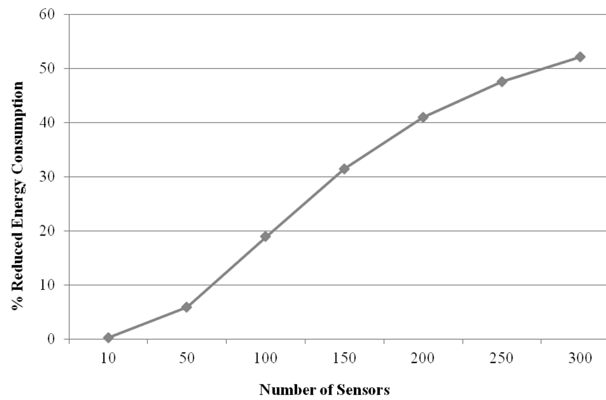


Figure 2.18: Scaled analytical results

Where, *U* indicates the voltage requirement for SunSPOT mote (5V), *RadioOnTime* is the amount of time a node performs communication and computation, and *RadioOffTime* is the time amount without communication and computation. These properties do not depend on the number of nodes in the single-hop case, but they depend quadratically to the number of nodes in the multi-hop case. In the formula, *NormalCurrent* indicates the current drawn in the mote's *run* mode (120 mA), and *IdleCurrent* is the current drawn during SunSPOT's *idle* mode (24 mA). The data is from [40].

Figure 2.17 illustrates the analytical reduction in energy requirement for `Average` and `NetVariance` queries for up to five sensor nodes, for both single- and multi-hop topologies. Here the reduction in energy consumption is approximately linear to the

Sensors	ILP (s)	Heuristic (ms)
1	0.1	0.011
2	0.1	0.024
3	3.3	0.040
4	176.2	0.063
5	*	0.093
6	*	0.114
7	*	0.144
8	*	0.165
9	*	0.198
10	*	0.229

Table 2.2: GLPK running time on `Average` instances

number of nodes, as is the case with the experimental results. In analytical results, the energy requirement with in-network computation can be reduced up to 5% with single hop and 45–46% with multi-hop. In our experimental results, on the other hand, the improvement was 5–7% with single-hop and 22–24% with multi-hop.

There are two potential causes for the difference between analytical and experimental results, specifically for multi-hop. The first can be attributed to slight changes in the environment, such as the light-intensity sensed by the motes at the time of the experiments, and the fluctuations in the power supply output. We observed in our initial experiments that the capacity of the batteries used varies widely, making it difficult to obtain measurement for a prolonged period. We therefore used an external power supply, but this still did not completely eliminate variations in the input current.

Figure 2.18 shows the percentage of reduction in energy requirement between in-network computation and forwarding from 5–300 nodes. The results in Figure 2.18 are for the `Average` query with window size 5 and a sampling period extended to two hours, as a low sample rate is typical in real wireless sensor networks. Thus, with the reduced communication overhead, we observed that the energy requirement can be reduced up to 51% in a network of 300 nodes with multi-hop topology. Importantly, we can confirm the approximately linear improvement in the energy requirement with the increasing number of nodes, demonstrating a potential scalability.

2.4.3 Heuristic Efficiency

We compare the runtime of the placement heuristics that we employed with the solving of an *integer linear programming* (*ILP*) formulation of the multiway cut, using the GLPK [87] with the AMPL [88] program, which is discussed in Section 2.4.3.1. The comparison is set out in Table 2.2. We ran both experiments on a dual-core 2.2 GHz Intel machine with 4 GB RAM. In Table 2.2, **Sensors** indicate the number of sensor nodes, **ILP** indicates the running times for GLPK runs in seconds (where $*$ represents a run that takes more than 2 h), and **Heuristic** indicates the running times for the heuristic runs in ms. As expected, the time taken to execute the heuristic was much smaller than for GLPK. We expect the difference to be more pronounced as we increase the number of nodes.

2.4.3.1 AMPL Program

```
 1: set V:={1..n};
 2: set T within V;
 3: set E:=V cross V;
 4: param cost{E};
 5: var x{V,T}, binary;
 6: var y{E}, binary;
 7: minimize objective:
 8: sum{(u,v) in E} cost[u,v] * y[u,v];
 9: subject to
10: mapping_constraint {u in V}:
11:   sum {t in T} x[u,t] = 1;
12: terminal_mapping {u in T}:
13:   x[u,u] = 1;
14: cut_constraint{(u,v) in E, t1 in T,
     t2 in T:t1<>t2}:
15:   y[u,v] - x[u,t1] - x[v,t2] >= -1;
```

Figure 2.19: AMPL program for multiway cut problem

Figure 2.19 is the ILP formulation in AMPL of the multiway cut. Here, V is a set of vertices in stream graph and T is set of terminal vertices in V; $E = V \times V$ is a set of edges/channels; cost : $E \to \mathbb{Q}^+$ is a mapping from an edge to the communication cost (bandwidth requirement) on that edge; $x : V \to T \to \{0,1\}$ maps each vertices to one of the terminals; $y : E \to \{0,1\}$ indicates whether or not the specified edge is a cut edge.

In the program, the objective of ILP problem is to minimise total cost of cut edges as follows:

$$\sum_{(u,v)\in E} cost(u,v) \times y(u,v).$$

This minimisation objective is shown on Line 8. The constraints for ILP are defined on lines $10--15$, where *mapping_constraint* indicates that each node in the graph is mapped to exactly one terminal node; *terminal_mapping* specifies that each terminal node must be mapped to itself; and *cut_constraint* indicates that for each cut edge, (u,v), u and v are assigned to distinct terminal nodes. The constraint encodes a logical relationship between the variables x and y as logic predicates stating that if $x(u,t1)$ and $x(v,t2)$ hold then (u,v) is a cut edge, formally written as

$$\begin{aligned} & x(u,t1) \Rightarrow x(v,t2) \Rightarrow y(u,v) \\ \equiv \quad & x(u,t1) \Rightarrow (\neg x(v,t2) \vee y(u,v)). \end{aligned}$$

We transform this logic formula into an arithmetic one by rewriting implication $\Rightarrow$ as $\geq$, disjunction as addition, and logical negation as arithmetic negation incremented by 1 :

$$\begin{aligned} & x(u,t1) \leq (1 - x(v,t2)) + y(u,v) \\ \equiv \quad & -1 \leq y(u,v) - x(u,t1) - x(v,t2). \end{aligned}$$

The last formula appears on Line 15.

2.4.4 Computation Resource Usage

We compare the number of operators placed in the network between our approach and data forwarding. We ran this experiment using the `Average` query varying the size of the query (the number of `Sense` operators). We applied our placement algorithm on two network models: First model is uniform, where all the nodes start with same maximum energy level; and the second model is non-uniform, where energy levels are set non-uniformly to the nodes, here, a third of the nodes are set with a third of maximum energy level (thus three times more costly to execute an operator than for the remaining sensor nodes using maximum energy level). The number of nodes in the network, however, is the same as the number of `Sense` operators in the query, and hence is proportional to the size of the query. We show the result in Figure 2.20, where we ran the placement algorithm and counted the number of all operators placed in-network

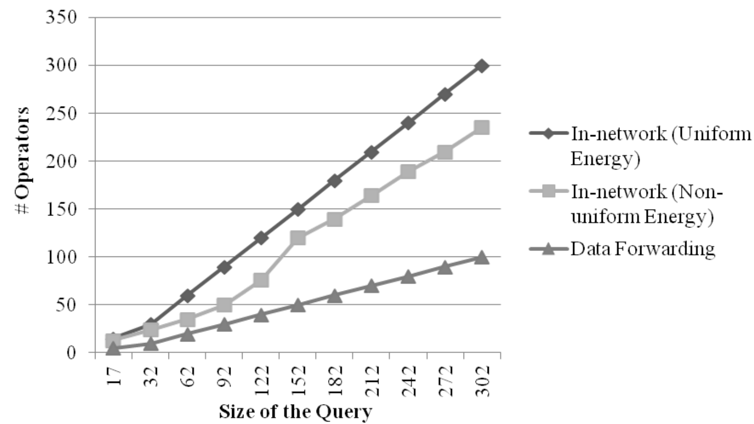


Figure 2.20: Number of operators placed in the network (on the sensor nodes excluding the base station)

(on the sensor nodes excluding the base station). The figure illustrates the computation requirement for data forwarding.

In Figure 2.20, the difference in the number of operators between our approach and data forwarding indicates the difference in the amount of additional memory/computation power required by our approach, compared with that of data forwarding. On average, our approach with network model of non-uniform energy level constantly requires approximately 125% more computation resources from the network. With uniform maximum energy level the approach requires 195% more computation resources. The figure illustrates that our approach with uniform-energy network model requires 31% more computation resources than that of non-uniform energy levels model. The difference in the number of operators between the two in-network approaches is because both the network with non-uniform energy levels has some nodes with smaller available energy, and our algorithm places more operators on the base station.

We compare our result with the previous result in Figure 2.18, where we considered uniform computation costs. Hence, the reduction in energy consumption indicated in the figure corresponds only to the savings in communication costs. At 300 nodes, the cost saving is more than 50%; therefore, when the computation resources are costly, such as when most of the sensor nodes are low on available energy, it is no longer beneficial to perform in-network computation. Intuitively, in such situation, it is best to perform all computations in the base station.

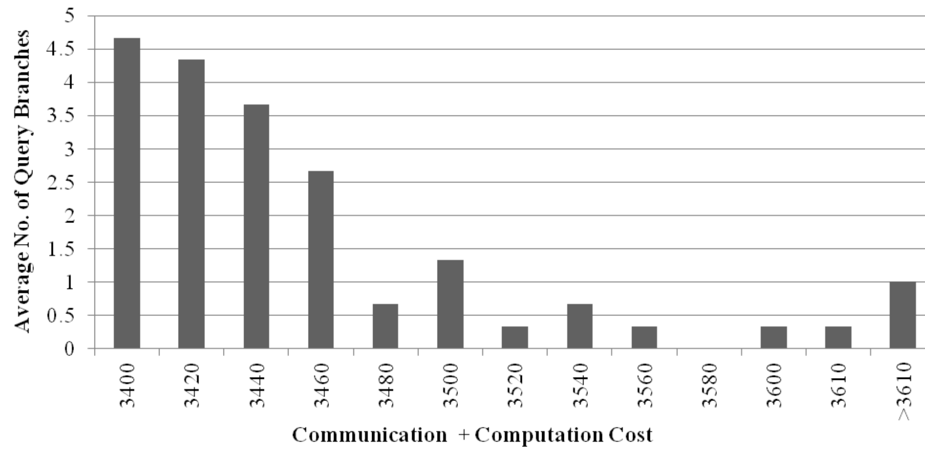


Figure 2.21: Cost distribution over query branches

We similarly employ the information on the number of in-network operators to predict the memory usage, assuming 100 MB memory consumption per operator. Based on SunSPOT specification of 512 MB maximum memory capacity, we computed the radio off-time for implementing the curfew model in the runtime environment (Section 2.2.2).

2.4.5 Cost Distribution

We observed the cost distribution over the sensor network by deploying the `Average` query with the optimised operator placement and computing the total cost over each branch of the query. Each branch represents a communication link between a sensor node and the base station. Where energy levels were uniform among the sensor nodes, all branches have the same cost. However, each sensor node generally has different energy levels that can be queried before placing the operators on the network. Our experiment considered a network with 20 sensors with randomly assigned energy levels. Figure 2.21 demonstrates the cost distribution over query branches. The result was obtained by computing the average distribution of three experiments. The figure shows that most of the query branches run with low communication and computation cost. In other words, our heuristic places the query operators on randomly charged sensor operators in such a way that a major part of the query runs at low cost.

2.5 Chapter Summary

This chapter has described Curracurrong, a stream programming system, for distributed environments, that facilitates in-network computing. Curracurrong consists of a high-level query language and a query processor that translates network-level query to node-level execution environment. We have addressed the programming language challenge for WSNs by allowing application domain experts to develop flexible and complex applications with a simple query language. We employed a novel optimisation algorithm to find energy efficient placements of stream operators to sensor nodes. The experimental results confirm the energy efficiency of the system.

Chapter 3

Curracurrong Cloud

Cloud (or utility) computing [89] has been a huge success to provide elastically adjustable computation resources for large and small enterprises [90]. Much of the large-scale data manipulation and computation in cloud platforms follows the Map-Reduce [91, 92] programming paradigm, which has been the foundation of widely used platforms like Apache Hadoop [93]. The Map-Reduce model is essentially one of batch processing; starting with data in the file system, successive stages of computation transform the data step-by-step to produce the desired output at the end.

For a range of important applications, a different, stream-like, computational model is needed. These applications require the ability to quickly respond to changes to the cluster; this includes cluster monitoring, web-service rate limiting, and dynamic resource provisioning. For instance, a typical use case involves monitoring the system metrics across a cluster of hosts. Vital statistics from each individual node in the cluster – such as CPU utilisation, the number of bytes read and written to and from the disk and network interfaces, the number of page faults – are captured every second. These metrics are sent along a previously defined operator graph that deals with the data in a streaming manner and are acted on accordingly. A typical application would involve monitoring the average CPU utilisation across the cluster and would raise an alert when the average CPU utilisation across the cluster exceeds 95%. This can be used to trigger a provisioning task to add a node in the cluster, thus sharing the load and reduce the average CPU utilisation.

The key characteristic of stream applications is that new data keeps arriving and the computation keeps generating outputs. This style of computation has been studied extensively in smaller-scale environments, as a class of systems called data stream

management systems (DSMS) [94]. In the cloud context, a number of recent examples have been developed. Twitter's Storm system [31] and LinkedIn's Samza [32] are each intended to co-exist with a Hadoop ecosystem; Yahoo! Research has proposed S4 [33], and UC Berkeley offers Spark Streaming [34].

These existing cloud-located streaming platforms have many powerful features. However, there are several restrictions on these approaches. First, since these platforms are intended for data-intensive computation, they separate the computation in a cloud-hosted cluster that is different from the origins of the continual updates to the data. For the monitoring use-case above, there is no need to set up separate clusters within the cloud; instead, we would wish to run the alerting checks on the same nodes that are generating the observations. Second, existing proposals require the developer to write a monitoring pipeline with a great deal of code in Java or a similar language. Storm, for example, uses the Builder pattern extensively, so the computation has to be done by extending classes with a regrettably large amount of code. For the sort of application we are targeting, a simple pipeline ought to be expressible just as a combination of simple operators such as selection, merge, and window aggregation. Third, systems like Storm require explicit deployment: the developer needs to identify which node will perform each aspect of the computation. In contrast, we want the runtime environment to make sensible deployment decisions, to achieve efficiency (especially, to minimise inter-machine data transfer).

To overcome these drawbacks of existing designs, we exploited and made changes to our Curracurrong platform (Chapter 2) to run on cloud environment, which we will refer to as Curracurrong Cloud. Like WSN, a cloud-hosted cluster is an environment where data transmission should be minimised, and where rapid lightweight application coding is valuable.

The following program, written in the Curracurrong Query Language, illustrates the use of Curracurrong Cloud to perform the monitoring task described earlier in this section.

```
query monitor = (Sense[node="10.0.0.1"]
               | Sense[node="10.0.0.2"]
               | Sense[node="10.0.0.3"])
             -> Select[field="5"]
             -> Average[window="1000"]
             -> Threshold[check="$5 > 95"]
```

```
        -> Sink[sink="EmailAlert"]
```

In this chapter:

- We describe Curracurrong Cloud, a lightweight, distributed stream processing system designed for deployment in large distributed clusters hosted on a cloud. Curracurrong Cloud does not ask the programmer to decide on the placement of computation among nodes of the cluster, but rather automates the deployment decisions.
- We provide a preliminary evaluation of the system.

The chapter is organised in two sections. The first shows the changes made in Curracurrong system to run on a cloud environment. In the second, we present experimental results to demonstrate the efficiency and scalability of the system, especially of its automated placement decisions.

3.1 Curracurrong Cloud

This section describes the changes we made to the existing Curracurrong system so that it runs in a cloud-hosted cluster, where the computational resources are much less constrained compared with WSNs. We originally built the Curracurrong system using Java Micro Edition and targeted it towards the Squawk virtual machine that is designed for tiny embedded devices. However, the new system was implemented to run on Java Standard Edition so that it can be deployed in a cloud environment. We now describe the changes made to the code to achieve this.

3.1.1 ZigBee vs. UDP

The ZigBee *Radiogram* protocol is the default communication protocol used between SPOT nodes in a WSN. Curracurrong uses this protocol for data communication among the nodes in the network and with the base station. ZigBee provides a reliable, buffered, connectionless input/output communication mechanism between nodes. Since this means of communication is connectionless, it requires buffering and application-level acknowledgements to implement successful communication. This is further complicated by the fact that individual nodes have the capability to shutdown their communicating device

to conserve power. This is managed using node-level buffering in combination with a store-and-forward mechanism. A SPOT client connects to the server as follows:

```
RadiogramConnection conn = (RadiogramConnection)
             Connector.open("Radiogram://1.1.1.1:100");
```

where, *1.1.1.1* is the node address and *100* is the port number.

In the Curracurrong Cloud, we retain the *connectionless* nature of the communication between the nodes. This allows the system to deal with temporary node failures and network outages more easily. In the current version, we replaced the ZigBee protocol with the User Datagram Protocol (UDP) for communication between nodes in a cluster. This means that there is no guarantee of data delivery, ordering, or duplicate protection. This is similar in behaviour to the ZigBee Radiogram protocol, making UDP a natural replacement in the new deployment platform.

There is nothing in the code that prevents the use of any other communication protocol between the nodes. If a more reliable form of communication is needed, UDP can be replaced with Transmission Control Protocol (TCP/IP) without significant rewriting. The choice of UDP versus TCP is purely for the sake of simplicity and similarity with the ZigBee Radiogram protocol. A snippet of sample code for the datagram socket programming is given below.

```
DatagramSocket socket = new DatagramSocket();
DatagramPacket packet = new DatagramPacket(
                 buffer, offset, destination_address,
                 port_number);
```

3.1.2 Broadcast and Multicast

In a WSN environment, Curracurrong is able to leverage the ZigBee Radiogram protocol's inherent broadcast and multicast capability; this simplifies the task of communication across the nodes in the network. The base station sends the routing table and other administrative commands over broadcast or multicast.

Broadcast or multicast over UDP is not a viable option for communication in a public cloud. Amazon Web Services, for instance, blocks all broadcast and multicast UDP datagrams, even when a Virtual Private Cloud is defined. This behaviour is due to

the immense complexities involved in efficiently implementing it in a multi-tenant IaaS environment, where there are constant updates to routing tables.

We used multiple unicast operations to simulate broadcast in current version of the Curracurrong Cloud to work around this restriction. A simple way to manage and implement this is to define a CNAME that contains IP address entries for all the nodes in the cluster. This allowed ease of dynamically adding and deleting entries in the cluster. The task of broadcasting or multicasting messages to the entire cluster, then, consisted of sending a unicast message to each address defined by the CNAME.

3.1.3 Scheduler

The scheduler component was re-implemented from the previous version. The existing Curracurrong system for WSN inserts all the operators in a wait queue until they have data to be processed in their input channels. An operator ready to be executed is removed from the wait queue and added to the ready queue before it can be executed. Just one independent scheduler thread schedules and executes operators atomically.

The modification to the scheduler enables it to more efficiently handle both event- and time-triggered operators simultaneously. The earlier design was focused on efficiently managing power use in sensor nodes; however, power consumption is less of an issue in the cloud, and reducing message latency and network bandwidth use is a higher priority.

To address this, we made two major changes to the scheduler. Each event-triggered operator is executed in its owns thread, and blocks execution when there are no data in its input channels. When data are written to a channel, the operator is woken and its execute() method is called in the thread of control asynchronously, independently of the scheduler thread.

On the other hand, time-triggered operators are executed in the context of the scheduler thread. This allows the scheduler to precisely control the execution of each time-triggered operator, ensuring that its execute() method is executed at regular intervals. This dual scheduling mechanism ensures that time-triggered operators are executed precisely at regular time intervals as specified in the query, and that event-triggered operators do not hinder the execution of time-triggered operators by ensuring that each is executed in a separate thread.

3.1.4 Operators

Curracurrong Cloud extends the built-in operator library to expand the functionality of the system. The existing Curracurrong system for WSN only allows time-triggered sense operators that operate at user-specified time intervals. By adapting sense operator design, we can change the scheduling semantics to allow both time- and event-triggered sensors. This new functionality enables the user to process only data of interest. In Curracurrong for WSN, the operator library contained one default time-triggered sense operator, while with adaptation we have introduced multiple sensor operators into the library.

In the Curracurrong system for WSN, sense operators always operated in time-triggered scheduling mode. For the cloud version, we extended this capability to allow the scheduling mode to be set in the query definition via the *sched* node property. If the scheduling mode is not set for a sense operator, it defaults to time-triggered mode. In the event-triggered mode, the sense operator executes only when there is an event to be processed. For instance, in case of an error log file sensor, the sensor blocks until an error log entry is written to the log file.

In addition, we extended the sense operator to enable the sensor to be set via the *sensor* property. The sensor parameter takes a valid class name derived from the Java class *Sensor*. An example of the error log file sensor is called the *ErrorLogSensor*. The following specifies the way to define a sense operator with the properties defining the scheduling semantics and the sensor class name.

```
Sense[sensor=<sensor_name>,
    sched="EventTriggered"|"TimeTriggered",
    node=<address>]
```

Like the sense operator, we extended the functionality of the sink operator. The new sink operator can load any user defined class that is derived from the *Sink* class if specified. The default sink still dumps the received data to a log file, but can be overridden using the *sink* property as follows:

```
Sink[sink=<sink_name>, email=<email_address>,
   db=<DB_connection_string>, logfile=<file_name>]
```

Apart from the default sink operator that logs the end data in a file, we have introduced Sink classes that can be loaded from sink operators to allow the user to store

the data in a user-specified database (*DBsink*), send an alert email to a registered email address (*EmailSink*), and write the result to a remote machine (*RemoteSink*).

Similarly, we added operators that perform useful tasks in a stream graph. For instance, we introduced a *Skyline* operator to the library that records the maximum and minimum values for the data during a specified time or message window, and passes the envelope values to the next operator. The Skyline operator is represented as follows:

```
Skyline[type="message"|"time", window=<size>]
```

The operator can be used to detect events to trigger provisioning events in a cluster, when the maximum envelope for a sustained period may trigger the launching of new virtual machines (VMs). A continuous period of low utilisation, where maximum values are below a threshold could trigger a scale down effectively shutting down instances in the cluster.

3.2 Evaluation

For evaluation, we deployed the Curracurrong Cloud runtime environment on a cloud-hosted cluster to evaluate its performance. The nodes and the server components of the system were deployed on a cluster using Amazon Web Services (AWS) and Elastic Compute Cloud (EC2). We used Virtual Private Cloud (VPC) to easily define our cluster firewall rules. Out of the six EC2 instances, one instance had Curracurrong Cloud server running on it, while the others had Curracurrong Cloud nodes on them. Due to the unreliability of communication over UDP, we evaluated the current version of the system with six instances, to answer following questions:

- How efficient is our automated operator placement mechanism?
- Does the network load affect the latency of data records?
- How does the scheduling semantic affect the message latency and network bandwidth?

3.2.1 Operator Placement Algorithm Efficiency

Network communication cost is typically higher than computational cost in most network applications. Curracurrong strives to move computation to the edge nodes to

reduce the network traffic. We devised a simple experiment to calculate the unoptimised approach: the raw data is sent to the server for further computation against an optimised approach in which the computation is performed in the network where the operator scheme pushes computation to the edge nodes.

We used the following query to evaluate the efficiency of the algorithm:

```
query place = Sense[node=<address>] ->
   Select[field=<list_of_fields>] ->
   Sink[logfile=<file_name>]
```

The query *place* induces a simple in-network computation, where the optimisation algorithm allocates the select operator on the edge node instead of the server. With optimised placement, the number of packets transferred between node and server is reduced by the number of fields discarded by the Select operator. Figure 3.1 shows the comparison between default data forwarding and in-network computation. The in-network computation reduces the amount of network data by almost 40% compared with default placement.

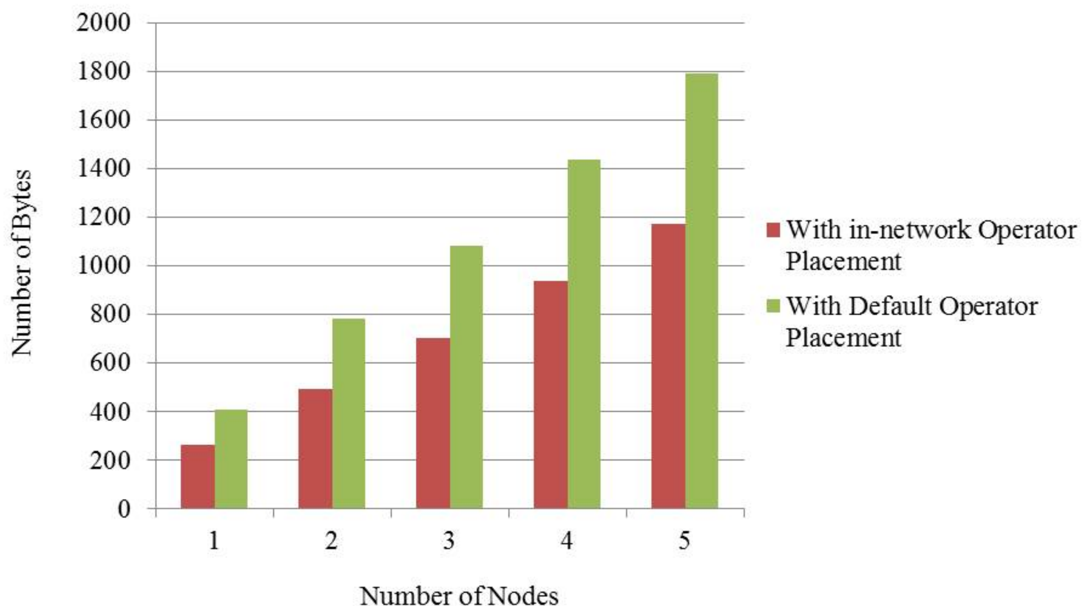


Figure 3.1: Network data transfer with operator placement

We compared the runtime of the in-network placement algorithm with the default placement. Figure 3.2 shows the efficiency of the placement algorithm. As expected,

the time taken by the default placement approach is uniform and nearly 0.3 ms, whereas the time taken by the in-network placement heuristic increases with the number of nodes but appears to be typically less than 2 ms for five nodes. The advantage of optimised network usage may allow us to dispense with milliseconds of algorithm runtime.

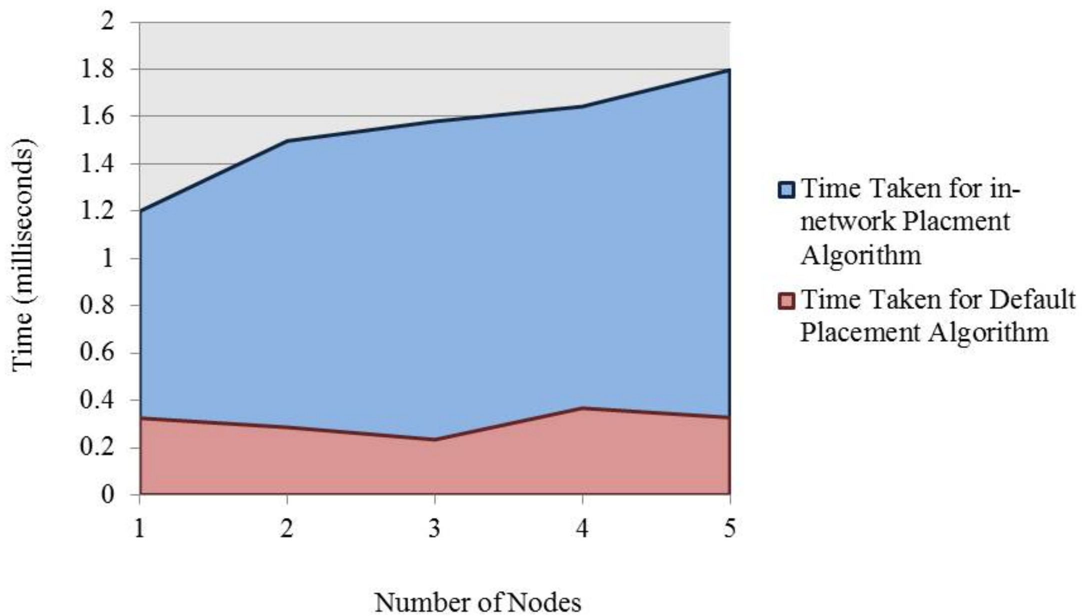


Figure 3.2: Placement algorithm efficiency

3.2.2 Measuring Message Latency

To measure latency, we monitored the health of the cloud-hosted cluster at predefined intervals and send an alert to the server if a particular system parameter reaches a threshold. For measuring the latency of an alert message, we computed the difference between the time at which a cluster instance collects system parameters and the time at which the corresponding alert message is received at the sink/server. We ran the following query to measure the latency and observe the effect of system/network load on it.

```
query threshold = Sense[node=<address>]
               -> Threshold["$7 > 95"]
               -> Sink[logfile=<file_name>]
```

We executed *threshold* queries with five cloud-hosted instances, where each instance ran a Sense operator at regular intervals of 5 s, and collects system parameters

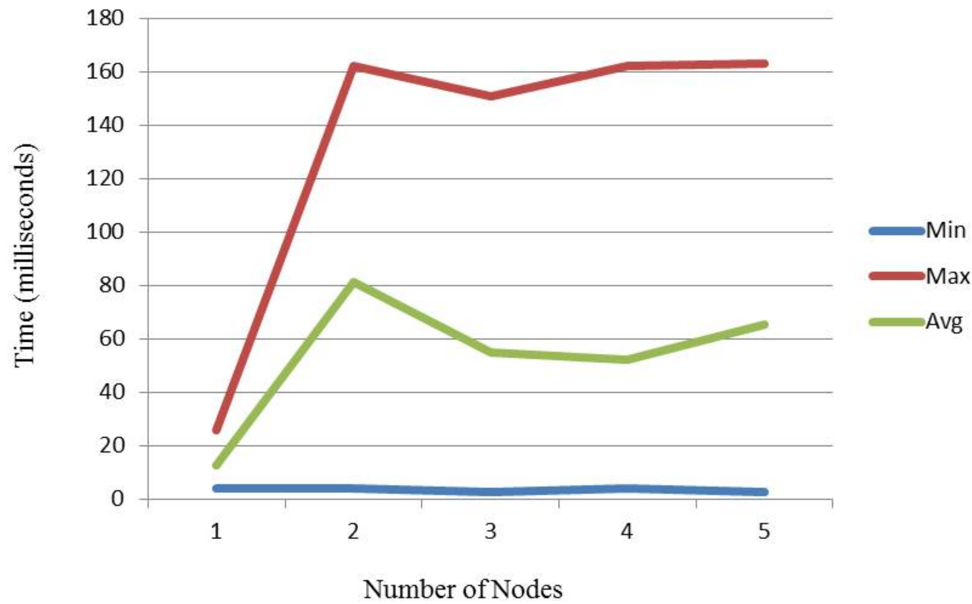


Figure 3.3: Effect on latency while varying the network load

such as CPU usage, disk write, disk read, and number of bytes/packets sent over the network. In the experiment, we monitored CPU usage and checked whether its usage exceeded 95%. In the query, Threshold operator was set to check the seventh field of the incoming sensed data. We varied the system load and gradually increased the CPU usage of each instance to more than 95%. When the measured value reached the threshold, it sent an alert to the sink operator and logged them in a file along with the local time.

As shown in Fig. 3.3, the effect of system/network load on the latency is in milliseconds. With one node, the average latency is 10 ms, which increases with the number of nodes. On average the delay is nearly 60 ms, with all five nodes sending alerts to the sink. The result ensures the scalability of our system, where alerts are sent to the server within few milliseconds irrespective of the network load.

3.2.3 Time-triggered vs. Event-triggered

The Curracurrong Cloud supports operators with two scheduling semantics: time- and event-triggered, as described in Section 3.1. This section evaluates the effect of scheduling semantics of the Sense operators on the latency and network bandwidth while running the following time-triggered query:

```
query time-t = Sense[sensor="ErrorFileSensor",
                  file=<error-file>,
                  node=<address>,
                  sched="TimeTriggered",
                  interval=<time_in_milliseconds>]
            -> Sink[logfile=<file-name>]
```

and the following event-triggered query:

```
query event-t = Sense[sensor="ErrorFileSensor",
                   file=<error-file>,
                   node=<address>,
                   sched="EventTriggered"]
             -> Sink[logfile=<file-name>]
```

In these queries, the ErrorFileSensor operator reads from an error log file and generates a message for each new entry in the file. In time-triggered mode, the Sense operator checks the error log file for new log messages at the specified interval. In event-triggered mode, the Sense operator reads the message as soon as the error log file has a new entry. A simple shell script with a *while loop* sleeps for 1 s and writes the current system timestamp to the error log file.

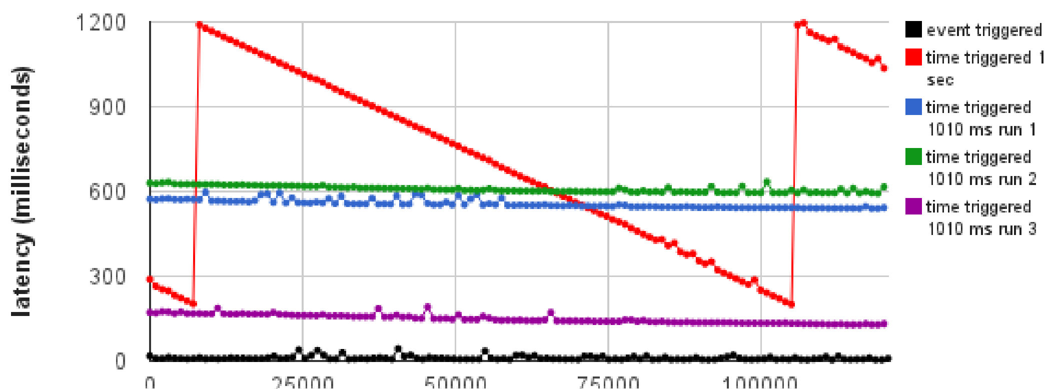


Figure 3.4: Effect of the sense time interval setting on time-triggered Sensors

Figure 3.4 shows the impact of latency of the message from when it is logged to the error log file to when it is received at the Sink. In the event triggered mode, the latency of the messages is well within 120 ms. In time-triggered mode, the latency shows a

saw-tooth pattern where the peaks are at a regular intervals corresponding to the time interval setting. The messages are read only when the operator is scheduled, giving rise to the characteristic pattern. The latency in the event-triggered mode does not display such a pattern. In time-triggered mode with sense interval setting of 1 s, rather than a reasonably flat line, we see a longer saw-tooth pattern.

Figure 3.4 shows three separate runs of 120 ms, where the sense time interval is 1010 ms. The latencies are relatively stable throughout. Each run displays a different latency between a few milliseconds to just over 1 s. This saw-tooth pattern occurs because the shell script takes 1010 ms to run. When the Sensor sense interval is exactly 1 s, the Sensor is just out of sync by approximately –10 ms causing the latency to drop slowly and then rise sharply in the characteristic saw-tooth pattern.

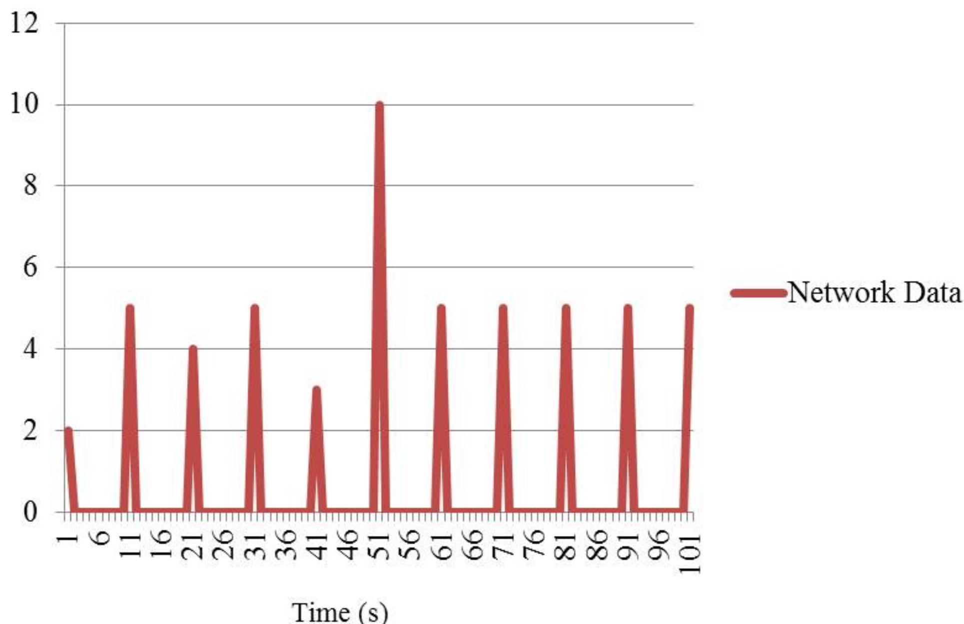


Figure 3.5: Network data with time-triggered sensors

Next, we compare the network bandwidth usage for both scheduling semantics. Figure 3.5 displays the network bandwidth while using the time-triggered ErrorFileSensor. The number of packets transferred are plotted over the network against time. Here, sensor time interval is set to 10 s, and error file entry is simulated at every 2 s. The sense operator queues up the sensed data locally at the node for 10 s – illustrated in the figure with no network activity during that interval. Otherwise, the network data shows a spike at every 10th second, when the sense operator sends the buffered data out over

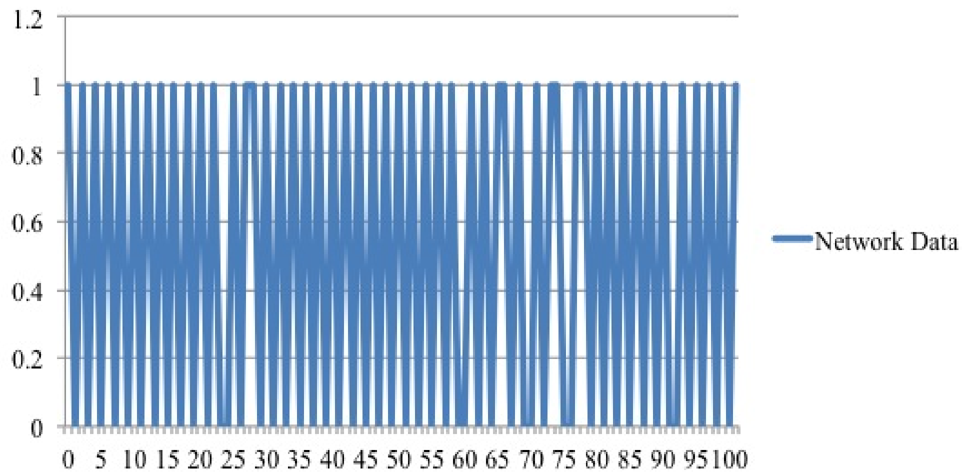


Figure 3.6: Network data with event-triggered sensors

the network.

We then ran the ErrorFileSensor in a event-triggered mode and plotted the network data as displayed in Fig. 3.6. The data was sent over the network as soon as it was available, which is shown as a spike every 2 s. The network is continuously busy and the bandwidth is not well-used. This scheduling semantics can be used for time-critical applications; otherwise time-triggered semantic is a better option for optimised network bandwidth usage.

3.3 Chapter Summary

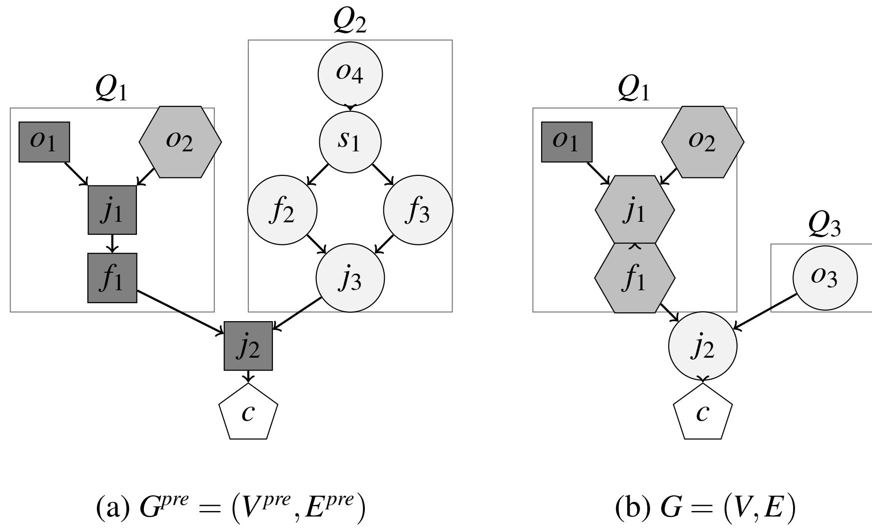
This chapter has provided a detailed description of the Curracurrong Cloud, a lightweight, distributed stream processing system designed for large distributed clusters hosted on clouds. Our changes in system design and technology enable Curracurrong to run on a cloud-based cluster of nodes instead of an ad-hoc cluster of wireless sensor nodes. We developed a simple cluster monitoring application to demonstrate the technology and used parts of it to evaluate the performance characteristics of our system. We analysed the results of the evaluations.

Chapter 4

Migrating Operator Placement

In sensing applications, queries need to be dynamically replaced on the nodes due to the changes in nodal energy level and in the network configurations. This chapter proposes a method for energy-efficient query execution in relation to operator migration. We introduce the *migrating operator placement problem* (*MOPP*), which arises in applications such as mobile cloud computing [95, 96] and wireless sensor networks [50, 97, 49]. The queries are represented as stream graph using stream programming model similar to MaD-WiSe [30] and query processing [97]. The stream programming model (surveyed in [98]) expresses computations by independent computational units called *operators* and communication *channels* between them. In our context, the stream graph consists of operators that sense the environment and operators that process and forward the information to a base station. To minimise the energy cost of the system, we placed operators on nodes so that the associated operator placement costs become minimal. The costs comprised (1) placement of an operator to a specific sensor node, (2) sending data between two connected operators, and (3) transition incurred by migrating operators from its current node to a new node. The operator placement can be conducted either when the stream graph representing queries changes, the distributed system changes, and/or the energy levels of the nodes change after a fixed time interval.

As an example, consider a volcanic activity detection application that deploys several sensors in the proximity of the volcano and measures temperature, gas, and seismic activity. In an initial step, the application executes two queries: The first query collects the temperature reading from the sensors placed at distant locations and computes the average temperature. The second query collects the reading from seismometer, computes the high- and low-gain moving averages, and reports an event if the ratio of the



(a) $G^{pre} = (V^{pre}, E^{pre})$ (b) $G = (V, E)$

Figure 4.1: Running example. (a) Stream graph G^{pre} for queries Q_1 and Q_2 with placement p^{pre}; (b) Stream graph G for query Q_1, newly added query Q_3, deleted query Q_2, and new placement p.

two averages is beyond threshold. The results from the two queries are merged to detect the volcanic activity as illustrated in Fig. 4.1 (a), which shows the stream graph of the queries. Subgraph Q_1 represents the temperature-reading query with two sense operations o_1 and o_2, a join operation j_1 to merge the data from the two sensors, and a filter f_1 to compute the average temperature. The second subgraph Q_2 represents the seismic query with a sense operation o_4 that reads the data from a seismometer, a split operator s_1 that duplicates the incoming data to compute the high- and low-gain exponentially weighted moving average (EWMA) [11] using filters f_2 and f_3, and a joiner j_3 that merges the data from two filters to compute the ratio. Finally, a joiner j_2 combines the data from both the queries Q_1 and Q_2 before forwarding it to the sink c. In Fig. 4.1 (a), operator placement is shown by the shapes of the graph nodes; i.e., rectangle, hexagon, circle, and pentagon denote the placement of operators to sensor nodes n_1, n_2, n_4 and base station b, respectively.

In our volcanic activity detection application, sensor nodes might be destroyed or become temporarily unavailable. Hence, the migration of operators from inoperable to operable sensor nodes is of paramount importance. In addition, the application may require the alteration of queries: e.g., the observation from seismic event detection to the measurement of expelled gases. For this example, the altered stream graph is shown

Op\Node	n_1	n_2	n_3	b
o_1	0	∞	∞	∞
o_2	∞	0	∞	∞
j_1	10	6	13	20
f_1	7	4	15	17
o_3	∞	∞	∞	∞
j_2	9	12	11	13
c	∞	∞	∞	0

(a) Placement cost for Q_1 and Q_3

Op\Op	o_1	o_2	j_1	f_1	o_3	j_2	c
o_1	0	0	5	0	0	0	0
o_2		0	13	0	0	0	0
j_1			0	21	0	0	0
f_1				0	0	7	0
o_3					0	10	0
j_2						0	11

(b) Communication cost for Q_1 and Q_3

Op\Node	n_1	n_2	n_3	b
j_1	0	10	16	21
f_1	0	11	13	19
j_2	0	8	5	10

(c) Transition cost for Q_1

Figure 4.2: Incurred costs in operator placement

in Fig. 4.1 (b), which contains new query Q_3, and from which query Q_2 of the original graph was removed. For the operator placement of the altered stream graph, we take placement cost, communication cost, and transition costs into account that are tabularised in Fig. 4.2. The placement of sense operators o_1, o_2, o_3, and the sink c are not arbitrary, and are fixed to nodes n_1, n_2, n_3, and base station b, respectively. The placement of the other operators is chosen so that energy costs of the placement are minimised. While recomputing the placement for modified stream graph, the difference of placement between the old and modified graphs incurs *transition* cost. In our example, query Q_1 carries transition cost for the operators j_1, f_1, and j_2, since they are placed on different sensor nodes in the old and new stream graphs (Fig. 4.1 (a) and Fig. 4.1 (b), respectively).

The contributions of this chapter are as follows.

- We introduce the NP-hard *migrating operator placement problem* (*MOPP*) that computes a placement and minimises computation, communication, and transition costs.

- We devise a dynamic program that computes an optimal operator placement for *compositional* streams which is a sub-class of general stream graphs in polynomial time.

- To improve the performance of our approach, we introduce a locality heuristic that, at any step in the algorithm, only considers the placement onto a subset of the network close to the base station.
- We present an experimental evaluation confirming the efficiency of our approach and its effectiveness in reducing energy usage.

The structure of the chapter is as follows. Section 4.1 explains the problem statement as a discrete optimisation problem, and argues that the general problem is NP-hard. Section 4.2 details the graph composites. Section 4.3 details the dynamic program for compositional streams and its correctness and complexity proofs. We present an experimental evaluation that confirm the efficacy of our approach, together with locality heuristics that improve its efficiency in Section4.4.

4.1 Migrating Operator Placement Problem

We introduce the MOPP. At any one time in the program execution, there are a number of active queries, so that the whole program can be represented as a stream graph $G = (V,E)$, with $V = V_{se} \cup V_{si} \cup V_{fi}$, with V_{se} a set of *sense* operators, V_{si} a singleton of a *sink* operator, and V_{fi} a set of *filters*, that performs computation on its input and sends the result to the output. V_{se}, V_{si}, and V_{fi} are pairwise disjoint, and $E \subseteq V^2$. We are given a set N of sensor nodes with a distinguished element b called the *base station*. We introduce a *placement* $x : V \to N$ where x_u is a variable denoting the sensor on which $u \in V$ is placed. We write X for an unordered sequence of such variables, and X_G for a restriction of the sequence to placements of all nodes in V [1]. The placement x is constrained so that $x_u \neq x_v$ when $u \neq v$ and $u,v \in V_{se}$ (assuming $|N|$ is sufficiently large), and $x_u = b$ for $u \in V_{si}$.

We assume there is a change in the graph such as the addition and removal of queries, such as we had a *pre-existing* graph $G^{pre} = (V^{pre}, E^{pre})$ that transformed into $G = (V,E)$. We denote the *pre-existing* placement $x^{pre} : V^{pre} \to N$, with variables x_u^{pre} for $u \in V^{pre}$ denoting $x^{pre}(u)$. The problem of MOPP is to find the placement x for the new graph $G = (V,E)$ with the following three costs minimised.

[1] A variable x_u for some u can appear more than once in the sequence, therefore for sequences X_1 and X_2, $X_1 = X_2$ modulo reordering and repetition. $X_1 \setminus X_2$ removes all occurrences of x_u from the placement sequence X_1 for any $x_u \in X_2$. $X_1 \cup X_2$ concatenates both sequences.

Placement cost: This is the local cost incurred by placing an operator on a given sensor node. The cost depends on the available energy level on sensor node and computational load for executing the operator. For an operator $u \in V$ and the placement x, the placement cost is $w_P(u, x_u) \in \mathbb{Q}^+$.

Communication cost: This is associated with each channel. For a channel $(u, v) \in E$, the cost is a product of two values: $w_C(u, v) \in \mathbb{Q}^+$ denoting the bandwidth requirement of the channel, and $w_D(x_u, x_v) \in \mathbb{Q}^+$ denoting the *distance factor* between the allocated nodes. By knowing the maximum distance factor, it is possible to incorporate it into $w_C(u, v)$ and allowing $w_D(x_u, x_v) = 1$ when $x_u \neq x_v$ and 0 otherwise, as assumed in Fig. 4.2 (b).

Transition cost: This is the cost incurred while moving an operator to a new sensor node. In other words, if an operator u is to be relocated to other sensor node, the transition cost is $w_T(u, x_u^{pre}, x_u) \in \mathbb{Q}^+$.

The following function encapsulates both placement and transition costs.

$$w_{TP}(u, x_u) = \begin{cases} w_P(u, x_u) + w_T(u, x_u^{pre}, x_u) & \text{if } u \in V^{pre} \cap V, \\ w_P(u, x_u) & \text{otherwise.} \end{cases}$$

Here we assume that the costs for removing or installing queries onto the network are insignificant, where the nodes may already include the program or library functions to execute the operators. Although it may seem at first that it is less expensive to remove the queries entirely from the network and reinstall them rather than transitioning the operators, here we assume a need to preserve the existing computation and to pay the associated costs. The transition costs arise from transferring computation along network nodes from source to destination. In wireless sensors, this movement is costly [99].

The cost $f_G(X_G)$ of MOPP to be minimised, given a stream graph $G = (V, E)$, is the sum of computation and transition costs for each operator, and the communication costs are:

$$f_G(X_G) = \sum_{u \in V} w_{TP}(u, x_u) + \sum_{(u,v) \in E} w_C(u, v) \cdot w_D(x_u, x_v) \tag{4.1}$$

Theorem 1. *MOPP is NP-hard.*

Proof. We show the NP-hardness by reducing the multiway cut problem (see [82]) to MOPP. The input of a multiway cut problem is an undirected graph $G^{MC} = (V^{MC}, E^{MC})$

and a terminal set $T = \{t_1, \ldots, t_l\} \subseteq V^{MC}$. A multiway cut is a partition of V^{MC} into disjoint sets $R_1, \ldots, R_l$ $\left(\bigcup_i R_i = V^{MC}\right)$ such that terminal $t_i \in T$ is contained in R_i for all i. We denote by $\delta(R_i) = |(E^{MC} \cap R_i) \times (V^{MC} \setminus R_i)|$ the number of cut-edges that separate the disjoint set R_i from nodes outside the set. The goal of the multiway cut is to minimise the number of cut-edges $\frac{1}{2}\sum_i \delta(R_i)$.

We map multiway cut to an instance of MOPP such that the vertex set V^{MC} becomes set V, the edges E^{MC} become the channels E, and the terminals T correspond to the set of sensor nodes N, i.e., $N = V_{si} \cup V_{se}$. For MOPP, we assume in addition that the placement of the operators is unchanged such that transition costs are zero, and w_{TP} represents placement cost as follows.

$$w_{TP}(u,t_i) = \begin{cases} \infty, & \text{if } u \in T \wedge t_i \neq u \\ 0, & \text{otherwise,} \end{cases}$$

Note that the mapping $p : V \rightarrow N$ partitions the domain V, i.e., $R_i = \{u | x_u = t_i\}$ for all i, and $\bigcup_i R_i = V$. We also assume that for any $u, v \in V$, $w_C(u,v) = 1$. Further, $w_D(x_u,x_v) = 1$ when $x_u \neq x_v$ and 0 otherwise. By rewriting the objective of the *MOPP*, we obtain:

$$\begin{aligned} f_G(X_G) &= \sum_{u \in V} w_{TP}(u,x_u) + \sum_{(u,v) \in E} w_C(u,v) \cdot w_D(x_u,x_v) \\ &= \sum_i \begin{cases} \infty, & \text{if } x_{t_i} \neq t_i \\ 0, & \text{otherwise} \end{cases} + \sum_{(u,v) \in E} \begin{cases} 1, & \text{if } x_u \neq x_v \\ 0, & \text{otherwise} \end{cases} \end{aligned}$$

The first summand is a dualised constraint $x_{t_i} = t_i$; that is, the function $f_G(X_G)$ is less than ∞ if $x_{t_i} = t_i$ for all i. Since $x_{t_i} = t_i$, the terminal t_i is contained in R_i, and:

$$\begin{aligned} \sum_{(u,v) \in E} \begin{cases} 1, & \text{if } x_u \neq x_v \\ 0, & \text{otherwise} \end{cases} &= \sum_{(u,v) \in E} (\exists i : (u,v) \in R_i \times (V \setminus R_i)) \\ &= \sum_i |(E \cap R_i) \times (V \setminus R_i)| = \sum_i \delta_i(R_i). \end{aligned}$$

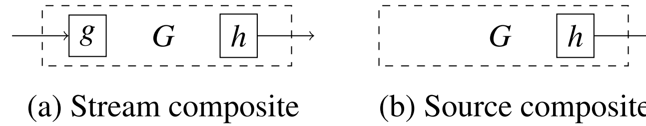
This shows the reduction of multiway cut problem to MOPP and hence, MOPP is NP-hard.

□

4.2 Graph Composites

Here we introduce a class of stream graphs called *compositional* streams that are constructed from building blocks called *composites* (cf. StreamIt [71]). Although MOPP is generally NP-hard, compositional streams can be solved in polynomial time.

There are two types of composites: *stream* composites, which are graphs $G = (V,E,g,h)$ with $g = s(G)$ and $h = t(G)$ are respectively *input* and *output* operators, and *source* composites, graphs $G = (V,E,h)$, with $h = t(G) \in V$ as its output operator. A source composite G has no input operator – that is, $s(G) \notin V$. The templates for the composites are as follows.



(a) Stream composite (b) Source composite

Fig. 5.2 displays the types of stream and source composites. A stream composite is either (a) a filter or sink operator, (b) a *pipeline* that joins the output of a stream composite with the input of another stream composite, or (c) a *split-join* composite that splits an input into the inputs of two stream composites and combines the outputs of the streams. A source composite is either (d) a sense operator, (e) a pipeline joining the output of a source composite with the input of a stream composite, (f) a *join* composite joining the outputs of two source composites into one, or (g) a program (a pipeline that joins the output of a graph with a base station). The split-join and the join composites are constructed using special operators called *splitters* and *joiners*, which respectively split the tokens of a communication channel and join them.

We define stream composite class $\mathbb{S}$ as follows.

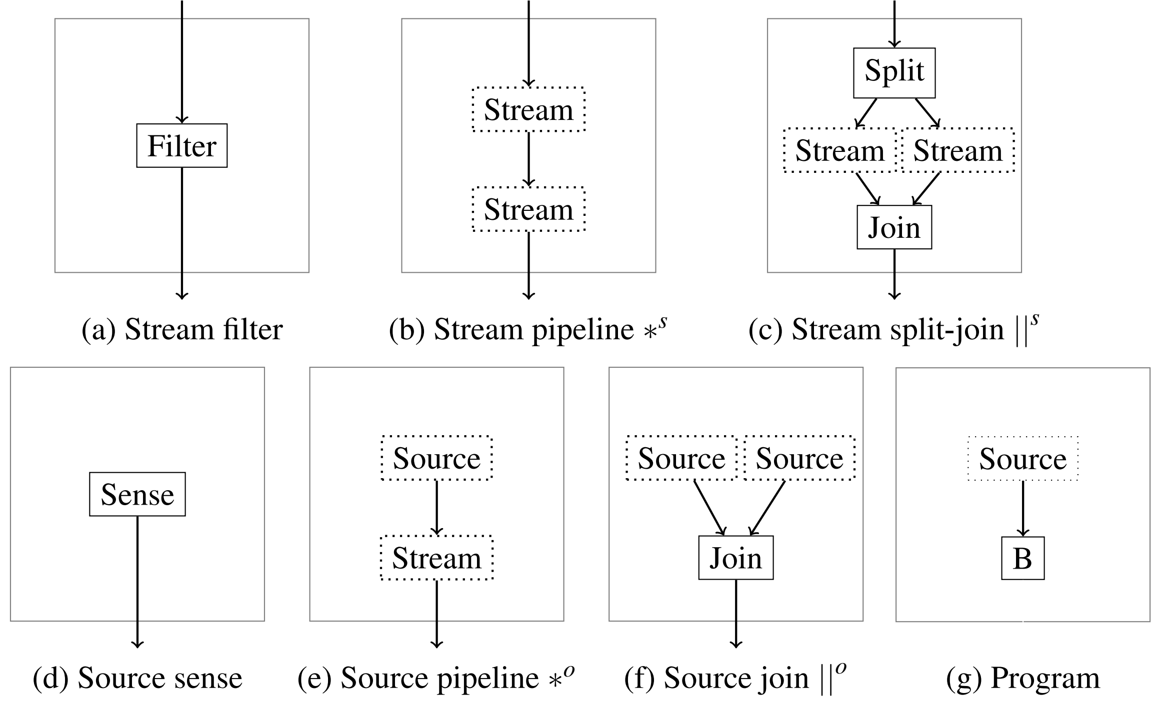
(*R1*) If $V = V_{fi} = \{u\}$ or $V = V_{si} = \{u\}$ then $G_u = (\{u\},\emptyset,u,u) \in \mathbb{S}$.

(*R2*) If $G_1 = (V_1,E_1,g_1,h_1) \in \mathbb{S}$ and $G_2 = (V_2,E_2,g_2,h_2) \in \mathbb{S}$, and $V_1 \cap V_2 = \emptyset$, then the pipeline $G_1 *^s G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup \{(h_1,g_2)\}, g_1, h_2) \in \mathbb{S}$.

(*R3*) If $G_1 = (V_1,E_1,g_1,h_1) \in \mathbb{S}$ and $G_2 = (V_2,E_2,g_2,h_2) \in \mathbb{S}$, and $V_1 \cap V_2 = \emptyset$, then the split-join composite $G_1 ||^s G_2 = (V_1 \cup V_2 \cup \{s, j\}, E_1 \cup E_2 \cup \{(s,g_1), (s,g_2), (h_1, j), (h_2, j)\}, s, j) \in \mathbb{S}$, where $s, j \notin V_1 \cup V_2$ are splitter and joiner respectively.

(*R4*) Nothing else is in $\mathbb{S}$.

We define source composite class $\mathbb{O}$ as follows.



(a) Stream filter (b) Stream pipeline $*^s$ (c) Stream split-join $||^s$

(d) Source sense (e) Source pipeline $*^o$ (f) Source join $||^o$ (g) Program

Figure 4.3: Composites

($R5$) When $V = V_{se} = \{u\}$, then $G_u = (\{u\}, \emptyset, u) \in \mathbb{O}$.

($R6$) If $G_1 = (V_1, E_1, h_1) \in \mathbb{O}$ and $G_2 = (V_2, E_2, g_2, h_2) \in \mathbb{S}$, and $V_1 \cap V_2 = \emptyset$ the pipeline $G_1 *^o G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup \{(h_1, g_2)\}, h_2) \in \mathbb{O}$.

($R7$) If $G_1 = (V_1, E_1, h_1)$ and $G_2 = (V_2, E_2, h_2)$ are in $\mathbb{O}$, and $V_1 \cap V_2 = \emptyset$, then $G_1 ||^o G_2 = (V_1 \cup V_2 \cup \{j\}, E_1 \cup E_2 \cup \{(h_1, j),\ (h_2, j)\}, j) \in \mathbb{O}$, where $j \notin V_1 \cup V_2$ denotes a joiner.

($R8$) Nothing else is in $\mathbb{O}$.

A *program* is a composite satisfying ($R6$), but with $G_2 = G_u$ and $u \in V_{si}$. We now define the function *sub* that returns the set of immediate children of a composite G, called its *substreams*, as follows.

$$sub(G) = \begin{cases} \{G_1, G_2\} & \text{if } G = G_1 *^o G_2 \text{ or } G = G_1 *^s G_2, \\ \{G_j, G_1, G_2\} & \text{if } G = G_1 ||^o G_2 = (V, E, j), \\ \{G_s, G_j, G_1, G_2\} & \text{if } G = G_1 ||^s G_2 = (V, E, s, j), \text{ and} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

4.3 Dynamic Programming for MOPP

4.3.1 Dynamic Program

We define a dynamic program that computes the cost of allocating query operators to available sensors in the network. The program is defined using two functions $\hat{f}_G^{\mathbb{S}} : N^2 \to \mathbb{Q}^+$ and $\hat{f}_G^{\mathbb{O}} : N \to \mathbb{Q}^+$ depending on whether G is a stream or source composite. The arguments of $\hat{f}_G^{\mathbb{S}}$ are the sensors to which the input and output vertices of G are respectively placed, whereas the argument of $\hat{f}_G^{\mathbb{O}}$ is the placement of the output vertex of G.

Definition 1 (Dynamic Program $\hat{f}_G^{\mathbb{S}}$ and $\hat{f}_G^{\mathbb{O}}$)**.**

1. $G \in \mathbb{S}$.

- *If* $G = G_u$ *with* $V = V_{fi} = \{u\}$ *then* $s(G) = t(G) = u$ *and*

$$\hat{f}_G^{\mathbb{S}}(x_{s(G)}, x_{t(G)}) = \begin{cases} w_{TP}(u, x_u) & \text{if } x_{s(G)} = x_{t(G)}, \\ \infty & \text{otherwise.} \end{cases}$$

- *If* $G = G_u$ *with* $V = V_{si} = \{u\}$, *then* $s(G) = t(G) = u$ *and*

$$\hat{f}_G^{\mathbb{S}}(x_{s(G)}, x_{t(G)}) = \begin{cases} w_{TP}(u, x_u) & \text{if } x_{s(G)} = x_{t(G)} = b, \\ \infty & \text{otherwise.} \end{cases}$$

- *If* $G = G_1 *^s G_2$ *then*

$$\begin{aligned} & \hat{f}_G^{\mathbb{S}}(x_{s(G)}, x_{t(G)}) \\ = \; & \min_{x_{t(G_1)}, x_{s(G_2)}} \\ & \left\{ \begin{array}{l} \hat{f}_{G_1}^{\mathbb{S}}(x_{s(G)}, x_{t(G_1)}) + \hat{f}_{G_2}^{\mathbb{S}}(x_{s(G_2)}, x_{t(G)}) + \\ w_C(t(G_1), s(G_2)) \cdot w_D(x_{t(G_1)}, x_{s(G_2)}) \end{array} \right\} \end{aligned}$$

Intuitively, the cost is the sum of the communication cost between the output operator $t(G_1)$ *of* G_1 *and the input operator* $s(G_2)$ *of* G_2 *and the costs for the substreams* G_1 *and* G_2.

- *If* $G = G_1||^s G_2$ *then*

$$\begin{aligned} & \hat{f}^{\mathbb{S}}_{G}(x_{s(G)}, x_{t(G)}) \\ = \; & \min_{x_{s(G_1)}, x_{t(G_1)}, x_{s(G_2)}, x_{t(G_2)}} \\ & \left\{ \begin{array}{l} \hat{f}^{\mathbb{S}}_{G_s}(x_s, x_s) + \hat{f}^{\mathbb{S}}_{G_j}(x_j, x_j) + \\ \hat{f}^{\mathbb{S}}_{G_1}(x_{s(G_1)}, x_{t(G_1)}) + \hat{f}^{\mathbb{S}}_{G_2}(x_{s(G_2)}, x_{t(G_2)}) + \\ w_C(s, s(G_1)) \cdot w_D(x_s, x_{s(G_1)}) + \\ w_C(t(G_1), j) \cdot w_D(x_{t(G_1)}, x_j) + \\ w_C(s, s(G_2)) \cdot w_D(x_s, x_{s(G_2)}) + \\ w_C(t(G_2), j) \cdot w_D(x_{t(G_2)}, x_j) \end{array} \right\} \end{aligned}$$

Here, given two substreams $sub(G) = \{G_1, G_2\}$ *(see also Fig. 5.2 (c)), joiner* j, *and splitter* s, *the cost is the sum of the communication costs for channels* $(s, s(G_1))$, $(t(G_1), j)$, $(s, s(G_2))$, *and* $(t(G_2), j)$, *the costs for the substreams* G_1 *and* G_2, *and the costs for the splitter* s *and joiner* j.

2. $G \in \mathbb{O}$.

- *If* $G = G_u$ *with* $V = V_{se} = \{u\}$ *then*

$$\hat{f}^{\mathbb{O}}_{G}(x_{t(G)}) = w_{TP}(u, x_u).$$

- *If* $G = G_1 *^o G_2$ *then*

$$\begin{aligned} & \hat{f}^{\mathbb{O}}_{G}(x_{t(G)}) \\ = \; & \min_{x_{t(G_1)}, x_{s(G_2)}} \\ & \left\{ \begin{array}{l} \hat{f}^{\mathbb{O}}_{G_1}(x_{t(G_1)}) + \hat{f}^{\mathbb{S}}_{G_2}(x_{s(G_2)}, x_{t(G)}) + \\ w_C(t(G_1), s(G_2)) \cdot w_D(x_{t(G_1)}, x_{s(G_2)}) \end{array} \right\} \end{aligned}$$

Here the cost is the sum of the communication cost of $(t(G_1), s(G_2))$ *and the costs of substreams* G_1 *and* G_2.

- *If* $G = G_1||^o G_2$ *then*

$$\begin{aligned} & \hat{f}^{\mathbb{O}}_G(x_{t(G)}) \\ = \; & \min_{x_{t(G_1)}, x_{t(G_2)}} \\ & \left\{ \begin{array}{l} \hat{f}^{\mathbb{S}}_{G_j}(x_j, x_j) + \hat{f}^{\mathbb{O}}_{G_1}(x_{t(G_1)}) + \hat{f}^{\mathbb{O}}_{G_2}(x_{t(G_2)}) + \\ w_C(t(G_1), j) \cdot w_D(x_{t(G_1)}, x_j) + \\ w_C(t(G_2), j) \cdot w_D(x_{t(G_2)}, x_j) \end{array} \right\} \end{aligned}$$

The cost here is the sum of communication costs on the channels $(t(G_1), j)$ *and* $(t(G_2), j)$ *and the costs for substreams* G_1 *and* G_2, *and the joiner* j.

The following theorem states the correctness of the dynamic program

Theorem 2. *For any program* $G \in \mathbb{O}$,

$$\min_{x_{t(G)}} \hat{f}^{\mathbb{O}}_G(x_{t(G)}) = \min_{X_G} f_G(X_G).$$

Proof. Theorem 2 is immediate from the following lemma. □

Lemma 1. $\hat{f}^{\mathbb{S}}_G(x_{s(G)}, x_{t(G)}) = \min_{X_G \setminus \{x_{s(G)}, x_{t(G)}\}} f_G(X_G)$ *when* $G \in \mathbb{S}$ *and* $\hat{f}^{\mathbb{O}}_G(x_{t(G)}) = \min_{X_G \setminus \{x_{t(G)}\}} f_G(X_G)$ *when* $G \in \mathbb{O}$.

Proof. We prove by induction. Here we consider one base case when $G = G_u \in \mathbb{S}$ and $u \in V_{si}$, and one inductive case for $G = G_1||^o G_2$ with joiner j. Other cases can be proven similarly.

When $G = G_u \in \mathbb{S}$ and $u \in V_{si}$, then necessarily $x_{s(G)} = x_{t(G)} = x_u = b$. Now, $X_G \setminus \{x_{s(G)}, x_{t(G)}\}$ is empty, and therefore, $\hat{f}^{\mathbb{S}}_G(x_{s(G)}, x_{t(G)}) = w_{TP}(u, x_u) = f_{G_u}(x_u) = \min_{X_G \setminus \{x_{s(G)}, x_{t(G)}\}} f_G(X_G)$.

When $G = G_1||^o G_2 = (V, E, j)$, we consider that $G_1 = (V_1, E_1, t(G_1))$, $G_2 = (V_2, E_2, t(G_2))$, $G_j = (\{j\}, \emptyset, j, j)$, and $G = (\{j\} \cup V_1 \cup V_2, \{(t(G_1), j), (t(G_2), j)\} \cup E_1 \cup E_2, j)$. We assume inductively that

$$\begin{aligned} & \hat{f}^{\mathbb{S}}_{G_j}(x_j, x_j) = \min_{X_{G_j} \setminus \{x_j\}} f_{G_j}(X_{G_j}) = f_{G_j}(x_j), \\ & \hat{f}^{\mathbb{O}}_{G_1}(x_{t(G_1)}) = \min_{X_{G_1} \setminus \{x_{t(G_1)}\}} f_{G_1}(X_{G_1}), \text{ and} \\ & \hat{f}^{\mathbb{O}}_{G_2}(x_{t(G_2)}) = \min_{X_{G_2} \setminus \{x_{t(G_2)}\}} f_{G_2}(X_{G_2}) \end{aligned}$$

Substituting the rhs of the above equations for the lhs in Definition 1 we get

$$\begin{aligned}
\hat{f}_G(x_{s(G)}, x_{t(G)}) = & \min_{X_{G_1} \cup X_{G_2}} f_{G_j}(x_j) + f_{G_1}(X_{G_1}) + f_{G_2}(X_{G_2}) + \\
& w_C(t(G_1), j) \cdot w_D(x_{t(G_1)}, x_j) + \\
& w_C(t(G_2), j) \cdot w_D(x_{t(G_2)}, x_j). \\
= & \min_{X_G \setminus \{x_j\}} \sum_{u \in V} w_{TP}(u, x_u) + \\
\sum_{(u,v) \in \{(t(G_1),j),(t(G_2),j)\} \cup E_1 \cup E_2} & w_C(u,v) \cdot w_D(x_u, x_v) \\
= & \min_{X_G \setminus \{x_{t(G)}\}} f_G(X_G)
\end{aligned}$$

□

4.3.2 Memo Table

For every call to $\hat{f}_G^{\mathbb{S}}$ and $\hat{f}_G^{\mathbb{O}}$, the algorithm allocates a table containing the possible return values of the call. Each table is a matrix $T(G)$, where for $G \in \mathbb{S}$, $T(G)_{x,y} = \hat{f}_G^{\mathbb{S}}(x,y)$, and when $G \in \mathbb{O}$, $T(G)$ is a row matrix with $T(G)_x = \hat{f}_G^{\mathbb{O}}(x)$. For a call to $\hat{f}_G^{\mathbb{S}}$, the rows and columns of the table correspond to the possible placements of $s(G)$ and $t(G)$. For a call to $\hat{f}_G^{\mathbb{O}}$, the table only has a single row, and the columns similarly correspond to the placements of $t(G)$.

We illustrate the tabling mechanism using the stream in Fig. 4.1 (b). The algorithm starts with the tables for the operators, as shown by the first three tables of Fig. 4.4, which reflect the computation cost of an operator on a specific sensor node. The rows and columns of the tables are labelled with the sensor nodes and the base station. The entries of the tables are computed from the placement and transition costs of operators displayed in Fig. 4.2 (a) and (c). For example, $T(G_{j_1})_{n_2,n_2} = \hat{f}_{G_{j_1}}^{\mathbb{S}}(n_2,n_2) = 16$ reflects the fact that when the operator j_1 is placed on sensor node n_2, the sum of the placement and transition costs for the operator is 16, as the placement cost for j_1 on n_2 is 6 and the transition cost from n_1 to n_2 is 10. Fig. 4.4 (d) shows a table for one sense operator o_1, which has fixed placement on node n_1. Similarly, the tables are generated for other sense operators based on Fig. 4.2 (a).

After generating the tables for each operator, the algorithm builds the tables for the larger composites. Let us now consider how the elements of $T((G_{o_1}||^o G_{o_2}) *^o G_{f_1})$ (Fig. 4.4 (f)) is computed – in particular, when $x_{t(G)} = n_2$. To obtain this element, according to Definition 1 for $\hat{f}_G^{\mathbb{O}}$ with $G_1 *^o G_2$, we enumerate possible placements for $t(G_{o_1}||^o G_{o_2}) = j_1$ and for $s(G_{f_1}) = f_1$ such that $\hat{f}_G^{\mathbb{O}}(x_{t(G)})$ is minimal, for $x_{t(G)} = n_2$.

$x_{s(G)} \backslash x_{t(G)}$	n_1	n_2	n_3	b
n_1	10	∞	∞	∞
n_2	∞	16	∞	∞
n_3	∞	∞	26	∞
b	∞	∞	∞	41

(a) $T(G_{j_1})$

$x_{s(G)} \backslash x_{t(G)}$	n_1	n_2	n_3	b
n_1	9	∞	∞	∞
n_2	∞	20	∞	∞
n_3	∞	∞	16	∞
b	∞	∞	∞	23

(b) $T(G_{j_2})$

$x_{s(G)} \backslash x_{t(G)}$	n_1	n_2	n_3	b
n_1	7	∞	∞	∞
n_2	∞	15	∞	∞
n_3	∞	∞	28	∞
b	∞	∞	∞	36

(c) $T(G_{f_1})$

$x_{t(G)}$	n_1	n_2	n_3	b
	0	∞	∞	∞

(d) $T(G_{o_1})$

$x_{t(G)}$	n_1	n_2	n_3	b
	23	21	44	59

(e) $T(G_{o_1}||^o G_{o_2})$

$x_{t(G)}$	n_1	n_2	n_3	b
	49	36	70	78

(f) $T((G_{o_1}||^o G_{o_2}) *^o G_{f_1})$

$x_{t(G)}$	n_1	n_2	n_3	b
	62	66	59	76

(g) $T(((G_{o_1}||^o G_{o_2}) *^o G_{f_1})||^o G_{o_3})$

$x_{t(G)}$	n_1	n_2	n_3	b
	∞	∞	∞	70

(h) $T((((G_{o_1}||^o G_{o_2}) *^o G_{f_1})||^o G_{o_3}) *^o G_b)$

Figure 4.4: Memo tables for operators and composites (see Fig. 4.1(b))

The element is therefore the minimal value among the following.

$$T(G_{o_1}||^o G_{o_2})_{n_1} + T(G_{f_1})_{n_2,n_2} + w_C(j_1, f_1), \quad \text{where } x_{j_1} = n_1 \text{ and } x_{f_1} = n_2 \qquad (1)$$
$$T(G_{o_1}||^o G_{o_2})_{n_2} + T(G_{f_1})_{n_2,n_2}, \quad \text{where } x_{j_1} = n_2 \text{ and } x_{f_1} = n_2 \qquad (2)$$
$$T(G_{o_1}||^o G_{o_2})_{n_3} + T(G_{f_1})_{n_2,n_2} + w_C(j_1, f_1), \quad \text{where } x_{j_1} = n_3 \text{ and } x_{f_1} = n_2 \qquad (3)$$
$$T(G_{o_1}||^o G_{o_2})_{b} + T(G_{f_1})_{n_2,n_2} + w_C(j_1, f_1), \quad \text{where } x_{j_1} = b \text{ and } x_{f_1} = n_2 \qquad (4)$$

Here $t(G) = t(G_{f_1})$ such that $\hat{f}^{\mathbb{S}}_{G_{f_1}}(x_{s(G_{f_1})}, x_{t(G)}) < \infty$ only when $x_{s(G_{f_1})} = x_{t(G_{f_1})} = x_{t(G)} = n_2$ (see Fig. 4.4 (c)), and therefore in the above we do not consider the values computed from $T(G_{f_1})_{z,n_2}$ with $z \neq n_2$ as they cannot be minimal. From Fig. 4.4 (b) we know that the value of $w_C(j_1, f_1)$, is 21 and is added when $x_{j_1} \neq x_{f_1}$, since we assume that $w_D(x_{j_1}, x_{f_1}) = 1$ when $x_{j_1} \neq x_{f_1}$ and 0 otherwise. The minimal value of 36 is given

by (2).

The tables also contain the information on optimal assignments. After constructing all tables, we reconstruct the solution recursively in the following way. Given a composite G in the graph, with an element of interest in the table $T(G)$ (this value is the optimal value when G is the top-level program), we can compute the placements for the input/output operators of the immediate children from the definition of the dynamic program. For example, when $G = G_1 *^o G_2$, the following holds from the dynamic program (Definition 1):

$$\begin{aligned} & T(G)_{x_{t(G)}} \\ = \; & \min_{x_{t(G_1)}, x_{s(G_2)}} \left\{ \begin{array}{l} T(G_1)_{x_{t(G_1)}} + T(G_2)_{x_{s(G_2)}, x_{t(G)}} + \\ w_C(t(G_1), s(G_2)) \cdot w_D(x_{t(G_1)}, x_{s(G_2)}) \end{array} \right\} \end{aligned}$$

Therefore, we compute the placement $x_{t(G_1)}$, $x_{s(G_2)}$ by enumerating their possible value pairs (for which there are $|N|^2$) and for each value pair, see if the rhs sub-expression $T(G_1)_{x_{t(G_1)}} + T(G_2)_{x_{s(G_2)}, x_{t(G)}} + w_C(t(G_1), s(G_2)) \cdot w_D(x_{t(G_1)}, x_{s(G_2)})$ equals $T(G)_{x_{t(G)}}$. We further explain this method via example. In Fig. 4.2, the optimal cost of 70 for the whole program is given in the table $T((((G_{o_1} ||^o G_{o_2}) *^o G_{f_1}) ||^o G_{o_3}) *^o G_b)$ – here, a shaded element. Optimal cost is computed using the shaded elements for $T(((G_{o_1} ||^o G_{o_2}) *^o G_{f_1}) *^o G_{o_3})$ and $T(G_b)$ (not shown), and this value is further computed from the shaded cell $T((G_{o_1} ||^o G_{o_2}) *^o G_{f_1})$, and so on. After marking the values, the optimal placement can thus be inspected from the tables, which is $o_1 \mapsto n_1$, $o_2 \mapsto n_2$, $o_3 \mapsto n_3$, $j_1 \mapsto n_2$, $f_1 \mapsto n_2$, $j_2 \mapsto n_3$, and $c \mapsto b$. The placement is partially shown by the shaded elements of Fig. 4.4, and is indicated in Fig. 4.1 (b) with nodes enclosed in rectangle, hexagon, circle, and pentagon respectively indicate the placement of the operators to node n_1, n_2, n_3, and b.

4.3.3 Complexity Bounds

We now discuss the complexity bounds of the algorithm based on the dynamic program. In the following discussions, $m = |V|$ is the number of operators of the composite under consideration, and $n = |N|$ is the number of sensor nodes. There are two important procedures in the algorithm:

1. construction of tables for each composite, and

2. reconstruction of optimal solution from the tables.

The following theorem holds:

Theorem 3. *Given a compositional stream graph G and with operators V to be placed on nodes N, there is an algorithm to compute optimal placement with time complexity in $O(mn^4)$.*

Proof. We first discuss several lemmas that specify the complexity bounds of each of the above procedures. Here $|V| = m$ and $|N| = n$.

To reason about the complexity bounds of steps 1 and 2, we first state a bound on the number of tables constructed by the algorithm.

Lemma 2. *The number of tables constructed is in $O(m)$.*

Proof. Here we first prove that for every composite $G = (V, E, g, h)$ or $G = (V, E, h)$, the number of generated tables is $|V| + b$, with $0 \leq b < |V|$.

In the base case where $|V| = 1$, the number of tables is 1, which clearly satisfies the invariant for $b = 0$.

Next we show the proofs for when $G = G_1 *^o G_2$ and $G = G_1||^s G_2$ only, since other cases can be proven similarly.

In case $G = G_1 *^o G_2$, with $G_1 = (V_1, E_1, h_1)$ and $G_2 = (V_2, E_2, g_2, h_2)$, we assume the number of tables for G_1 and G_2 are respectively $|V_1| + b_1$ and $|V_2| + b_2$, where $0 \leq b_1 < |V_1|$ and $0 \leq b_2 < |V_2|$. We add another table for the composite such that the total number of table is $|V_1| + |V_2| + b_1 + b_2 + 1 = |V| + b_1 + b_2 + 1$ since $|V| = |V_1| + |V_2|$. Here, $0 \leq b_1 + b_2 + 1 < |V|$.

In the case of $G = G_1||^s G_2$, with $G_1 = (V_1, E_1, g_1, h_1)$ and $G_2 = (V_2, E_2, g_2, h_2)$, we assume the number of tables for G_1 and G_2 are respectively $|V_1| + b_1$ and $|V_2| + b_2$, where $0 \leq b_1 < |V_1|$ and $0 \leq b_2 < |V_2|$. Here we introduce two more actors – the splitter and the joiner such that $|V| = |V_1| + |V_2| + 2$. – and add three more tables, one table each for the splitter and the joiner, and another table for the whole composite. Therefore the number of tables is $|V_1| + |V_2| + 2 + b_1 + b_2 + 1 = |V| + b_1 + b_2 + 1$. Here, $0 \leq b_1 + b_2 + 1 < |V|$.

Since for any composite $G = (V, E, g, h)$ or $G = (V, E, h)$ the number of tables is $|V| + b$ for some $0 \leq b < |V|$, the number of tables is bounded by $2|V|$, hence in $O(m)$. □

The construction of each table itself is in $O(n^4)$, and therefore the following lemma holds:

Lemma 3. *In placing a program G with operators V on nodes N, the minimal cost can be computed in* $O(mn^4)$.

Proof. We first demonstrate that at each table construction step, the computation time required is in $O(n^4)$, with $n = |N|$. Here we consider two cases of composites.

- If G is an operator, the table construction involves considering just the table size, which is n^2.

- In case $G = G_1 *^s G_2$, to compute the value of a table entry $T(G)_{x_{s(G)},x_{t(G)}}$, we need to consider all possible combinations for $x_{t(G_1)}$ and $x_{s(G_2)}$. Their number is in $O(n^2)$, and since there are n^2 entries in the table to be constructed, the complexity will be in $O(n^4)$.

Although the proof for all cases is not shown here; the case when $G = G_1 *^s G_2$ has the worst complexity bound, and therefore the table construction has a complexity bound in $O(n^4)$. Since from Lemma 2 the number of tables constructed is in $O(m)$ with m the number of operators of a program, the complexity of table construction is in $O(mn^4)$.

□

Lastly, the reconstruction of the optimal solution from the tables (Section 4.3.2) is in $O(mn^2)$.

Lemma 4. *Given a program G with operators V to be placed on N, given minimal placement cost, the placement can be computed in* $O(mn^2)$.

Proof. The first challenge is to determine the complexity bound for recovering a solution from the substream tables, given the optimal value in the table of a composite G. For this, we assume that G is an arbitrary composite, but we only consider two of them as others can be proven similarly. First, in case $G = G_u = (\{u\}, \emptyset, u, u)$, there are already placements $x_{s(G)}$ and $x_{t(G)}$, and therefore the recovery terminates in $O(1)$. In case $G = G_1 *^o G_2$, we need to choose the values for $x_{t(G_1)}$ and $x_{s(G_2)}$ such that $T(G)_{x_{t(G)}} = \min_{x_{t(G_1)},x_{s(G_2)}} T(G_1)_{x_{t(G_1)}} + T(G_2)_{x_{s(G_2)},x_{t(G)}} + w_C(t(G_1), s(G_2)) \cdot w_D(x_{t(G_1)}, x_{s(G_2)})$ (see Definition 1). This process examines the possible placement pairs for $t(G_1)$ and $s(G_2)$, for which there are n^2, and the execution time is therefore in $O(n^2)$, which is also the

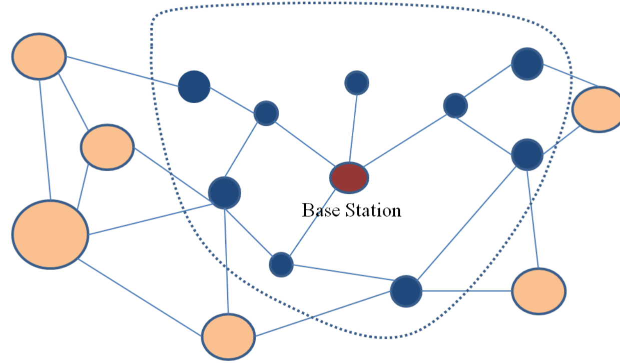


Figure 4.5: Sub-network two hops away from base station

worst possible running time bound for any composite. Since from Lemma 2 the number of tables is in $O(m)$, the complexity bound for recovering the solution from the memo tables is in $O(mn^2)$.

□

From Lemmas 3 and 4, the most complex of the three procedures mentioned above is table construction, hence Theorem 3. □

The most expensive of these three steps is step 2, where the computation cost is due to: a) generating $O(m)$ tables (one for each composite), and b) computing the elements of each table. For each table, there are at most n^2 elements to be computed, and the computation requires examining the tables of the substreams, a process in $O(n^2)$. Therefore, the whole process is executed in $O(m \cdot n^2 \cdot n^2) = O(mn^4)$.

4.3.4 Mitigating Complexity by Locality

The time complexity of the dynamic program increases quartic in the number of sensors, which makes the running time of the dynamic program infeasible for large networks. To overcome or at least alleviate this problem, we restrict the search space of the dynamic program. A sub-network of the sensor network is created that contains sensor nodes executing sense operators, the base station, and proximate nodes. The size of the sub-network is smaller and hence improves the running time of the dynamic program; however, the placement of operators becomes restricted and will affect the energy-efficiency.

For creating the sub-network, we use a heuristic that chooses nodes for the sub-network that are in the proximity of the base station. Fig. 4.5 illustrates an example

network with 16 nodes, including a base station. In the figure, the size of the node indicates its distance from the base station; i.e., bigger nodes are further away from the base station. Our simple heuristic chooses a sub-network whose nodes are at most two hops away from the base station, indicated in Fig. 4.5 by nine nodes enclosed within a dotted shape. We increase or decrease the size of the sub-network by setting the distance parameter, i.e., number of hops from the base station.

4.4 Experimental Evaluation

We evaluate our techniques for the migrating operator placement problem (MOPP) as follows:

- Is dynamic programming more efficient than integer linear programming (ILP) for MOPP ?
- What is the energy efficiency of in-network processing in comparison with data forwarding?
- How does a restricted local search space affect speed and quality of the placement?

For our experimental results we used a simulation approach to answer the experimental questions above. We implemented the simulator in Python, which generates sensor networks and stream queries, and runs the simulation in a fixed-time step mode. In each time step the following events may be triggered: adding a query, migrating a query, deleting query, adding a sensor node, and removing a sensor node. For activities including communication, execution, and migration of operators, energy is consumed at sensor node level. In each time step the simulator adjusts the energy levels of a sensor node depending on the activities in the time step and the running overheads of the sensor.

For our experiments we used instances of wireless sensor networks that are generated from the Erdös-Rényi graph model $G(n, p)$ [100], where n indicates the number of sensor or mobile nodes, and p is the probability with which each edge will be included in the graph. This model is simple and widely used for mathematical analysis that generates network graphs with varying degree of connectivity [101, 102]. In our experiments we generated the network graphs with $p = 0.5$.

Operators	Nodes	ILP (s)	DP (s)
4	1	0.1	0.029
8	2	0.1	0.054
11	3	3.3	0.112
14	4	176.2	0.371
17	5	$*$	1.165
20	6	$*$	3.173
23	7	$*$	7.659
26	8	$*$	16.291
29	9	$*$	32.088
32	10	$*$	60.234

Table 4.1: GLPK and dynamic program running times

User queries are introduced and deleted throughout the simulation. A query is generated by a probabilistic scheme: We assign a composite (cf. Fig. 5.2) a probability (i.e. 25% for sense and filter, and 10% for other composites). The composition tree of a query is recursively constructed. The composite of a node in the composition tree is chosen randomly according to its probability, and the construction ends as soon as all leaves in the composition tree are sense operators (cf. Fig. 5.2(d)).

We ran our simulator and the algorithm for MOPP on a Core2 Duo 2.99 GHz Intel machine with 4 GB RAM. The simulator and placement algorithm were implemented in Python.

4.4.1 ILP vs. Dynamic Programming

We formulated the MOPP as an AMPL program [88] and compared the running time of the dynamic program (cf. Section 4.3) with the running time of the ILP solver GLPK [87], that solves our AMPL program. In Table 4.1, **Operators** represents the total number of operators in a given set of queries, **Nodes** represent the number of sensor nodes, **ILP** represents the running time for GLPK runs in seconds (where $*$ represents a run that takes more than two hours), and **DP** represents the running times for the dynamic program in seconds. The time taken to execute the dynamic program is substantially less than that for GLPK. As the table shows, the difference becomes more pronounced with larger problem sizes.

Operators	Nodes	cost(DP)	cost(Fwd)	DP (s)
2	2	39	39	0
6	2	95	121	0
8	3	208	296	0.015
12	5	428	225	0.031
17	3	395	584	0.016
18	8	215	517	3.5
20	6	295	579	1.64
26	13	470	844	72.782
38	10	669	1397	53.218
56	10	849	1782	60.891
57	12	815	1790	86.672

Table 4.2: Optimal in-network vs. data forwarding

4.4.2 In-network vs. Data Forwarding

Our algorithm (cf. Section 4.3) places operators on cost-efficient sensor nodes and hence resembles an in-network processing scheme, where computations of operators are performed inside the network to reduce communication overheads. An alternative scheme is *data forwarding*, which places all operators except the sense operations on the base station assuming that the base station's energy resources are unconstrained.

Table 4.2 provides a comparison of energy costs between the in-networking processing and the data forwarding schemes. The column **Operators** gives the number of operators in a query, and column **Nodes** denotes the number of sensor nodes. The energy costs for the in-network processing scheme are shown in column **cost(DP)** and column **cost(Fwd)** shows the data forwarding costs. With smaller network and query sizes, the cost incurred with dynamic programming is similar to data forwarding. The data forwarding scheme becomes energy-inefficient with bigger queries and sensor networks. In our experiment, the in-network processing scheme based on dynamic programming is up to 54% more energy-efficient and is well suited for energy-constrained query processing.

4.4.3 Locality of Placement

As shown in Table 4.2, the running time of the dynamic program increases rapidly with the number of sensor nodes. For large sensor networks, the algorithm's running time

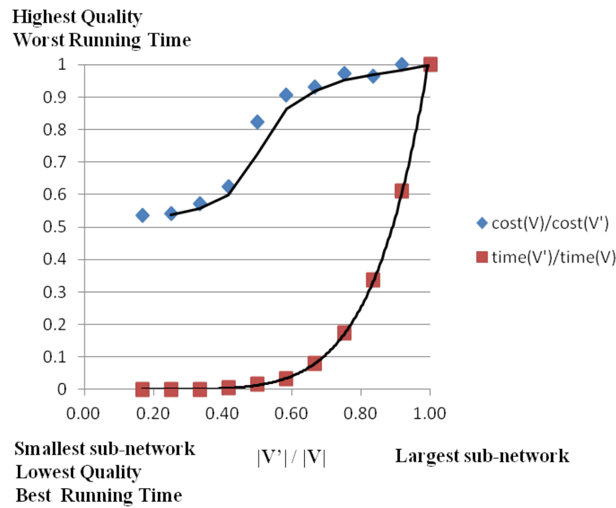


Figure 4.6: Locality heuristic vs. optimal in-Network

becomes prohibitive. To mitigate this issue, we propose a locality strategy (cf. Section 4.3.4) to select a sub-network for the dynamic program. The sub-network limits the possible placement and hence will have an impact on the placement quality – that is, it will be less than optimal. To evaluate this approach, we compute the running time and in-network processing cost for the various sizes of the sub-network and compare them with the optimal value (the result when considering all nodes in the network).

Fig. 4.6 shows the effect of locality. We represent the size of the actual network as $|V|$ and the size of a sub-network as $|V'|$. The cost incurred with $|V|$ is shown as cost(V) and the cost incurred with the sub-network of size $|V'|$ is shown as cost(V'). The in-network cost moves near to optimal as the size of the sub-network increases towards the actual size of the network. We notice that the worst placement quality appears to stabilise at 50% with the smallest sub-network sizes. The details provided in Fig. 4.6 allow us to choose the size of the sub-network; using this size, we can compute the nearly optimal placement of the query operators in a relatively short time. For our experimental data, a sub-network with the size of 60% of the actual network computes the placement in less than 5% running time, with a placement quality of 91% of the optimal.

Fig. 4.7 depicts the comparison between the cost of our in-network processing with locality and the cost incurred due to data forwarding. The worst results using locality are still significantly better than data forwarding. This result supports our argument that optimised in-network processing is superior to data forwarding, even with the locality

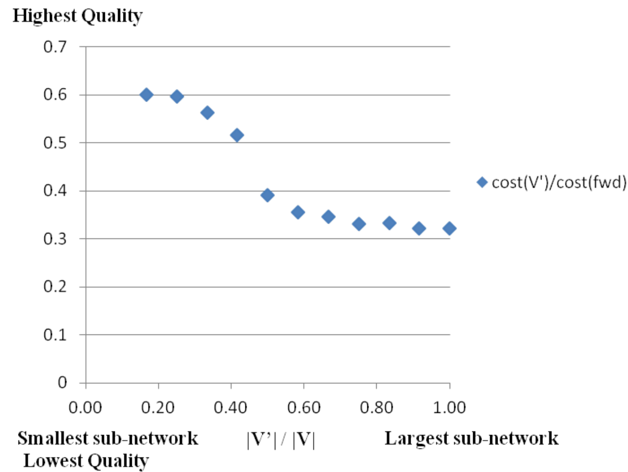


Figure 4.7: Locality heuristic vs. data forwarding

heuristic.

4.5 Chapter Summary

We have presented the migrating operator placement problem (MOPP) on wireless sensor networks and mobile clouds. The model is applicable to stream graphs, where the system consists of operators and communication channels between them. The objective was to minimise the energy costs of computation, communication and transition due to the changes in the stream graph and the migration of operators. We devised an algorithm based on dynamic programming that computes optimal operator placement on the network, with running time polynomially bounded to the input graph size for an important subclass of the stream graphs. To improve our algorithm, we developed a heuristic that considers only nodes with close proximity to the base station. We presented an experimental evaluation that confirms the efficiency of our approach, by comparing it with integer linear programming. Using a simulator, we demonstrated the effectiveness of the optimal placement by comparing the computed energy usage with that of data forwarding. The results show that our approach is almost 54% more energy-efficient. Our heuristic significantly improves the running time of the dynamic programming algorithm and computes nearly optimal placement in less than 5% of the running time.

Chapter 5

Timeliness in Curracurrong

A wide range of sensors are deployed at distant locations to collect a stream of data like temperature rise near disaster-prone areas and intrusion at highly secured locations. The data collected at sensors can be numerical, digital, or discrete; to extract complex information, the sensed data need to be filtered, transformed, and merged. Some of the existing systems – including Aurora [27], Medusa [26], Borealis [28], Mad-WiSe [30], and Curracurrong [70] – express WSN queries with stream data processing. Existing systems emphasise the provision of energy efficiency and flexibility in WSN applications. Apart from those features, it is important that sensed data should reach the base station reliably and on time for time-critical applications such as health and disaster area monitoring.

The ad-hoc infrastructure and resource constraints in WSN increase the uncertainty of successful and real-time data transmission. Approaches to overcoming such challenges in WSN have been studied for a decade [103]. Recent studies have analysed end-to-end timeliness in terms of probability distribution [104, 105] and first-order statistics [106]. For unreliable networks, work on real-time queuing theory provides stochastic models [107]. This chapter sets out a comprehensive approach for determining delays in stream query processing using event causality concepts. Based on event causality, we introduce the notion of timeliness into the semantics for stream processing, and an algorithm to measure end-to-end delays in processing.

Consider a disaster area monitoring application in the Curracurrong system [70], which deploys several proximate sensors and measures temperature change. The query collects the average temperature reading from the sensors placed at distant locations and checks whether the reading goes beyond a certain threshold. Figure 5.1 (a) shows

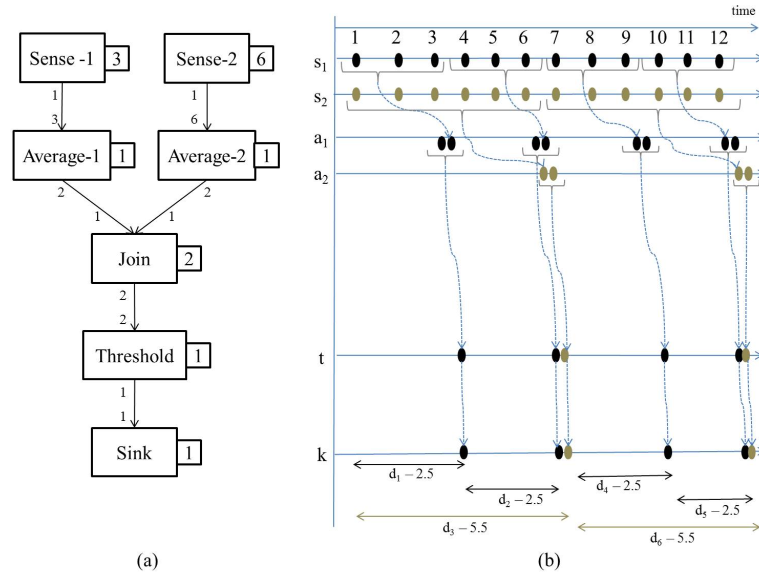


Figure 5.1: Stream graph and event causality with synchronised sensors

a stream graph for Curracurrong query of the given application, where the result is recorded at the base station. The stream graph is composed of: two sense operations `Sense-1` and `Sense-2`; two filters `Average-1` and `Average-2` to compute average temperature from `Sense-1` and `Sense-2`, respectively; a join operation `Join` to merge data from two filters; a filter `Threshold` that checks whether the average temperature readings are above threshold; and a `Sink` operator, where the final data is recorded. All operators are connected with uni-directional communication channels.

In the example, sensor nodes are deployed at distant locations and operators are placed in the intermediate sensor nodes before the data reach the sink (at the base station). It is important that temperature-related data collected at sensors reach the sink on time so that the user can take required action. Determining the freshness of the data relies on knowing how long it takes to propagate data from a sensor to the sink. Various operations on the sensed data insert certain delays in information generation at the sink (Figure 5.1 (b)). The figure shows that both sensors, represented as s_1 and s_2, generate data with uniform frequency and propagate the sensed data to consecutive operators until they reach the sink operator. During data propagation, the intermediate operators, such as average and threshold, consume more than one data token and take time for computation, both of which ultimately result in delay. Sense operators continuously generate data at every second and propagate them to average operators.

Filter `Average-1`, represented as a_1, has window size 3, and therefore waits 3 s for data availability and computes the average, adding a few milliseconds delay. Likewise, filter `Average-2`, represented as a_2, waits for 6 s and inserts a few milliseconds delay during computation. The `Join` operator merges the data from two average filters and forwards them to `Threshold` (t) without any delay. The `Threshold` operator inserts a few milliseconds delay before the data token finally reaches the sink, k. The first data token generated at the sense reaches the sink with 2.5 s delay. The delays are not the same for each token reaching the sink, for two reasons: the different data rates of each operator; and sensor periodicity. The challenge is thus how to compute precise end-to-end delays in a stream graph, as shown in Fig. 5.1 (b).

We used compositional stream graphs to establish a causality relationship for tokens and a notion of a time steady state to compute end-to-end delays of stream queries. Our approach was built on top of Curracurrong framework [70]. In summary, the chapter makes three contributions:

1. a denotational semantics (cf. Section 5.2) for stream data processing that explains time information propagation,

2. an algorithm (cf. Section 5.3) to measure the end-to-end delays in a stream graph, and

3. an experimental evaluation (cf. Section 5.4) to show the efficiency and effectiveness of our approach in determining end-to-end delays.

The chapter is organised as follows. Section 5.1 defines our model and formally describes the problem definition statement. Section 5.2 defines compositional stream graphs and semantics. Section 5.3 introduces an algorithm and defines the abstraction of concrete semantics to determine delays. In Section 5.4 we present experimental evaluations to confirm the efficacy of our approach, together with periodicity scaling.

5.1 System Model and Problem Definition

As in the data flow model [108], we represent a WSN query as a *stream graph* $G = (V, E)$, whose vertices V are called *operators* and whose edges $E \subseteq V \times V$ are called *channels*. The *source* of an edge (u, v), denoted by $src(u, v)$, is u and the destination of an edge (u, v), denoted by $dest(u, v)$, is v. A channel $(u, v) \in E$ queues data elements

called *tokens*, which are passed from the output of operator u to the input of operator v. The stream graph is augmented by two functions $c : E \rightarrow \mathbb{N}$ and $p : E \rightarrow \mathbb{N}$, that associate with each channel $e \in E$ the number of consumed tokens $c(u,v)$ and produced tokens $p(u,v)$ for edge (u,v). We refer to graph $G(V,E,c,p)$ as a *synchronous stream graph*. In our model, stream query generates data tokens (along a physical timeline), which provide a useful abstraction for event-based distributed systems. We define an event as a generation of a data token, as follows.

Definition 2. *An* event *is the production of token v at time $t \in \mathbb{R}^+$, represented as a pair (v,t). We use variables x, y, z and their indexed versions to denote events, and Ω to denote the set of all events. We define a function τ such that when $x = (v,t)$, $\tau(x) = t$.*

In event-based systems, *causality* is a key concept, which was introduced as a unique formal modelling of communicating actors in [109]. The model concerns causality as the propagation of events across the system. An actor has a semantics called its *causality interface*, which relates input with output events (pairs of time and data values). The system semantics are algebraically defined in terms of the semantics of each component, connected by the edges of the graph. Similarly, we formally define causality as follows.

Definition 3. *The causality relation $\delta(x,y)$ indicates that event x causes event y. Necessarily for any events x and y, $\delta(x,y) \Rightarrow \tau(x) \leq \tau(y)$. The causality relation is transitive; i.e., if $\delta(x,y)$ and $\delta(y,z)$ holds, then $\delta(x,z)$ holds. The causality relation is antisymmetric such that for any $x,y \in \Omega$, if $\delta(x,y)$ and $\delta(y,x)$ both holds, then $\tau(x) = \tau(y)$ and therefore $x = y$. Each event $x \in \Omega$ has an associated set $\Psi(x)$ of events that caused it, obtained through the function $\Psi : \Omega \rightarrow \mathcal{P}(\Omega)$, which is defined as $\Psi(x) = \{y \in \Omega | \delta(y,x)\}$.*

Figure 5.1 (b) shows the causality relation between events over time span t for the stream graph in Figure 5.1 (a). Figure 5.1 (b) represents the events generated at `Sense-1` as $x_1^{s_1}, x_2^{s_1}$, and $x_3^{s_1}$ when three data tokens are produced. `Average-1` operator consumes the three data tokens and generates two tokens represented as events $x_1^{a_1}$ and $x_2^{a_1}$. These are 'caused' by previous events $x_1^{s_1}, x_2^{s_1}$, and $x_3^{s_1}$. Hence, there are causality relationships between two sets of events, such that $\{\delta(x_1^{s_1}, x_1^{a_1}), \delta(x_1^{s_1}, x_2^{a_1}), \delta(x_2^{s_1}, x_1^{a_1}), \delta(x_2^{s_1}, x_2^{a_1}), \delta(x_3^{s_1}, x_1^{a_1}), \delta(x_3^{s_1}, x_2^{a_1})\} \subset \Omega$. In the same way, at the sink the final event generated is x_1^k, which represents the data token at the output channel of the sink operator. For example, in Figure 5.1 (b), $\Psi(x_1^k) = \{x_1^{s_1}, x_2^{s_1}, x_3^{s_1}, x_1^{a_1}, x_2^{a_1}, x_1^t, x_1^k\}$.

5.1.1 Problem Definition

From the definition of event causality, an event at the sink is causally dependent on the event(s) generated at the source of the stream graph. We measured the delays in the stream graph – the time taken by a data token associated with an event x to reach the sink:

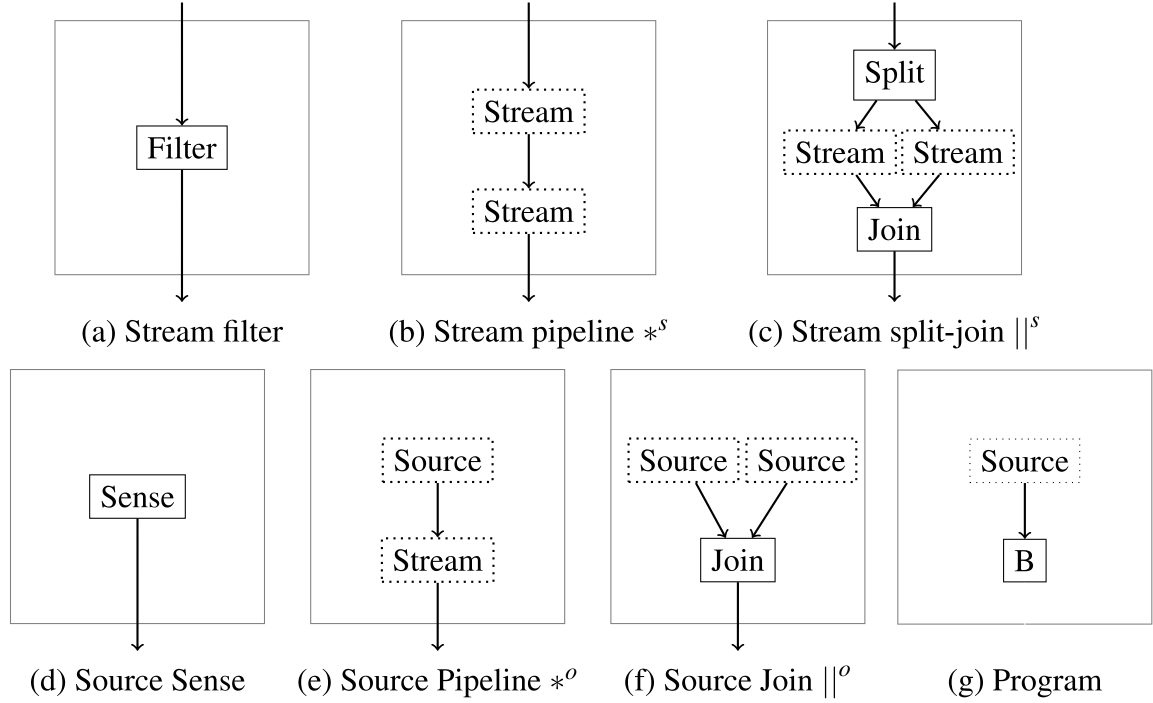
$$\Delta(x) = \tau(x) - \min_{y \in \Psi(x)} \tau(y). \tag{5.1}$$

$\Delta(x)$ represents the maximum delay until event x occurs from the occurrence time of another event that causes it. As mentioned in the system model, an event (at the sink) may depend causally on the number of source events, based on the nature of an application. Where this is this case, delays are measured between an event x at the sink and the first source event that causally generates the event x. For the same reason, in the second term of Equation 5.1, we represent the minimum occurrence time among all events that are the prerequisites of event x. For example, in Figure 5.1 (b), the first event generated at the sink is x_1^k and the list of events that causally generates the event at sink is $\Psi(x_1^k) = \{x_1^{s_1}, x_2^{s_1}, x_3^{s_1}, x_1^{a_1}, x_2^{a_1}, x_1^t, x_1^k\}$. In a specific instance of an event timeline (Figure 5.1 (b)), $\tau(x_1^{s_1}) \leq \tau(x_2^{s_1}) \leq \tau(x_3^{s_1}) \leq \tau(x_1^{a_1}) \leq \tau(x_2^{a_1}) \leq \tau(x_1^t) \leq \tau(x_1^k)$; therefore the delay to be measured is $\Delta(x_1^k) = \tau(x_1^k) - \tau(x_1^{s_1})$.

To compute delays, we consider the time taken for computation at each operator during query processing. For the simplicity of our model, we assume uniform time for communication between computing nodes. Any delay caused by communication failure and data transmission is modelled as an additional delay operator, referred to as *delayop* and added between the original stream graph operators. For example, communication interruption between operators u and v is represented as an operator *delayop*, such that $(u, delayop) \in E$ and $(delayop, v) \in E$.

5.2 Semantics for Stream Graph Composites

Figure 5.3 represents the ways to build a compositional stream graph, using the composites in Figure 5.2 (explained in Chapter 4), for the example `Average` query. We start with the actual query as program composite, shown as the outermost dotted rectangle. The source composite in the program is expanded as a source pipeline $*^o$, consisting of source and stream composites. The stream composite in the pipeline is transformed



(a) Stream filter (b) Stream pipeline $*^s$ (c) Stream split-join $||^s$

(d) Source Sense (e) Source Pipeline $*^o$ (f) Source Join $||^o$ (g) Program

Figure 5.2: Composites

to a filter composite for `Threshold` operator, whereas the source composite is further expanded as source join composite $||^o$, with two source composites expanded as source pipelines $*^o$. Two source pipelines are expanded `Sense` as source composites and `Average` as filter composites.

5.2.1 Semantics

Figure 5.4 defines the semantics for the stream and source composites, where we assume that each stream s belongs to the domain *Stream*.

Definition 4. *A Stream is inductively defined as follows:*

1. $[\,]$, *called the* empty *stream, is in Stream*.
2. *If* $s \in Stream$, *then* $[x : s] \in Stream$, *with* $x \in \Omega$.

In any WSN query, a sense operator works as a source and atomically generates a stream of tokens, and is defined as source composite $O[\![G]\!]$ with $G = (u, \emptyset, u, u)$. The sensed raw data is further filtered and processed by filter composites. A filter f consumes $|c_f|$ tokens from its input channel and generates output stream with $|p_f|$ tokens.

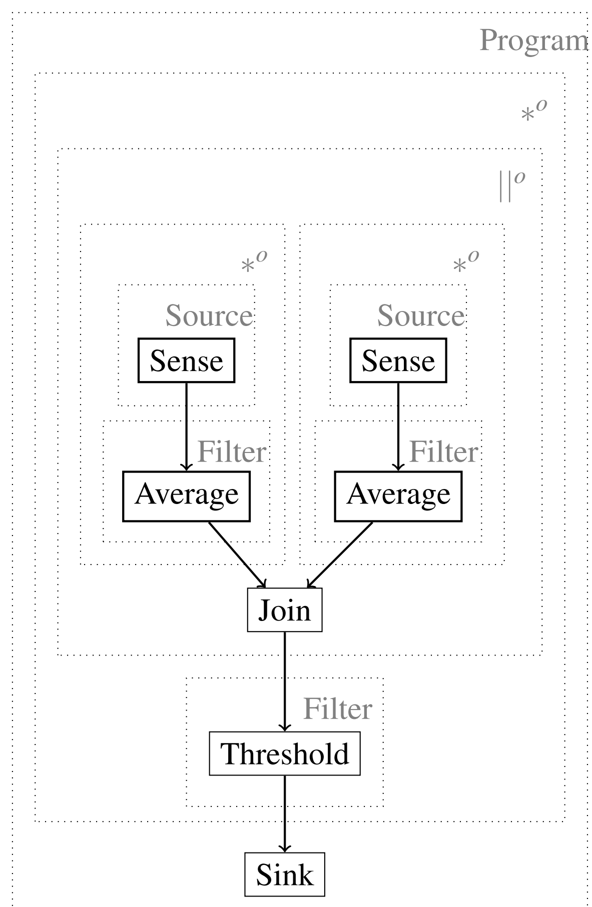


Figure 5.3: Example of a compositional stream graph using stream and source composites

Since each channel has finite memory, stream graph requires to have all the channels in the same state as they were in original state; to achieve this, filters are executed repeatedly for certain number of times – referred as repetition vector r_f. The detailed explanation of repetition vector $\vec{r}$ is given in Section 5.3.1.1. The semantics for filter is presented as a stream composite rule $\mathcal{S}[\![G]\!]$ with $G = (f, \emptyset, f, f)$. Some data processing requires a number of consecutive composites connected as a pipeline. A pipeline composite forwards an output stream of tokens to the input of another composite; this is explained with stream and source pipeline semantics $\mathcal{S}[\![G_1 *^s G_2]\!]$ and $O[\![G_1 *^o G_2]\!]$, respectively. Some WSN queries may collect data from different sensors and combine those results using join composite while some other queries may process sensed data in parallel using split-join composite; the semantics for both composites are defined as $O[\![G_1 ||^o G_2]\!]$ and $\mathcal{S}[\![G_1 ||^s G_2]\!]$, respectively.

In our model, both join and split-join composites propagate the incoming data tokens based on their times of occurrences and, hence, produce an interleaved merge of two incoming data streams. We denote the join interleaving as a function $\oplus$.

Definition 5. *The function $\oplus$: Stream $\times$ Stream $\rightarrow$ Stream is defined inductively as follows.*

1. $[\,] \oplus s = s$,
2. $s \oplus [\,] = s$,
3. $[x_1 : s_1] \oplus [x_2 : s_2] = \begin{cases} [x_1 : s_1 \oplus [x_2 : s_2]] & \text{if } \tau(x_1) \geq \tau(x_2) \\ [x_2 : [x_1 : s_1] \oplus s_2] & \text{otherwise} \end{cases}$

It is important to note here that the sense operators in our system model generate atomic data tokens on a physical timeline; therefore a stream of events at a sense operator is ordered by time. We define an *ordered* stream as follows:

Definition 6. *A stream s is ordered, iff*

1. $s = [\,]$,
2. $s = [x]$,
3. *for any* $x, y \in s$, *if* $s = [\ldots, x, \ldots, y, \ldots]$, *then* $\tau(x) \geq \tau(y)$.

$\mathcal{S} : \mathbb{S} \rightarrow \textit{Stream} \rightarrow \textit{Stream}$

$$\mathcal{S}[\![G]\!] = \lambda s \; . \begin{cases} s' & \text{if } |s| = r_f \cdot c_f \\ [\,] & \text{otherwise} \end{cases}$$

where $G = (\{f\}, \emptyset, f, f)$ with f a filter and if $|s| = r_f \cdot c_f$, then $s = [x_{r_f \cdot c_f}, \ldots, x_1]$, $s' = [x'_{r_f \cdot p_f}, \ldots, x'_1]$ and for any $1 \leq i \leq r_f$, $[x'_{i \cdot p_f}, \ldots, x'_{(i-1) \cdot p_f + 1}] = \hat{f}[x_{i \cdot c_f}, \ldots, x_{(i-1) \cdot c_f + 1}]$. Further, for any $1 \leq i \leq r_f \cdot p_f$, $\tau(x'_i) = \tau(x_{(\lfloor i/p_f \rfloor + 1) \cdot c_f}) + w_f$.

$$\mathcal{S}[\![G_1 *^s G_2]\!] = \lambda s \; . \; \mathcal{S}[\![G_2]\!]\mathcal{S}[\![G_1]\!]s$$

$$\mathcal{S}[\![G_1 ||^s G_2]\!] = \lambda s \; . \; \mathcal{S}[\![G_1]\!]s \oplus \mathcal{S}[\![G_2]\!]s$$

$\mathcal{O} : \mathbb{O} \rightarrow \textit{Stream}$

$$\mathcal{O}[\![G]\!] = [x_{r_u \cdot p_u}, \ldots, x_1]$$

where $G = (\{u\}, \emptyset, u, u)$ is a sense operator, $x_1, \ldots, x_{r_u \cdot p_u}$ are periodic sensor reading events such that $\tau(x_{i+1}) - \tau(x_i)$ is constant for any i such that $1 \leq i \leq r_f \cdot p_f - 1$.

$$\mathcal{O}[\![G_1 *^o G_2]\!] = \mathcal{S}[\![G_2]\!]\mathcal{O}[\![G_1]\!]$$

$$\mathcal{O}[\![G_1 ||^o G_2]\!] = \mathcal{O}[\![G_1]\!] \oplus \mathcal{O}[\![G_2]\!]$$

Figure 5.4: Denotational semantics for stream composites. Given an operator u, c_u is the value of $c(v,u)$, p_u is the value of $p(u,v)$, and r_u is the repetition for the operator u (Section 5.3). Given a filter f, $\hat{f}$ is the function implemented by the filter, which takes input data tokens as arguments and produces a sequence of output data tokens, and w_f is the computation time. Operator $\oplus$ is as for Definition 5.

As shown in Figure 5.4, $\mathcal{S}[\![f]\!]$ defines the semantics for the filter f. It consumes $[x_{c_f}, \ldots, x_1]$ tokens and produces $[x'_{p_f}, \ldots, x'_1]$ tokens. $\lfloor i/p_f \rfloor + 1$ is the position of firing, which produces the i-th token at the output. The relationship between the set of incoming events and the set of outgoing events is shown in Figure 5.5. To define the causality between incoming and outgoing events at any stream graph composite, we prove that for any graph composite, the outgoing stream preserves the orderedness whenever the incoming stream is ordered.

Theorem 4. *The following two holds.*

1. *For any $G \in \mathbb{S}$, if s is ordered then $\mathcal{S}[\![G]\!]s$ is ordered.*

2. *For any $G \in \mathbb{O}$, $\mathcal{O}[\![G]\!]$ is ordered.*

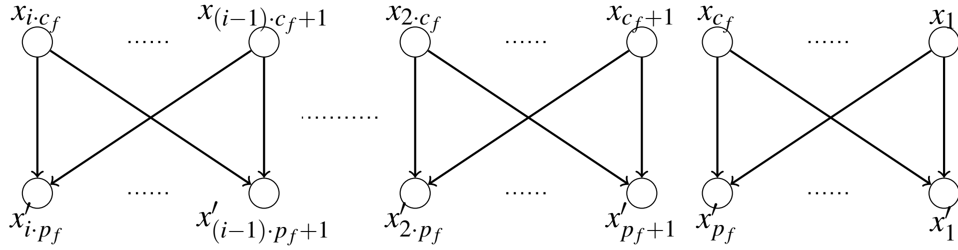


Figure 5.5: Relationship between tokens consumed and produced

Proof. For the inductive proof of order preservation in any graph composite, we first shows that the interleaving function $\oplus$ preserves the order of output stream.

Lemma 5. *Given ordered streams $s_1, s_2, s_1 \oplus s_2$ is ordered.*

Proof. We prove by induction.

1. If $s_1 = [\,]$, then $s_1 \oplus s_2 = s_2$ is ordered.

2. If $s_2 = [\,]$, then $s_1 \oplus s_2 = s_1$ is ordered.

3. If $s_1 = [x_1 : s'_1], s_2 = [x_2 : s'_2]$, then:

   - In case $\tau(x_1) \geq \tau(x_2)$, $s_1 \oplus s_2 = [x_1 : s'_1 \oplus s_2]$. By induction hypothesis, $s'_1 \oplus s_2$ is ordered and $\tau(x_1) \geq \tau(x)$, for any $x \in s'_1 \oplus s_2$, because since s_1 and s_2 are ordered,

     $$\tau(x_1) \geq \tau(x) \text{ for any } x \in s'_1 \text{ and}$$
     $$\tau(x_1) \geq \tau(x_2), \text{ and } \tau(x_2) \geq \tau(x) \text{ for any } x \in s_2.$$

     Therefore, $s_1 \oplus s_2$ is ordered.

   - In case $\tau(x_2) > \tau(x_1)$, $s_1 \oplus s_2 = [x_2 : s_1 \oplus s'_2]$. By induction hypothesis, $s_1 \oplus s'_2$ is ordered and $\tau(x_2) \geq \tau(x)$, for any $x \in s_1 \oplus s'_2$, because since s_1 and s_2 are ordered,

     $$\tau(x_2) \geq \tau(x) \text{ for any } x \in s'_2 \text{ and}$$
     $$\tau(x_2) > \tau(x_1), \text{ and } \tau(x_1) \geq \tau(x) \text{ for any } x \in s_1.$$

     Therefore, $s_1 \oplus s_2$ is ordered.

□

Next we prove Theorem 4 and provide separate proofs for the two cases.

1. We prove the first part of the lemma by induction.

   - If $G \in \mathbb{S}$ is a filter f, that is, $G = (\{f\}, \emptyset, f, f)$, then

     (a) In case $|s| = r_f \cdot c_f$, then

     $$\begin{aligned} \mathcal{S}[\![G]\!]s &= [x_{r_f \cdot p_f}, \ldots, x_1] \\ &= \hat{f}[x_{r_f \cdot c_f}, \ldots, x_{(r_f-1)\cdot c_f+1}] \\ &\quad ++ \ldots ++ \hat{f}[x_{c_f}, \ldots, x_1] \\ &= \hat{f}[x_{r_f \cdot c_f}, \ldots, x_{(r_f-1)\cdot c_f+1}] \\ &\quad ++ \ldots ++ \hat{f}[x_{i\cdot c_f}, \ldots, x_{(i-1)\cdot c_f+1}] \\ &\quad ++ \ldots ++ \hat{f}[x_{j\cdot c_f}, \ldots, x_{(j-1)\cdot c_f+1}] \\ &\quad ++ \ldots ++ \hat{f}[x_{c_f}, \ldots, x_1] \end{aligned}$$

     In the above, since s is ordered, $\tau(x_{i\cdot c_f}) \geq \tau(x_{j\cdot c_f})$, for any $1 \leq j < i \leq r_f$, therefore, $\tau(x_{i\cdot c_f}) + \Delta_f \geq \tau(x_{j\cdot c_f}) + \Delta_f$. If $\hat{f}[x_{i\cdot c_f}, \ldots, x_{(i-1)\cdot c_f+1}] = [x'_{i\cdot p_f}, \ldots, x'_{(i-1)\cdot p_f+1}]$ and $\hat{f}[x_{j\cdot c_f}, \ldots, x_{(j-1)\cdot c_f+1}] = [x'_{j\cdot p_f}, \ldots, x'_{(j-1)\cdot p_f+1}]$, then from the semantics definition,

     $$\begin{aligned} \tau(x'_{i\cdot p_f}) = \ldots = \tau(x'_{(i-1)\cdot p_f}) = \tau(x_{i\cdot c_f}) + \Delta_f, \\ \tau(x'_{j\cdot p_f}) = \ldots = \tau(x'_{(j-1)\cdot p_f}) = \tau(x_{j\cdot c_f}) + \Delta_f. \end{aligned}$$

     Since, $\tau(x_{i\cdot c_f}) + \Delta_f \geq \tau(x_{j\cdot c_f}) + \Delta_f$, $\mathcal{S}[\![G]\!]s$ is ordered.

     (b) In case $|s| \neq r_f \cdot c_f$, then $\mathcal{S}[\![G]\!]s = [\,]$ and by definition $[\,]$ is ordered.

   - If $G \in \mathbb{S}$, is $G_1 *^s G_2$, then by the semantics, $\mathcal{S}[\![G_1 *^s G_2]\!]s = \mathcal{S}[\![G_2]\!]\mathcal{S}[\![G_1]\!]s$. By induction hypothesis, $\mathcal{S}[\![G_1]\!]s'$ is ordered, for any s', and $\mathcal{S}[\![G_2]\!]s'$ is ordered, for any s'. Therefore, $\mathcal{S}[\![G_2]\!]\mathcal{S}[\![G_1]\!]s$ is ordered.

   - If $G \in \mathbb{S}$, is $G_1||^s G_2$, then by semantics, $\mathcal{S}[\![G_1||^s G_2]\!] = \lambda s\ .\ \mathcal{S}[\![f]\!]s \oplus \mathcal{S}[\![g]\!]s$. By induction hypothesis, $\mathcal{S}[\![G_1]\!]s'$ is ordered, for any s', and $\mathcal{S}[\![G_2]\!]s'$ is ordered, for any s'. From Lemma 5, $\mathcal{S}[\![G_1]\!]s \oplus \mathcal{S}[\![G_2]\!]s$ is ordered.

2. Next we prove the second part of the lemma by induction.

   - If $G \in \mathbb{O}$ is a sense operator, then by the semantics, $O[\![G]\!] = s = [x_{r_u \cdot p_u}, \ldots, x_1]$ and for any $x_i, x_j \in s$, if $i > j$, then $\tau(x_i) \geq \tau(x_j)$. Thus $O[\![G]\!]$ is ordered.

- If $G \in \mathbb{O}$ is $G_1 *^o G_2$, then by the semantics, $O[\![G_1 *^o G_2]\!] = \mathcal{S}[\![G_2]\!] O[\![G_1]\!]$. By induction hypothesis, $O[\![G_1]\!]$ is ordered and $\mathcal{S}[\![G_2]\!]s'$ is ordered, for any s'. Therefore, $\mathcal{S}[\![G_2]\!] O[\![G_1]\!]$ is ordered.
- If $G \in \mathbb{O}$ is $G_1||^o G_2$, then by the semantics, $O[\![G_1||^o G_2]\!] = O[\![G_1]\!] \oplus O[\![G_2]\!]$. By induction hypothesis, $O[\![G_1]\!]$ and $O[\![G_2]\!]$ are ordered and from Lemma 5, $O[\![G_1]\!] \oplus O[\![G_2]\!]$ is ordered.

□

The semantics defined in this section explain the execution of a query using stream and source composites. The regular semantics in Figure 5.4 are later used to compute delays with an abstract interpretation technique, where we focus only on the time information associated with an event.

5.3 Algorithm

The algorithm computes the delay, which is the timespan between causally dependent tokens at the source and the sink operators. The causality is a non-constructive definition because an infinite number of tokens arise at the sources and the sink, and hence the time delay cannot be computed by enumerating input and output tokens. To overcome this issue, we introduce the notion of a *time steady state*. The time steady state is an interval on the timeline with a number of events that re-occur ad infinitum. The existence of a time steady state is due to the periodicity of sense operators, and to the number of firings for each operator (which results in no net change in tokens). After determining the time steady state, we simulate its execution to deduce the delays of all infinite tokens. The algorithm has two parts: the first part computes the time steady state in several steps; the second part simulates the execution via an abstract interpretation framework.

5.3.1 Finding Time Steady State

In Figure 5.1 (a), each token $x_i^{s_1}$ generated at source `Sense-1` has a time-stamp, such that $\tau(x_i^{s_1}) = i \cdot \vec{t}(s_1)$, where $\vec{t}$ is a *periodicity* of the sense operator. If multiple sense operators generate stream of tokens, there is a point in time when all of them generate tokens simultaneously. We determine a time interval T such that for all sense operators

$v \in V_{se}$, there exists i and $T = i \cdot \vec{t}(v)$. The homogeneous stream graph is a special instance, where all the edges $(u,v) \in E$ have equal consumption and production rates, such that $c(u,v) = p(u,v) = 1$. In a homogeneous stream graph, all operators are periodic and tokens re-occur with a periodicity of the least common multiplier of periodicities of the sense operators. The firing of an operator and generation of output tokens depend on the time when all the input tokens are available at the input channel; therefore filters are aperiodic. For example, in Figure 5.1, `Threshold` requires two tokens in its input channel before its execution, and generates a single token at output. On the timeline, the threshold generates the first token after $t = 3$ followed by next two tokens after $t = 6$; this shows the aperiodic nature of a filter operator.

To determine the periodicity and therefore the time interval for re-occurrence of tokens in non-homogeneous stream graphs, we observe a point in time when each operator repeatedly executes a certain number of times leaving no net change of tokens on all the connecting channels. Such a point in time re-occurs when the periodicity $\vec{t}(v)$ of a sense operator v is multiplied by the number of firings $\vec{r}(v)$, such that $T = \vec{r}(v) \cdot \vec{t}(v)$, for any $v \in V_{se}$; and is referred as *time steady state*. The number of firings for all operators in the stream graph is represented as a repetition vector $\vec{r}$. In the following sections, we explain in detail two sub-steps to determine time steady state: computation of repetition vector $\vec{r}$ and of periodicity vector $\vec{t}$.

5.3.1.1 Computation of Repetition Vector

A repetition vector $\vec{r}$ is a concept adopted from a periodic schedule in a synchronous data flow [110]. The vector denotes the number of firings for each operator in a stream graph during a periodic schedule. Note that a periodic schedule exists for G if and only if there is a repetition vector $\vec{r}$ that satisfies following balance equations [71].

$$p((u,v)) \cdot \vec{r}(u) = c((u,v)) \cdot \vec{r}(v) \qquad \forall (u,v) \in E \tag{5.2}$$

$$\vec{r}(u) > 0 \qquad \forall u \in V \tag{5.3}$$

Since every operator needs to be invoked at least once in a schedule, the elements of $\vec{r}(u)$ for all operators $u \in V$ are greater than or equal to one. We define balanced compositional stream graph followed by the lemma that proves the existence of a repetition vector.

Definition 7. *A stream graph in which each composite satisfies balance equations is referred to as a balanced compositional stream graph.*

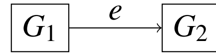
Lemma 6. *There is a repetition vector $\vec{r}$ in every balanced compositional stream graph G.*

Proof. We prove this lemma by induction, considering one base case when $G = G_u \in \mathbb{S}$ and $u \in V_{fi}$, and two inductive cases for $G = G_1 *^s G_2$ and $G = G_1 ||^s G_2$.

For the base case, $G = G_u \in \mathbb{S}$ and $u \in V_{fi}$, then $E = \emptyset$ and necessarily $\vec{r} = r(u) = k$, where k is a constant. We prove this lemma for following two inductive cases:

When $G = G_1 *^s G_2 = (V, E, s(G), t(G))$, we consider that $G_1 = (V_1, E_1, s(G_1), t(G_1))$, $G_2 = (V_2, E_2, s(G_2), t(G_2))$, and $G = (V_1 \cup V_2, E_1 \cup E_2 \cup \{(t(G_1), s(G_2))\}, s(G_1), t(G_2))$. We assume inductively that there exists $\vec{r}_1, \vec{r}_2$ such that they satisfy the set of balance equations Γ_1 and Γ_2 for graphs G_1 and G_2 respectively.

The graph $G = G_1 *^s G_2$ is represented as:



where two composites G_1 and G_2 are connected with an edge $e = (t(G_1), s(G_2))$. When the stream begins execution with n tokens on channel e, for it to return to the state with the same number of tokens on e, the number of "firings" of G_1 and G_2 denoted by $\vec{r}'(G_1)$ and $\vec{r}'(G_2)$ respectively, $\vec{r}'(G_1)$ and $\vec{r}'(G_2)$ have to satisfy the following balance equation:

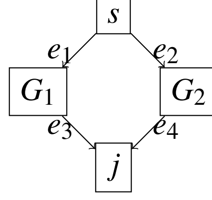
$$\vec{r}'(G_1) \cdot p(e) = \vec{r}'(G_2) \cdot c(e).$$

We construct $\vec{r}$ such that $\Gamma \vec{r} = 0$ in the following way:

$$\vec{r} = \begin{pmatrix} \vec{r}'(G_1)\vec{r}_1 \\ \vec{r}'(G_2)\vec{r}_2 \end{pmatrix}$$

When $G = G_1 ||^s G_2 = (V, E, s(G), t(G))$, we consider that $G_1 = (V_1, E_1, s(G_1), t(G_1))$, $G_2 = (V_2, E_2, s(G_2), t(G_2))$, and $G = (V_1 \cup V_2 \cup \{s, j\},\ E_1 \cup E_2 \cup\ \{(s, s(G_1)), (s, s(G_2)), (t(G_1)j), (j, s(G_2))\},\ s,\ j)$. We assume inductively that there exists $\vec{r}_1, \vec{r}_2$ such that they satisfy the set of balance equations Γ_1 and Γ_2 for graphs G_1 and G_2 respectively.

The graph $G = G_1 ||^s G_2$ is represented with following figure:



where two composites G_1 and G_2 are connected with splitter s and joiner j using four edges: $e_1 = (s, s(G_1)), e_2 = (s, s(G_2)), e_3 = (t(G_1), j)$, and $e_4 = (t(G_2), j)$. If there exists a repetition vector $\vec{r}'$ such that it satisfies following balance equations

$$\vec{r}'(s) \cdot p(e_1) = \vec{r}'(G_1) \cdot c(e_1)$$

$$\vec{r}'(s) \cdot p(e_2) = \vec{r}'(G_2) \cdot c(e_2)$$

$$\vec{r}'(G_1) \cdot p(e_3) = \vec{r}'(j) \cdot c(e_3)$$

$$\vec{r}'(G_2) \cdot p(e_4) = \vec{r}'(j) \cdot c(e_4)$$

then we construct $\vec{r}$ such that $\Gamma \vec{r} = 0$ in following way:

$$\vec{r} = \begin{pmatrix} \vec{r}'(G_1)\vec{r}_1 \\ \vec{r}'(G_2)\vec{r}_2 \\ \vec{r}'(s) \\ \vec{r}'(j) \end{pmatrix}$$

Other inductive cases can be proven similarly. □

In the example stream graph in Figure 5.1 (a), a small box next to each operator represents the number of firings in a repetition vector $\vec{r}$. The numbers are also given in the second column of Table 5.1.

5.3.1.2 Computation of Periodicity Vector

The sense operators in a stream query generate data token at regular interval on a physical timeline. As explained in our system model, each edge in a synchronous stream graph is associated with a consumption and production rate, presented as functions c and p, respectively. Except for the join operator, each operator in the stream graph has a regular pattern at which output data tokens are generated. The output data production rate solely depends on the periodicity at which sense operators generate raw data

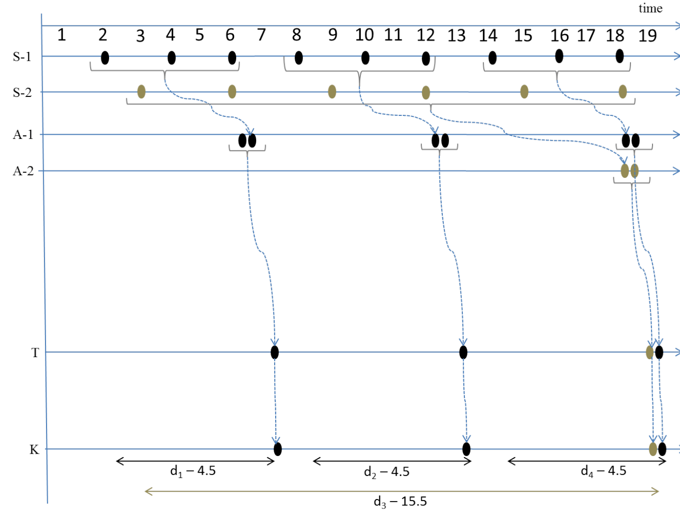


Figure 5.6: Event causality with the unsynchronised sensors from Figure 5.1(a)

tokens, and on the rate at which the operator consumes input tokens. The regularity of generating data tokens at every operator results in repetition of events in query processing (known as periodicity). The following lemma shows how to find a periodicity vector for the stream graph by propagating the periodicity of the sense operators.

Lemma 7. *There is a periodicity vector $\vec{t}$ in every compositional stream graph G.*

Proof. We prove this theorem by induction.

For the base case, $G = G_u \in \mathbb{S}$ and $u \in V_{se}$ and u generates data with known frequency f; thus G has a periodicity $\vec{t}(u) = 1/f$.

We inductively assume that a graph G with k operators has a periodicity $\vec{t}_k$, and prove that there is a periodicity in a graph with $k+1$ operators.

If the added operator u is a filter, the periodicity $\vec{t}_{k+1}(u) = \lceil \frac{c(k,k+1)}{p(k,k+1)} \rceil \vec{t}_k$.

If the added operator is a splitter s, the periodicity

$$\vec{t}_{k+1}(s) = \begin{cases} \vec{t}_k & \text{if duplicate splitter} \\ \vec{t}_k/m & \text{if round-robin splitter with } m \text{ outgoing channel} \end{cases}$$

If the added operator is a joiner j , the periodicity $\vec{t}_{k+1} = lcm(\vec{t}_k^0, \ldots, \vec{t}_k^m)$, where joiner has m incoming channels, and we assume periodicity $\vec{t}_k^0, \ldots, \vec{t}_k^m$ for each incoming channel $e_1, \ldots, e_m \in E$, respectively.

This proves that periodicity exists for every stream graph G. □

We now demonstrate the propagation of periodicity in the example stream graph for two different cases: with *synchronised* sensors (sensors having same periodicity), and *unsynchronised* sensors (different sensor periodicity). We first consider all sensors with the same periodicity, of 1 s, $\vec{t}(\texttt{Sense-1}) = \vec{t}(\texttt{Sense-2}) = 1$. We propagate periodicity over the stream graph as shown in Lemma 7, where a filter u's periodicity is computed as $\lceil \frac{c(v,u)}{p(v,u)} \rceil \cdot \vec{t}(v)$ and $(v,u) \in E$. As mentioned above, all operators except join have a predefined production and consumption rate, and thus patterns of events repeated at regular interval. Our model allows join to forward incoming data events based on their arrival time, so it is not straightforward to find the periodicity. We consider that events at join re-occur when both of its input streams have events generated at the same time; therefore, a join operator has periodicity that is lcm of the periodicity of two incoming streams. Table 5.1 shows the periodicity vector as $\vec{t}$ for the example stream graph (for both synchronised $\vec{t}_{sync}$ and unsynchronised $\vec{t}_{unsync}$).

A periodicity of join may change the number of firings required for its neighbour operators. In the example, the periodicity of `Average-1` is half the periodicity of `Average-2`, and `Average-1` operator needs to be repeated twice. The adjusted repetition vector for the operators are $\vec{r}'_{sync}(\texttt{Average-1}) = 2 \times \vec{r}_{sync}(\texttt{Average-1})$ and $\vec{r}'_{sync}(\texttt{Sense-1}) = 2 \times \vec{r}_{sync}(\texttt{Sense-1})$. Table 5.1 shows an entire new repetition vector $\vec{r}'$.

For the unsynchronised sensors (Figure 5.6), both sensors have different periodicity: $\vec{t}_{unsync}(\texttt{Sense-1}) = 2$ and $\vec{t}_{unsync}(\texttt{Sense-2}) = 3$. Table 5.1 shows the periodicity for the remaining operators as $\vec{t}_{unsync}$. For the periodicity of the join, the adjusted repetition vector for the operators is $\vec{r}'_{unsync}(\texttt{Average-1}) = 3 \times \vec{r}_{unsync}(\texttt{Average-1})$ and $\vec{r}'_{unsync}(\texttt{Sense-1}) = 3 \times \vec{r}_{unsync}(\texttt{Sense-1})$. $\vec{r}'_{unsync}$.

We next prove the existence of time steady state in any stream graph.

Theorem 5. *For every balanced compositional stream graph $G = (V,E)$, there exists a time steady state $T = \vec{r}(u) \cdot \vec{t}(u)$, for any $u \in V_{se}$.*

Proof. Lemma 6 shows that each balanced compositional stream graph has a repetition vector $\vec{r}$ and Lemma 7 ensures that there is a periodicity vector $\vec{t}$ in a stream graph. This proves that there is a time steady state

$$T = \vec{r}(u) \cdot \vec{t}(u), \text{ for any } u \in V_{se} \tag{5.4}$$

Operator	Repetition vector ($\vec{r}$)	Periodicity Vector ($\vec{t}_{sync}$)	($\vec{t}_{unsync}$)	Adjusted Repetition Vector ($\vec{r}'_{sync}$)	($\vec{r}'_{unsync}$)
`Sense-1`	3	1	2	**6**	**9**
`Sense-2`	6	1	3	6	6
`Average-1`	1	3	6	**2**	**3**
`Average-2`	1	6	18	1	1
`Join`	2	6	18	2	2
`Threshold`	1	6	18	1	1
`Sink`	1	6	18	1	1

Table 5.1: Repetition and periodicity vectors for the example stream graph in Figure 5.1 (a). Here, sync is the case when all sensors are synchronised, whereas unsync represents unsynchronised sensors.

in a stream graph. Time steady state computation uses an adjusted repletion vector $\vec{r}'$, if the periodicity vector changes the original repetition vector $\vec{r}$. □

For both cases, with synchronised and unsynchronised sensors in the example, the lengths of time steady state are $T_{sync} = \vec{r}'_{sync}(\texttt{Sense-1}) \cdot \vec{t}_{sync}(\texttt{Sense-1}) = \vec{r}'_{sync}(\texttt{Sense-2}) \cdot \vec{t}_{sync}(\texttt{Sense-2}) = 6$; and $T_{unsync} = \vec{r}'_{unsync}(\texttt{Sense-1}) \cdot \vec{t}_{unsync}(\texttt{Sense-1}) = \vec{r}'_{unsync}(\texttt{Sense-2}) \cdot \vec{t}_{unsync}(\texttt{Sense-2}) = 18$, respectively. We use the time steady state information to measure the delays during that interval, simulating the first time steady state to determine delays – the next step in the algorithm.

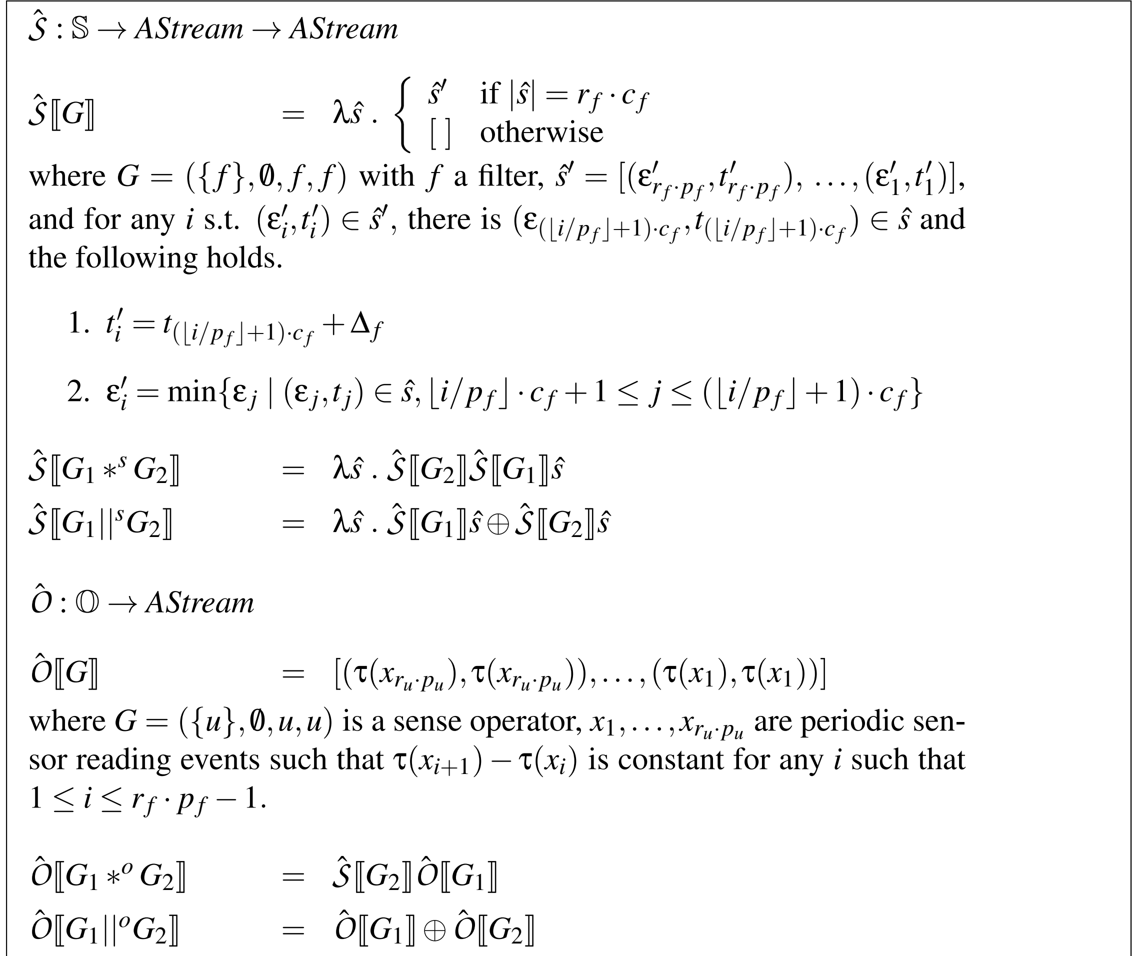
5.3.2 Simulation of Time Steady State

The second step of the algorithm simulates the first time steady state and computes delay for each event occurring at the sink. To compute delays, we focus only on the time information associated with each event and disregard actual data values. This provides an abstraction of the regular semantics of Figure 5.4, the elements of which are presented in Figure 5.7. We first define *abstract* stream inductively as follows.

Definition 8. *An abstract stream is either:*

1. $[\,]$, *or*

2. $[(\varepsilon, t) : \hat{s}]$ *with* $\hat{s}$ *an abstract stream, here* $\varepsilon = \min_{y \in \Psi(x)} \tau(y)$ *and* $t = \tau(x)$ *for some event* x*.*



$\hat{\mathcal{S}} : \mathbb{S} \rightarrow AStream \rightarrow AStream$

$$\hat{\mathcal{S}}[\![G]\!] = \lambda \hat{s}\,.\begin{cases} \hat{s}' & \text{if } |\hat{s}| = r_f \cdot c_f \\ [\,] & \text{otherwise} \end{cases}$$

where $G = (\{f\}, \emptyset, f, f)$ with f a filter, $\hat{s}' = [(\varepsilon'_{r_f \cdot p_f}, t'_{r_f \cdot p_f}), \ldots, (\varepsilon'_1, t'_1)]$, and for any i s.t. $(\varepsilon'_i, t'_i) \in \hat{s}'$, there is $(\varepsilon_{(\lfloor i/p_f \rfloor + 1)\cdot c_f}, t_{(\lfloor i/p_f \rfloor + 1)\cdot c_f}) \in \hat{s}$ and the following holds.

1. $t'_i = t_{(\lfloor i/p_f \rfloor + 1)\cdot c_f} + \Delta_f$
2. $\varepsilon'_i = \min\{\varepsilon_j \mid (\varepsilon_j, t_j) \in \hat{s}, \lfloor i/p_f \rfloor \cdot c_f + 1 \leq j \leq (\lfloor i/p_f \rfloor + 1) \cdot c_f\}$

$$\hat{\mathcal{S}}[\![G_1 *^s G_2]\!] = \lambda \hat{s}\,.\,\hat{\mathcal{S}}[\![G_2]\!]\hat{\mathcal{S}}[\![G_1]\!]\hat{s}$$

$$\hat{\mathcal{S}}[\![G_1 ||^s G_2]\!] = \lambda \hat{s}\,.\,\hat{\mathcal{S}}[\![G_1]\!]\hat{s} \oplus \hat{\mathcal{S}}[\![G_2]\!]\hat{s}$$

$\hat{\mathcal{O}} : \mathbb{O} \rightarrow AStream$

$$\hat{\mathcal{O}}[\![G]\!] = [(\tau(x_{r_u \cdot p_u}), \tau(x_{r_u \cdot p_u})), \ldots, (\tau(x_1), \tau(x_1))]$$

where $G = (\{u\}, \emptyset, u, u)$ is a sense operator, $x_1, \ldots, x_{r_u \cdot p_u}$ are periodic sensor reading events such that $\tau(x_{i+1}) - \tau(x_i)$ is constant for any i such that $1 \leq i \leq r_f \cdot p_f - 1$.

$$\hat{\mathcal{O}}[\![G_1 *^o G_2]\!] = \hat{\mathcal{S}}[\![G_2]\!]\hat{\mathcal{O}}[\![G_1]\!]$$

$$\hat{\mathcal{O}}[\![G_1 ||^o G_2]\!] = \hat{\mathcal{O}}[\![G_1]\!] \oplus \hat{\mathcal{O}}[\![G_2]\!]$$

Figure 5.7: Abstract semantics of stream composites

An ordered stream s is abstracted into an ordered abstract stream via the abstraction function $\beta : \mathit{Stream} \to \mathit{AStream}$, defined inductively as

$$\beta(s) = \begin{cases} [(\min_{y \in \Psi(x)} \tau(y), \tau(x)) : \beta(s')] & \text{if } s = [x : s'] \\ [\,] & \text{otherwise} \end{cases}$$

We assume that all abstract streams belong to the set *AStream*. *AStream* abstracts event information by disregarding actual data value and focusing on two separate pieces of time information, represented as a tuple. The first value in *AStream* tuple indicates the earliest time when an event y, which caused the current event x, was generated. The second value in the tuple indicates the time when current event x was generated. We use this abstracted time information and compute delay by subtracting the first from the second time information at any point in the stream graph.

For the abstraction, we extend $\oplus$ operation in the obvious way to operate on abstract streams. The following properties hold trivially.

Lemma 8. *For any $s \in \mathit{Stream}$, if s is ordered, then $\beta(s)$ is ordered, for any $s_1, s_2 \in \mathit{Stream}$, $\beta(s_1) \oplus \beta(s_2) = \beta(s_1 \oplus s_2)$, and $|s| = |\beta(s)|$.*

We define the abstract semantics of stream programs in Figure 5.7 and establish a property on the abstract streams produced by the abstract semantics.

Theorem 6. *For any $G \in \mathbb{O}$, $\hat{O}[\![G]\!] = \beta(O[\![G]\!])$, when we assumed that sensor readings correspond, that is, for any sensor u, with $G_u = (u, \emptyset, u, u), \beta(O[\![G_u]\!]) = \hat{O}[\![G_u]\!]$.*

As stream graphs in $\mathbb{O}$ are composed from stream graphs in $\mathbb{S}$, we first prove a related lemma on $\mathbb{S}$.

Lemma 9. *For any $G \in \mathbb{S}$ and stream s, $\beta(\mathcal{S}[\![G]\!]s) = \hat{\mathcal{S}}[\![G]\!]\beta(s)$.*

Proof. We employ structural induction.

- We assume that $G = (\{f\}, \emptyset, f, f)$. In case $|s| \neq r_f \cdot c_f$, then $\mathcal{S}[\![G]\!]s = \hat{\mathcal{S}}[\![G]\!]\beta(s) = [\,]$, and therefore $\beta(\mathcal{S}[\![G]\!]s) = \hat{\mathcal{S}}[\![G]\!]\beta(s)$. In case $|s| = r_f \cdot c_f$, from the semantics definition of filter in Figure 5.4 and 5.7, $|\mathcal{S}[\![G]\!]s| = |\hat{\mathcal{S}}[\![G]\!]\beta(s)| = r_f \cdot p_f$. We can therefore assume the following.
  - $s = [x_{r_f \cdot c_f}, \ldots, x_1]$,

- $\beta(s) = [(\varepsilon_{r_f \cdot c_f}, t_{r_f \cdot c_f}), \ldots, (\varepsilon_1, t_1)]$,
- $[x'_{r_f \cdot p_f}, \ldots, x'_1] = \mathcal{S}[\![G]\!]s$, and
- $[(\varepsilon'_{r_f \cdot p_f}, t'_{r_f \cdot p_f}), \ldots, (\varepsilon'_1, t'_1)] = \hat{\mathcal{S}}[\![G]\!]\beta(s)$.

The causality relation δ satisfies $\{(x_j,\ x'_i) \mid 1 \le i \le r_f, \lfloor i/p_f \rfloor \cdot c_f + 1 \le j \le (\lfloor i/p_f \rfloor + 1) \cdot c_f\} \subseteq \delta$.

By the definition of the function β, for $1 \le i \le r_f \cdot c_f$, $\varepsilon_i = \min_{y \in \Psi(x_i)} \tau(y)$. Since

$$\begin{aligned}
\Psi(x'_i) &= \{y \in \Omega | \delta(y, x'_i)\} \\
&= \{y \in \Omega | \delta(y, x_j), \\
&\qquad \lfloor i/p_f \rfloor \cdot c_f + 1 \le j \le (\lfloor i/p_f + 1) \cdot c_f\} \\
&\quad \cup \{x'_i\}
\end{aligned}$$

from the semantics of Figure 5.7, for any $1 \le i \le r_f \cdot p_f$,

$$\begin{aligned}
\varepsilon'_i &= \min\{\min_{y \in \Psi(x_j)} \tau(y) \mid (\varepsilon_j, t_j) \in \hat{s}, \\
&\qquad \lfloor i/p_f \rfloor \cdot c_f + 1 \le j \le (\lfloor i/p_f \rfloor + 1) \cdot c_f\} \\
&= \min\{\tau(y) \mid y \in \Psi(x_j), \\
&\qquad \lfloor i/p_f \rfloor \cdot c_f + 1 \le j \le (\lfloor i/p_f + 1) \cdot c_f\} \\
&\qquad \cup \{\tau(x'_i)\} \\
&= \min_{y \in \Psi(x'_i)} \tau(y)
\end{aligned}$$

Since $\delta(x_j, x'_i)$ holds when $\lfloor i/p_f \rfloor \cdot c_f + 1 \le j \le (\lfloor i/p_f \rfloor + 1) \cdot c_f$, $\varepsilon'_i = \min_{y \in \Psi(x_i)} \tau(y)$.

- We assume that $G = G_1 *^s G_2$ and that the property holds inductively, that is, for any $s \in \mathit{Stream}$,

  1. $\beta(\mathcal{S}[\![G_1]\!]s) = \hat{\mathcal{S}}[\![G_1]\!]\beta(s)$, and
  2. $\beta(\mathcal{S}[\![G_2]\!]s) = \hat{\mathcal{S}}[\![G_2]\!]\beta(s)$.

  Therefore, $\hat{\mathcal{S}}[\![G_1 *^s G_2]\!]\beta(s) = \hat{\mathcal{S}}[\![G_2]\!]\hat{\mathcal{S}}[\![G_1]\!]\beta(s) =$
  $\hat{\mathcal{S}}[\![G_2]\!]\beta(\mathcal{S}[\![G_1]\!]s) = \beta(\mathcal{S}[\![G_2]\!](\mathcal{S}[\![G_1]\!]s)) = \beta(\mathcal{S}[\![G_1 *^s G_2]\!]s)$.

- We assume that $G = G_1 ||^s G_2$ and that the property holds inductively, that is, for any $s \in \mathit{Stream}$,

  1. $\beta(\mathcal{S}[\![G_1]\!]s) = \hat{\mathcal{S}}[\![G_1]\!]\beta(s)$, and
  2. $\beta(\mathcal{S}[\![G_2]\!]s) = \hat{\mathcal{S}}[\![G_2]\!]\beta(s)$.

$\hat{\mathcal{S}}[\![G_1||^s G_2]\!]\beta(s) = \hat{\mathcal{S}}[\![G_1]\!]\beta(s) \oplus \hat{\mathcal{S}}[\![G_2]\!]\beta(s) = \beta(\mathcal{S}[\![G_1]\!]s) \oplus \beta(\mathcal{S}[\![G_2]\!]s)$. From Lemma 8, this equals $\beta(\mathcal{S}[\![G_1]\!]s \oplus \mathcal{S}[\![G_2]\!]s) = \beta(\mathcal{S}[\![G_1||^s G_2]\!]s)$.

□

We now prove Theorem 6.

Proof. We prove inductively.

- We assume that $G = (\{u\}, \emptyset, u, u)$. $\beta(O[\![G]\!] = \hat{O}[\![G]\!]$ holds by assumption.
- We assume that $G = G_1 *^o G_2$, and that the property holds inductively, that is, $\beta(O[\![G_1]\!]) = \hat{O}[\![G_1]\!]$. From Lemma 9, we obtain $\beta(\mathcal{S}[\![G_2]\!]s) = \hat{\mathcal{S}}[\![G_2]\!]\beta(s)$ for any $s \in \mathit{Stream}$. Now, $\beta(\mathcal{S}[\![G_1 *^o G_2]\!]) = \beta(\mathcal{S}[\![G_2]\!]O[\![G_1]\!]) = \hat{\mathcal{S}}[\![G_2]\!]\beta(O[\![G_1]\!]) = \hat{\mathcal{S}}[\![G_2]\!]\hat{O}[\![G_1]\!] = \hat{O}[\![G_1 *^o G_2]\!]$.
- We assume that $G = G_1||^o G_2$, and that the property holds inductively, that is,

  1. $\beta(O[\![G_1]\!]) = \hat{O}[\![G_1]\!]$, and
  2. $\beta(O[\![G_2]\!]) = \hat{O}[\![G_2]\!]$.

  Now, $\beta(O[\![G_1||^o G_2]\!]) = \beta(O[\![G_1]\!] \oplus O[\![G_2]\!])$. From Lemma 8, this equals $\beta(O[\![G_1]\!]) \oplus \beta(O[\![G_2]\!])$. From the induction hypothesis, this again equals $\hat{O}[\![G_1]\!] \oplus \hat{O}[\![G_2]\!] = \hat{O}[\![G_1||^o G_2]\!]$ by the semantics definition of Figure 5.4.

□

We compute the delays in Equation [5.1] using the above abstract semantics such that we compute the difference $t_i - \varepsilon_i$ for an event $x_i \in \Omega$, represented as (ε_i, t_i) with abstraction. For the example shown in Figure 5.1, we assume that all the filters take equal computation time of 0.25 s and we add them while measuring delays. Each event x in the stream graph is represented as a pair (ε, t), where $\varepsilon = \min_{y \in \psi(x)} \tau(y)$ and $t = \tau(x)$. Since we simulate delay measurement in the stream graph for the first time steady state, Figure 5.1 (b) has 6 events generated at both operators `Sense-1` and `Sense-2`. Sensor data is then forwarded to the other operators until they reach `Sink` and our semantics treat each data generation as an atomic event and propagate time information associated with each event as shown in Table 5.2. For unsynchronised sensors shown in Figure 5.6, the length of time steady state is 18 s and during the first time steady state, delays are as given in Table 5.3.

Operator	Events (ε, t) in *AStream*	Delays
`Sense-1`	(1,1) (2,2) (3,3) (4,4) (5,5) (6,6)	0,0,0,0,0,0
`Sense-2`	(1,1) (2,2) (3,3) (4,4) (5,5) (6,6)	0,0,0,0,0,0
`Average-1`	(1,3) (4,6)	2.25,2.25
`Average-2`	(1,6)	5.25
`Threshold`	(1,3) (4,6) (1,6)	2.5,2.5,5.5
`Sink`	(1,3) (4,6) (1,6)	2.5,2.5,5.5

Table 5.2: Delays with synchronised sensors (cf. Figure 5.1(b))

Operator	Events	Delays
`Sense-1`	(2,2) (4,4) (6,6) (8,8) (10,10) (12,12) (14,14) (16,16) (18,18)	0,0,0,0,0,0,0,0,0
`Sense-2`	(3,3) (6,6) (9,9) (12,12) (15,15) (18,18)	0,0,0,0,0,0
`Average-1`	(2,6) (8,12) (14,18)	4.25,4.25,4.25
`Average-2`	(3,18)	15.25
`Threshold`	(2,6) (8,12) (3,18) (14,18)	4.5,4.5,15.5,4.5
`Sink`	(2,6) (8,12) (3,18) (14,18)	4.5,4.5,15.5,4.5

Table 5.3: Delays with unsynchronised sensors (cf. Figure 5.6)

5.4 Experiments

We evaluate our approach for delays to answer the following questions:

- Does our approach efficiently measure delays?
- How does the stream graph parameters such as sensor periodicity and filter data rate affect the delay?
- How does periodicity scaling affect the efficiency and precision of delays?

We employed simulation to answer these questions. The simulator in Python was implemented, to generate stream queries, assign sensor periodicity, compute time steady state, and measure time delay for each token at the sink for one time steady state iteration.

To stress test the system, the queries in our experiments were generated by probabilistic scheme: We assign a composite (cf. Fig. 5.2) a probability (i.e. 35% for sense and filter, and 10% for other composites). The composition tree of a query was recursively constructed. The composite of a node in the composition tree was chosen randomly according to its probability and the construction ended as soon as all leaves

in the composition tree were sense operators. We ran our experiments on a Core2 Duo 2.99 GHz Intel machine with 4 GB RAM.

5.4.1 Computing Time Steady State

The efficiency of our approach depends on the length of the time steady state. Longer time steady state indicates that a larger stream is to be observed while computing delays in a stream graph. To compute the time steady state, we randomly generate stream graph queries as described above. The stream graph may or may not have a repetition vector; therefore, we check if every pair of operators in the stream graph satisfies the balance equation and have periodic schedule. The repetition vector ensures the existence of time steady state in the graph. As shown in Equation 5.4, the time steady state depends on two parameters: the repetition vector and the periodicity in the graph; and there exists a periodicity in any compositional stream graph as shown in Lemma 7. We simulate the periodicity of each sensor in the stream graph and propagate them over each composite of the graph.

Operators	Sensors	Tss-uniform	Tss-random	Tss-prime
5	2	15	30	15
7	3	10	30	700
10	3	120	120	1320
12	4	120	480	1230
15	6	30	900	23100
17	5	1800	36000	25200
19	6	2160	21600	498960
20	7	80	2400	30800
25	8	320	1920	24640
27	8	3600	14400	277200
56	18	1.1E+08	5.53E+08	8.52E+09
79	23	3.32E+08	1.99E+09	3.83E+11
97	30	7.17E+11	2.87E+12	5.52E+13

Table 5.4: Comparison of time steady state for the stream graphs with sensors having same periodicity (Tss-uniform), random periodicity (Tss-random), and periodicity prime (Tss-prime).

To measure the efficiency of the approach, we compute time steady state for three cases. In the first, we assign the same periodicity to each sensor and compute the time steady state. For the stream graph having the same repetition vector for each operator,

Application	Operators	Sensors	Tss-uniform	Tss-random	Tss-prime
Twitter real-time analytics [31]	10	5	1	24	210
Biometric data sampling [111, 112]	16	10	1	546480	6.47E+09
Radio astronomy imaging [113]	42	32	5	1.75E+09	6.47E+09
Volcanic activity detection [1, 45]	67	16	60	8038800	1.70E+08

Table 5.5: Time steady state for the real-world stream processing examples

time steady state coincides with steady state in synchronous data flow. In the second case, we assign periodicity randomly between 1 and 10 s to each sensor; in the third case, we assign periodicity to the sensors so that they are prime to each other. Table 5.4 compares the time steady state computed for the three cases. In the table, **Operators** gives the size of the stream graph, **Sensors** gives the number of sense operators, **Tss-uniform** is the time steady state, computed with same sensor periodicity, **Tss-random** is the time steady state with randomly assigned sensor periodicity, and **Tss-prime** is the time steady state with periodicity of sensors that are prime. We perform this experiment by varying the size of the stream graph.

As Table 5.4 shows, the time steady state computed with sensors of the same periodicity is the smallest among the three for all stream graphs; however, the time steady state is largest with sensors of prime periodicity. While propagating the periodicity in the stream graph, join has the periodicity that is the LCM of the periodicities for two incoming data streams. Thus, the value of Tss-uniform is minimised, Tss-prime is maximised, and Tss-random lies between the two. These facts are supported by another set of experiment, where we use real-time applications to measure time steady state; the results are in Table 5.5. We further check the effect of the size of the stream graph on the time steady state. As shown in Table 5.4, time steady state with 20 operators and 7 sensors is very small compared with other smaller graphs; this is due to the fact that the sample graph had smaller repetition vector and/or lower sensor periodicities. However, for very large stream graphs and larger number of sensors, the value of time steady state is noticeably higher, which has least impact on the scalability of our approach. Time steady state and delay are computed at server side, and WSN nodes are not involved in

the process. Hence, the computation of the delay can be performed off-line on powerful server machines. To improve the speed of time steady state computation for larger queries and network, parallelisation techniques can easily be applied during simulation phase, but the detail of these techniques is beyond the scope of this work.

5.4.2 Periodicity vs. Time Steady Sate

This section shows the impact of sensor periodicity on time steady state. We assign the same periodicity to each sensor in the stream graph and compare the time steady state by increasing periodicity from 1 s to 13 s. We perform this experiment by varying the size of the stream graph from 5 operators to 24 operators. Figure 5.8 depicts the *periodicity* vs. *time steady state* graph for five different stream graphs. As shown, the time steady state increases with sensor periodicity. From the results in Table 5.4 and Figure 5.8, our approach is most efficient with smaller and uniform sensor periodicity; however, it is least efficient with very large sensor periodicities that are prime to each other.

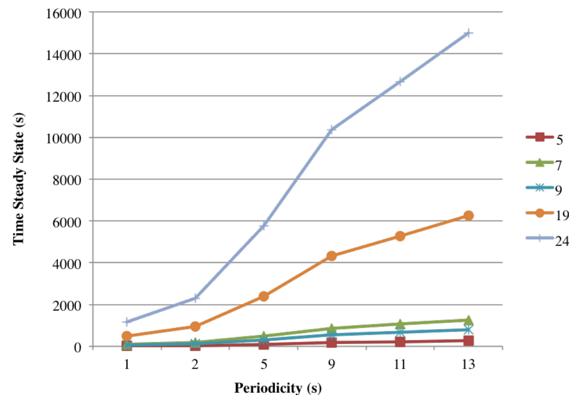


Figure 5.8: Sensor periodicity vs. time steady state

5.4.3 Filter Data Rate vs. Time Steady State

In this section, we observe the changes in time steady state with the change in filter data rates in the given stream graph. The filter data rate has a high impact on the repetition vector, and therefore on time steady state. Figure 5.9 illustrates the result of the experiment, where we keep the sensor periodicity the same for each sensor and vary the filter data rate. We collect the results for four different data rate settings,

including: a *uniform* data rate for each filter, where it consumes and produces one data token for one iteration of its operation; *highest-cons*, where each filter is set to consume a maximum number of data tokens, such as 5, and produces one data token; *highest-prod*, where each filter is set to consume one data token and produces maximum data tokens, such as 5; and *random*, which randomly assigns production and consumption rates for each filter in the stream graph. We performed these experiments by varying the size of the stream graph from 7 operators to 32 operators. Figure 5.9 gives the result for 10 different stream graphs. The time steady state with uniform filter data rate is the shortest, whereas *highest-cons* gives the longer time steady state due to the higher-repetition vector. *Highest-prod* data rate has no major impact, whereas the *random* data rate gives randomly distributed length of time steady state. From the results, our approach is most efficient when the stream graph has filter data rate as 1, and least efficient when filters have high data consumption rate.

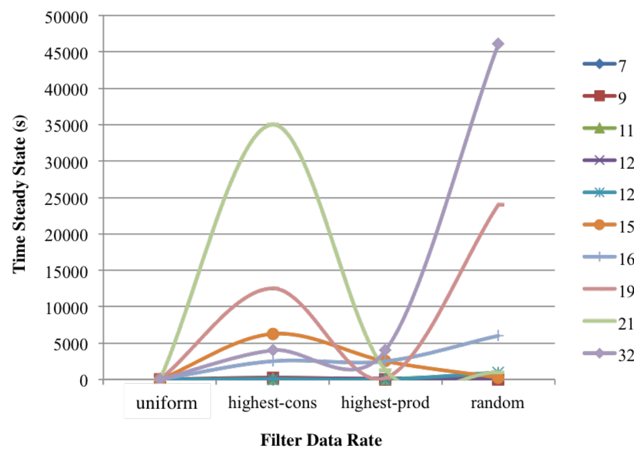


Figure 5.9: Filter data-rate vs. time steady state

5.4.4 Measuring Delay in the Stream Graph

We simulate the first time steady state, generate all the data tokens as events, propagate the time information associated with each token, and compute the delay at the sink. For stream graphs with more than one sensor, we compute the average of all delay computed for each sensor. We assume that each sensor in the stream graph starts producing data tokens at the same time.

Like time steady state, sensor periodicity has a large impact on the delay in stream

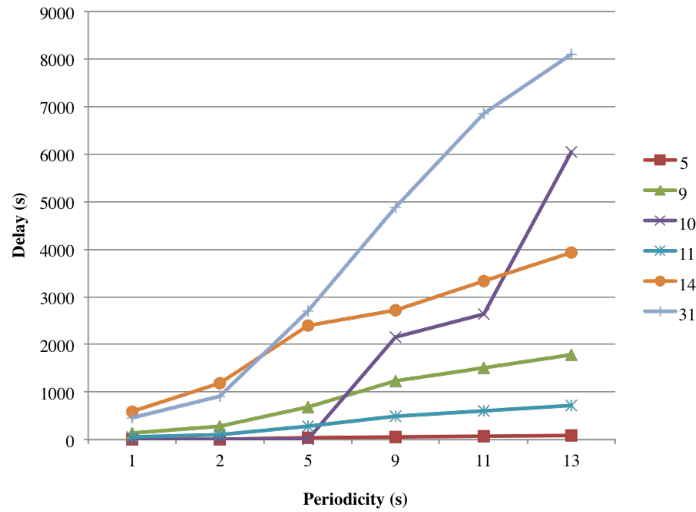


Figure 5.10: Sensor periodicity vs. time delay

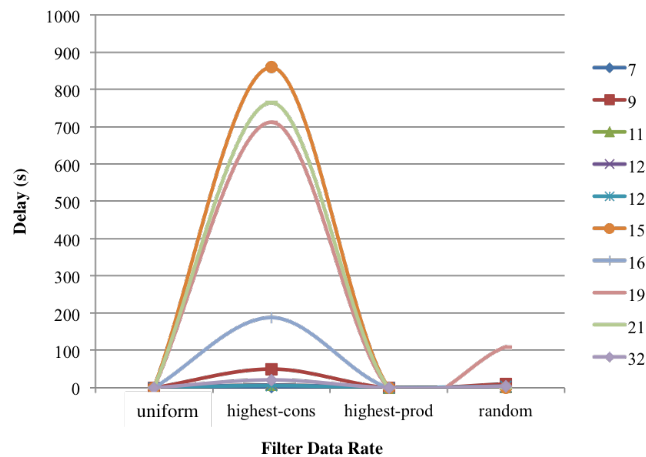


Figure 5.11: Filter data-rate vs. time delays

graph. Figure 5.10 shows the effect of sensor periodicity on the average delay by varying the size of the stream graph. We assign the same periodicity to each sensor in the stream graph, and increase the periodicity from 1 s to 13 s to observe the change in the average delay. We perform this experiment by varying the size of the stream graph from 5 to 31 operators. Figure 5.10 shows the results for six stream graphs. The observations show increased delay with increased sensor periodicity.

In addition to sensor periodicity, the data rate of filter operators dramatically increases token delay. Figure 5.11 illustrates the graph of delays for different filter data rate settings (cf. 5.4.3). We performed this experiment for the same set of input stream

graphs varying the size from 7 to 32 operators. As shown in the figure, *highest-cons* setting gives highest delays in the stream graph because each filter waits longer for the availability of incoming data tokens. In contrast, stream graphs with filters giving the same data rate or lower consumption rate (highest-prod) have the smallest delays.

5.4.5 Effect of Periodicity Scaling

As Table 5.4 shows, the length of the time steady state increases noticeably when the periodicity of sensors in the stream graph are prime to each other. Moreover, as shown in Figure 5.8, with higher sensor periodicity the time steady state increases. As mentioned before, the larger the time steady state, the more iterations it takes to compute the delay for the data tokens generated in a particular time steady state. To mitigate this issue, shortened the time steady state, thus generating fewer data tokens for delay measurement. We found that time steady state depends on two parameters in the stream graph: repetition vector and periodicity. It is difficult to alter the stream graph, and therefore the repetition vector for the given query; however, periodicity in the graph can easily be changed by scaling the periodicity of sensors. To evaluate the effect of this scaling, we first compute the time steady state by assigning *random* periodicity to the sensors; this is compared against the time steady state computed from *scaled* periodicity. We perform this experiment by varying the size of the stream graph and scaling down the periodicity of each sensor from 10% to 90%.

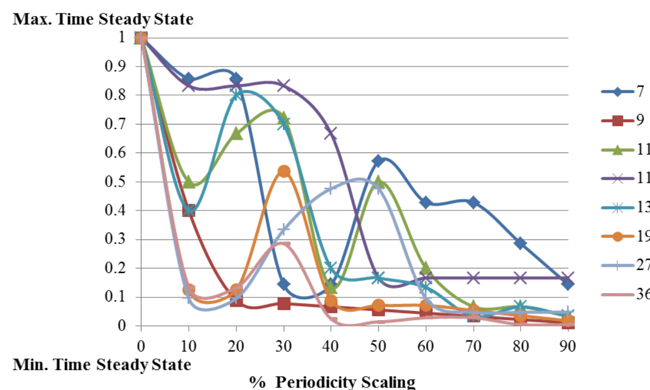


Figure 5.12: Percentage periodicity scaling vs. time steady state

Figure 5.12 shows the effect of periodicity scaling on the time steady state. The data were collected by varying the size of stream graph from 7 operators to 36 operators.

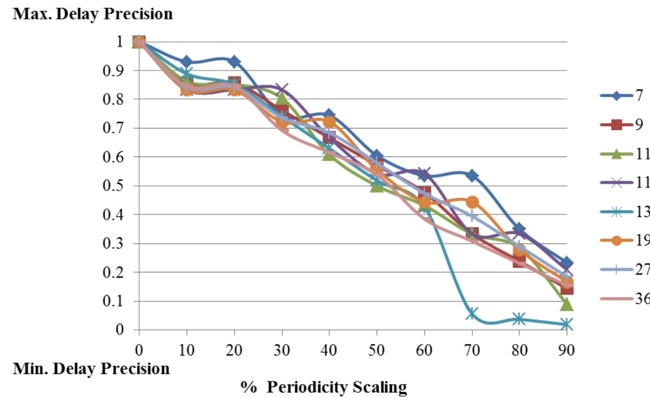


Figure 5.13: Percentage periodicity scaling vs. time delay

The figure represents the results of 8 stream graphs. The periodicity scaling for sensors reduces the time steady state; however, the decline in length of time steady state is not monotonic, because periodicity scaling sometimes results in more prime numbers, thus yielding longer time steady states. Our data show that 60% or more scaling in periodicity results in 80% shorter time steady state on average.

In addition to time steady state, the scaling in periodicity affects the precision of the delay measured for each data token during time steady state. The delay measured without applying periodicity scaling has the greatest precision. Applying periodicity scaling changes the delay measures and therefore reduces the precision. Figure 5.13 demonstrates the effect of periodicity scaling on measured delay precision. For any stream graph, scaling down sensor periodicity decreases the precision of delay measurement monotonically. For example, a 10% scaledown produces nearly optimal precision in delay measurement; a 90% scaledown results in the least precise delay. Figure 5.12 and 5.13 show that our approach of measuring delay in the stream graph using time steady state is more efficient with periodicity scaling but with a cost of precision in delay. It is a separate problem to find another efficient scheme that will to scale periodicity to produce a good trade-off between time steady state and delay precision; but this is beyond the scope of this thesis.

5.5 Chapter Summary

We have presented a new approach to computing delays in stream query processing. The approach represents stream-based query as a compositional stream graph and establishes causality relationships between each event generated during query run-time. Our model represents an event as a production of data tokens that yields time information along with actual data. We defined an interval over timeline as time steady state, and simulated the first time steady state. The experimental evaluation confirms the efficiency of the approach by varying various parameters, including sensor periodicity and operator data rates. The results of the simulations demonstrate the delay measurement by varying periodicity and operator data rate. 60% or more scaling in periodicity has an average of 80% shorter time steady state. The periodicity scaling improves the efficiency of our approach, although at the cost of a certain loss of precision in delay information.

Chapter 6

Related Work

Detailed surveys of stream data processing have been published by Stephens in [15], and by Golab and Ozsu in [94]. This chapter summarises the related work. It is structured as follows: Section 6.1 is a broad historical summary of programming models, languages, and stream processing engines. Sections 6.2 and 6.3 set out details about data processing in WSNs and the cloud, respectively. Section 6.4 lists previous work related to timeliness in stream data processing.

6.1 Stream Processing

The idea of stream processing originated from data flow and synchronous data flow approaches [16, 114, 110], and has become increasingly important for two main reasons. First, general-purpose uni-core processors have reached their limit of performance growth, and multi-core processors are becoming the industry standard. Second, applications integrated into the notion of a "stream" are becoming popular and widespread. Traditional von Neumann architecture [115] was built on a global program counter and a shared memory between program and data. However, both had become bottlenecks for highly parallel programs and restricted throughput by increasing latency. Several mechanisms were proposed to alleviate this problem; one of them was data flow architectures [114], which do not have a program counter, and where the execution of instructions is solely determined based on the availability of their operands. The asynchronous nature of the data flow model can lead to non-determinism, and the cyclic network can suffer from deadlocks. To avoid these problems, SDF was developed; the

model was first explained in Kahn's process networks (KPN) [16] and Dennis' data flow [108].

KPN assumes a network of concurrent processes that communicate over unbounded FIFO channels. A process in KPN is specified as a sequential program that executes concurrently with other processes. The KPN model is deterministic; this allows the exploitation of significant amounts of scheduling freedom while mapping processes to hardware. Processes in KPN run autonomously and provide synchronisation via the blocking read. Since control is completely distributed to the individual processes, there is no global scheduler present; thus, partitioning KPN over processors is simpler. In addition, since the communication between processes occurs over the FIFO channels, there is no notion of global memory, and therefore resource contention never occurs. Both models – KPN and Dennis' data flow model – are compared in [116], which shows how the formal semantic methods of Kahn can be adapted to Dennis' data flow. Kahn's data flow is based on a denotational notion of processes as continuous functions on infinite streams, while Dennis's data flow is based on operational notion of atomic firings of actors.

Synchronous data flow is a restricted form of KPN, where processes execute atomically. Unlike KPN, in SDF the number of data items read and produced by a network process is known at compile time. Because of deterministic processing it is possible to statically check the network processing for deadlocks. The amount of buffering required in an application can also be determined statically, which helps in determining potential scheduling and optimisation for computation. In data flow, a program is divided into pieces, known as nodes or blocks, which can execute (fire) whenever input data are available. An application in SDF is described as SDF graph – a directed graph with nodes indicating computational block and edges representing communicating data paths between nodes. SDF graphs are closely related to computation graphs [18], where each input to a block is associated with two numbers: a threshold, and the number of data tokens consumed. The threshold indicates the number of data tokens required to invoke the block, and can be different from the number of tokens consumed. In contrast with the standard SDF model, Curracurrong graphs are restricted to compositional stream graphs.

With advances in architectures and programming models, researchers found problems in using conventional imperative programming languages to implement applications on data flow hardware, in particular associated with locality. The problem was due

to certain aspects of imperative languages, such as assignments. By restricting assignments, researchers could create languages that more naturally worked well with data flow architecture and could also run more efficiently on it. These languages, which can be compiled into data flow graphs, are called data flow programming languages.

6.1.1 Stream Processing Languages

We now list the programming languages that are based on the data flow, SDF, and stream programming models. Languages such as Lucid [19], VAL [117], and SISAL [20] are based on the data flow programming model. The popularity of these languages was followed by the development of synchronous data flow languages, including LUSTRE [21], Signal [23], and ESTEREL [22], which were developed using the SDF model. Languages such as Brook [118], OpenCL [119], and CUDA [120] are parallel programming languages that were invented for heterogeneous processors, including GPGPUs. Beneath these languages is a lightweight programming model that can be easily integrated with C/C++ compilers; these languages have a single actor, known as a kernel, that runs in parallel threads over multiple processors. Due to this data parallelism, the model used in Brook, OpenCL, and CUDA is also referred to as a single program multiple data (SPMD) model. Languages subsequently developed, such as StreamIt [71] and IBM Stream processing language (SPL) [121] introduced task and data parallelism using a stream processing model. The model is based on the concept of monad from functional programming. Monad is a structure, where computations are defined as sequences of steps; this allows the programmer to build pipelines that process data in steps. We give briefs on these programming languages with examples as follows.

- *Lucid* – Lucid [19] was a popular language among all of the data flow languages developed during the 80's. Lucid was described as a functional language that was designed to enable formal proofs. Recursion was regarded as too restrictive for loop constructs, with the realisation that iteration introduced two non-mathematical features into programming: transfer and assignment. It was later apparent that Lucid's functional and single-assignment semantics were similar to those required for data flow machines; therefore, it was claimed to be a data flow language. In Lucid, the output of each computation unit is a function of its inputs, each variable is an infinite stream of values, and every function is a filter or

a transformer. The language is based on an algebra of history, where the history of changing values and history operations like first and next is recorded. The first interpreter for Lucid was pLucid.

Example – The example in Figure 6.1 generates the stream of all primes [19]. It generates the stream of natural numbers greater than 1 and passes it through a filter, which discards non-prime numbers. The operator *whenever* accomplishes the task of filtering, and the definition of `isprime` tests if a number is prime by running through all numbers greater than 2 whose squares are less than the current n.

```
prime
  where
     n whenever isprime(n)
     where
       n = 2 fby n + 1;
       isprime(n) = not(div(i,N)) asa div(i,N) or i * i > N;
       where
          N is current n;
          i = 2 fby i + 1;
          div(x,a) = a mod x eq 0;
       end
     end
  end
end
```

Figure 6.1: A Lucid example – `prime`

- *VAL* – VAL (Value-oriented Arithmetic Language) [117] is a synchronous functional language with single-assignment rule and implicit concurrency. In VAL, new values can be derived but cannot be modified; this principal enables values to be assigned to identifiers, but identifiers cannot be used as variables. This feature allows language to address certain issues arising from the automatic generation of concurrent implementation. A program in VAL consists of a series of functions, each of which could return multiple values. Loops in the language are formed similar to Lucid, and a parallel assignment construct. Some of the disadvantages of VAL are its lack of recursion, lack of general I/O, and the fact that nondeterministic programs cannot be expressed.

Example - An example in Figure 6.2 [122] computes the mean and standard deviation for parameters X, Y, and Z. VAL has the usual scalar types (integer, real, Boolean, and character) as well as arrays, records, and user-defined types. VAL is a strongly typed language and therefore each identifier like Mean and SD is specified with the data type. The function in VAL returns a tuple of values Mean and SD.

```
function stats(X, Y, Z: real) : real, real;
  let
    Mean : real := (X+Y+Z)/3;
    SD : real := sqrt((X ? Mean)**2 +
         (Y ? Mean)**2 + (Z ? Mean)**2)/3;
  in
    Mean, SD
  end
end
```

Figure 6.2: A VAL example – `stats`

- *SISAL* – The name SISAL is derived from Streams and Iteration in a Single Assignment Language [20]; it was originally written for data flow machines. SISAL is a structured functional language, providing conditional evaluation and iteration consistent with the single-assignment rule. It is a typed functional language designed for data flow computing machines, and allows recursive constructs and looping. The language, derived from VAL, offers strict semantics, implicit parallelism, and efficient array handling. SISAL is implemented for various data flow machines, including Manchester Machine, CRAY X-MP, DEC VAX, and HP.

  Example - Figure 6.3 is an example of a SISAL program; it computes the area under the curve $y = x^2$ between $x = 0.0$ and $x = 1.0$ [123]. Identifiers int, x, and y are initialised under the *for* block and modified in the *while-repeat* block. SISAL provides a way to assign a new value to a variable, using *old* keyword to access the previous value.

- *Lustre* – Lustre [21] is a formally defined and declarative synchronous language for reactive systems. Like Lucid, it is based on the description of stream processing as a system of equations. However, Lustre requires that the output at time t,

```
function Integrate {returns real)
  for initial
    int := 0.01
    y := 0.0;
    x := 0.02
  while
    x < 1.0
  repeat
    int := 0.01 * (old y + y);
    y := old x * old x;
    x := old x + 0.02
  returns
    value of sum int
  end for
end function
```

Figure 6.3: A SISAL example – `Integrate`

defined by such set of equations, depends only on inputs received either before or at time t. Those who developed Lustre refer to this property as causality. Lustre manages various clocks over which the filters execute within the program, ensuring synchronous behaviour of the language. Programs in Lustre are implemented via compilation into finite automata, providing strict analysis to detect potential deadlocks. SCADE is the later version of Lustre developed for industrial use in commercial products as a core language; SCADE is used for critical control software development.

***Example* -** An example is the piece of code in Figure 6.4, which emits the output O as soon as both inputs A and B have been received and resets the behaviour whenever the input R is received. Lustre programs are a list of modules called *nodes*; all nodes work synchronously and communicate via inputs and outputs.

- *Signal* – SIGNAL [23] is an applicative language designed to program synchronous real-time systems. Here, a process is a set of equations on elementary flows describing both data and control. The formal model provides the capability to describe systems with several clocks. Signal allows systems to be specified as block diagrams, where blocks represent components or subsystems and can be connected hierarchically to form larger blocks. In the language, reactions are

```
node EDGE(X:bool) returns (Y:bool);
let
  Y = false -> X and not pre(X);
tel

node ABRO (A,B,R:bool) returns (O: bool);
var seenA, seenB : bool;
let
  O = EDGE(seenA and seenB);
  seenA = false -> not R and (A or pre(seenA));
  seenB = false -> not R and (B or pre(seenB));
tel
```

Figure 6.4: A Lustre example – `ABRO`

transition relations in general; executing a reaction involves solving the corresponding fixpoint equation.

Example - An example in Figure 6.5 shows the addition of two integer numbers using Signal [124]. A function is referred as *process*, with inputs and outputs preceded by '?' and '!', respectively. The return statement in the example is enclosed between '|'s.

```
process ADD =
  ( ? integer A, B;
     ! integer S; )
  (| S := A + B |)
```

Figure 6.5: A Signal example – ADD

- *ESTEREL* – ESTEREL [22] was specifically designed to program reactive systems, where systems maintain permanent interaction with real-time process controllers, communication protocols, and human–machine interface drivers. While Lustre and Signal are declarative and focus primarily on specifying data flow, ESTEREL is imperative and suitable for describing control. It is a deterministic concurrent programming language where outputs of the system are in synchrony with the inputs. The notion of physical time is replaced with the notion of order: at each instant, an arbitrary number of events occur; events that occur at the same point in time are considered simultaneous. One of the ESTEREL's novel

features is *preemption* statements, which allow a clean, hierarchical description of state-machine-like behaviour.

Example - The following is an ESTEREL program. A program in ESTEREL is collection of modules with inputs and outputs. Here, the program waits for input A and B and emits output O when both inputs are available; it resets the behaviour whenever input R is available.

```
module ABRO:
input A, B, R;
output O;

loop
  [ await A || await B ];
  emit O
each R
end module
```

- *Brook* – Brook [118] was developed as a language for streaming processors. It is an ANSI C-like general-purpose stream programming language, designed to incorporate the ideas of data parallel computing and arithmetic intensity into a familiar, efficient language. Brook has different backends, including OpenMP CPU, OpenGL, DirectX 9 and AMD CTM. The language specification provides a collection of high-level stream operators that are useful for manipulating and reorganising stream data, such as grouping elements to a new stream.

  Example - Figure 6.6 shows a sample Brook code for adding two matrices over a GPU. A program in Brook has two parts: kernel definition that runs on the GPU, and host code that runs on CPU and manages kernel execution. In the example, `add` is the kernel definition that takes two 2D arrays as input and returns an output 2D array as addition of the two. The host program creates and initialises 2D arrays, copies arrays to GPU, calls the kernel, and gets the result from the GPU.

- *OpenCL* – Open Computing Language (OpenCL) [119] is an industry standard for task- and data-parallel heterogeneous computing on a variety of modern CPUs, GPUs, DSPs, and other microprocessor designs. It was proposed as a solution to

```
// kernel definition, runs on the GPU
kernel void add (float a<>, float b<>, out float c<>) {
  c = a + b;
}

// host code, runs on the CPU
int main( int argc, char** argv) {
  int i, j;
  // 2D stream declarations of 10 x 10
  float a<10,10>;
  float b<10,10>;
  float c<10,10>;

  // 2D arrays declarations of 10 x 10
  float input_a [10] [10];
  float input_b [10] [10];
  float output_c [10] [10];

  // initialise the 2D arrays
  for (i = 0; i < 10; i++) {
    for (j = 0; j < 10; j++) {
      input_a [i] [j] = (float) i;
      input_b [i] [j] = (float) j;
    }
  }

  // copy the 2D arrays to the GPU
  streamRead(a, input_a);
  streamRead(b, input_b);
  // call the kernel
  add (a, b, c);

  // get the result from the GPU
  streamWrite(c, output_c);
}
```

Figure 6.6: A Brook example – `add`

tackle many-core diversity in a unified way. OpenCL provides easy-to-use abstractions and a broad set of programming APIs. The language defines the core functionality that all devices support, as well as optional functionality for high-function devices. The key programming functions of OpenCL include managing the target devices' context, managing memory allocations, performing host–device memory transfers, and querying execution progress. An OpenCL program has two parts: compute kernels (executed on one or more OpenCL devices), and a host program that manages kernels execution. An instance of a kernel is called a work-item, and is executed for each point in the problem space; this feature guarantees very fine-grained parallelism.

Example - Figure 6.8 is the matrix addition example written in OpenCL. Like Brook, there are two parts of the program: kernel – here `VectorAdd` – which runs on GPU; and a host program that runs on CPU. The host program in OpenCL creates context on a GPU device, generates the list of GPU devices associated with context, allocates memory objects, creates the kernel, sets the work-item, and executes the kernel. At the end of kernel execution, the host program is responsible for releasing the kernel, CPU program memory, and kernel context.

- *CUDA* – Compute unified device architecture (CUDA) [120] was defined to ease the task of programming GPGPUs. The fundamental strength of the GPU is its extremely parallel nature. The CUDA programming model allows developers to exploit that parallelism by writing straightforward C code that will then run in thousands of parallel threads. A user writes a C function, called a kernel, and invokes as many threads as required to run that function. A grid of threads executes a CUDA kernel; due to GPU architecture, the threads are grouped into blocks that execute simultaneously on streaming multiprocessors. Threads within the block have access to the same shared memory and are scheduled in single instruction multiple data (SIMD) groups called warps. The language makes three key refinements to the core concepts of running kernel functions across many parallel threads: hierarchical thread blocks, shared memory, and barrier synchronisation. CUDA and OpenCL are active languages due to the development of new hardware accelerators.

  Example - A code in Figure 6.9 is an example of vector addition over GPU using CUDA. The host program initialises local parameters and allocates memory

```
__kernel void VectorAdd(__global const float* A,
 __global const float* B,
 __global float* C)
{
 // get index into global data array
 int iGID = get_global_id(0);
 // add the vector elements
 c[iGID] = a[iGID] + b[iGID];
}

// create the OpenCL context on a GPU device
 context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
  NULL, NULL, NULL);
 // get the list of GPU devices associated with context
 clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
 devices = malloc(cb);
 clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);

 // create a command-queue
 cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);

 // allocate the buffer memory objects
 memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY
               | CL_MEM_COPY_HOST_PTR,
               sizeof(cl_float4) * n, srcA, NULL);
 memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY
               | CL_MEM_COPY_HOST_PTR,
               sizeof(cl_float4) * n, srcB, NULL);
 memobjs[2] = clCreateBuffer(context, CL_MEM_READ_WRITE,
               sizeof(cl_float) * n, NULL, NULL);
 // create the program
 program = clCreateProgramWithSource(context, 1,
  (const char**)&program_source,
 NULL, NULL);
 // build the program
 clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

 // create the kernel
 kernel = clCreateKernel(program, "dot_product", NULL);
```

Figure 6.7: A OpenCL example – `VectorAdd`

```
// set the args values
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *) &memobjs[0]);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *) &memobjs[1]);
clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *) &memobjs[2]);

// set work-item dimensions
global_work_size[0] = n;
local_work_size[0]= 1;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL,
          global_work_size, 0, NULL, NULL);
// read output image
err = clEnqueueReadBuffer(cmd_queue, memobjs[2], CL_TRUE,
          0, n * sizeof(cl_float), dst,
          0, NULL, NULL);
// clean up
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmd_queue);
clReleaseContext(context);
```

Figure 6.8: A OpenCL example – `VectorAdd` continued

on the GPU; it configures threads using `dimBlock` and `dimGrid` functions, and runs the kernel along with thread configurations. After kernel execution, the host program releases memory on GPU.

- *StreamIt* – StreamIt [71] is an architecture-independent programming language specifically designed for modern high-performance streaming applications. The StreamIt language has two goals: first, to provide high-level stream abstractions that improve programmer productivity and program robustness; and second, to serve as a common machine language for grid-based processors. The language uses a set of programming constructs comprising source, sink, filter, pipeline, split-join, and feedback loop. The StreamIt language employs a synchronised model of streaming and allows a mechanism of delivering asynchronous messages (events) to the actors in a pipelined fashion. The language is an efficient combination of restricted form of SDF with events; it introduces a concept called teleport messaging for sending sporadic messages along the information wavefront. Upstream actors can send event messages to downstream actors with required iterations. In this case, compiler and scheduler can automatically take care of the required synchronisation. The SteamIt compiler performs stream-specific optimisations to achieve performance, and targets the shared-memory multicore architectures and clusters of workstations.

  Example - Using StreamIt language constructs, we can define addition of two matrices as in Figure 6.11. The program is a pipeline with a split-join along with filter *Adder*; Figure 6.10 [125] shows a corresponding stream graph. The program consists of three phases. In the first, the two vectors are distributed to each of the threads. The actual work is done in the second phase, where each thread computes the sum of each pair of elements it has received. In the final phase, the resulting vector is assembled from the partial results of each thread. The program starts by splitting each input between the *N* adder "threads". Each adder is given *k* elements at a time from each input, where the optimal value of k depends on the hardware. The results are combined by taking *k* values from each Adder so that the output vector is in the correct order. The adder takes two data items from its input queue referred as *pop* and pushes the addition value to its output queue.

- *IBM SPL* – IBM Stream processing language (SPL) [121] is the programming

```
__global__
void gpuVecAdd(float *A, float *B, float *C) {
 int tid = blockIdx.x * blockDim.x + threadIdx.x
 C[tid] = A[tid] + B[tid];
}

 int main() {
 int N = 4096;
 float *A = (float *)malloc(sizeof(float)*N);
 float *B = (float *)malloc(sizeof(float)*N);
 float *C = (float *)malloc(sizeof(float)*N)
 init(A); init(B);

 // Allocate memory on GPU
 float *d_A, *d_B, *d_C;
 cudaMalloc(&d_A, sizeof(float)*N);
 cudaMalloc(&d_B, sizeof(float)*N);
 cudaMalloc(&d_C, sizeof(float)*N);
 // Initialise memory on GPU
 cudaMemcpy(d_A, A, sizeof(float)*N, HtoD);
 cudaMemcpy(d_B, B, sizeof(float)*N, HtoD);
 // Configure threads
 dim3 dimBlock(32,1);
 dim3 dimGrid(N/32,1);

 // Run kernel on GPU
 gpuVecAdd <<< dimBlock,dimGrid >>> (d_A, d_B, d_C);
 // Copy results back to CPU
 cudaMemcpy(C, d_C, sizeof(float)*N, DtoH);

 // Deallocate memory on GPU
 cudaFree(d_A);
 cudaFree(d_B);
 cudaFree(d_C);

 free(A);
 free(B);
 free(C);
}
```

Figure 6.9: A CUDA example – `gpuVecAdd`

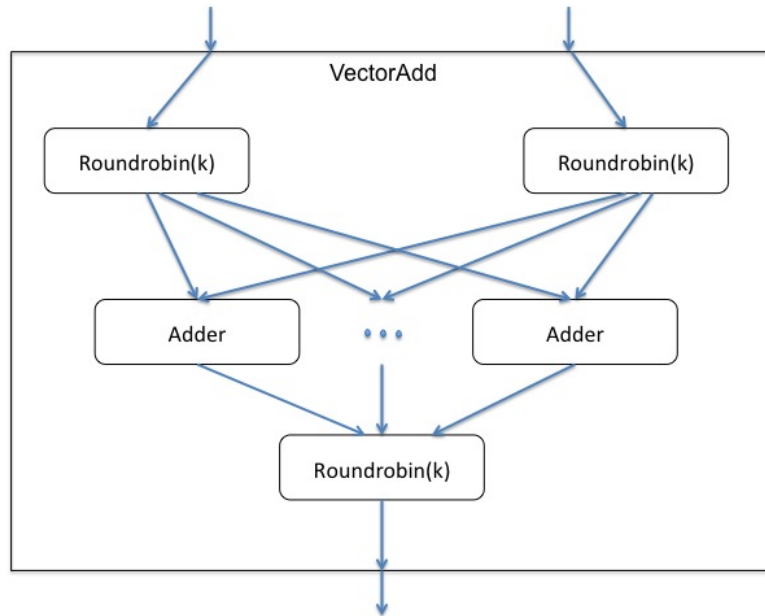


Figure 6.10: Stream graph for VectorAdd StreamIt program

language for IBM InfoSphere Streams, a platform for analysing continuous data streams with high data-transfer rates. To achieve high throughput and low latency, the platform deploys each application on a cluster of commodity servers. SPL provides an abstraction over the complexity of the distributed system by exposing a simple graph-of-operators view to the user. To facilitate the writing of well-structured and concise applications, SPL provides higher-order composite operators that modularise stream sub-graphs; it also provides a strong type system and user-defined operator models. An operator is a reusable stream transformer: each operator invocation transforms input streams into output streams. A stream connects to an operator at a port. Many operators have one input port and one output port, but operators can also have zero input ports, zero output ports, or multiple input or output ports.

Example - Figure 6.13 is a sample code written in IBM SPL [126]; this code reads a file, counts the number of lines, and writes the result in a text file. Figure 6.12 shows the stream graph for NumberedCat example, where Lines and Numbered are streams. The invocation of `FileSource` reads one line at a time from a file specified at submission-time. The invocation of Functor maintains a

```
(int,int)->int parallel VectorAdd(int N, int k) {
   parallel {
     split roundrobin(k);
     split roundrobin(k);
   }
   parallel {
     for (int i = 0; i < N; i++) {
     add Adder;
     }
   }
   join roundrobin(k);
 }
(int,int)->int filter Adder() {
   work push 1 pop 2 {
   push(pop(0) + pop(1));
   }
}
```

Figure 6.11: A StreamIt example – `VectorAdd`

state-variable *mutable*, which it increments each time a tuple arrives. SPL variables are immutable by default, so the mutable modifier is required for postscript operator ++. The output clause assigns the contents attribute of the output stream by casting the line number to a string, and concatenating it with the contents attribute of the input stream. Finally, the invocation of `FileSink` writes the results to a file named result.txt.

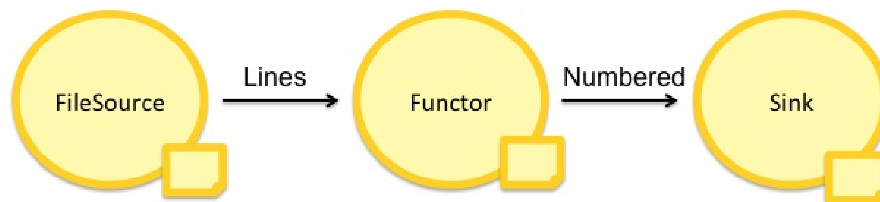


Figure 6.12: Stream graph for NumberedCat SPL program

The query language in Curracurrong is based on the notion of declarative languages. It allows the user to construct queries with compositional stream constructs; however, there is no loop construct. Like Lucid, VAL, and SISAL, Curracurrong is a functional

```
composite NumberedCat {
 graph
   stream<rstring contents> Lines = FileSource() {
     param  format        : line;
            file          : getSubmissionTimeValue("file");
   }
   stream<rstring contents> Numbered = Functor(Lines) {
     logic  state         : mutable int32 i = 0;
            onTuple Lines : i++;
     output Numbered      : contents = (rstring)i + " " + contents;
   }
   () as Sink = FileSink(Numbered) {
     param  file          : "result.txt";
            format        : line;
   }
}
```

Figure 6.13: A SPL example – `NumberedCat`

data flow language, where each actor/operator represents a function, and a compositional query represents a chain of functions. Unlike VAL and SISAL, our language does not have single-assignment restrictions and supports loops within an operator, not between operators. The language has all of the primitive data types along with arrays and records. Curracurrong does not employ the standard SDF model, unlike SDF languages like Lustre, Signal, and ESTEREL. Stream graphs in Curracurrong are restricted to structured/compositional graphs, with fixed numbers of tokens to be consumed and produced by each operator aside from join. Join semantics produces a token as soon as one is available at an input channel. However, we have defined a way to compute periodic schedule using the concept of *time steady state* (Chapter 5).

Like Brook, OpenCL, and CUDA Curracurrong supports task-parallelism by defining a query with parallel pipelines; but the current version of Curracurrong runtime does not support heterogeneous processors. Another difference between these parallel languages and Curracurrong is the number of actors/kernels defined per program or query – in OpenCL and CUDA, only one kernel is defined that runs on GPGPUs with multiple threads; in contrast, Curracurrong allows multiple operators deployed on WSN motes or cloud nodes.

Curracurrong has stream processing constructs, similar to StreamIt: source, sink, filter, and split-join. In addition to these constructs, Curracurrong's join construct merges pipeline streams from two different sources. There are two major differences between StreamIt and Curracurrong: (i) StreamIt follows SDF programming model, whereas Curracurrong does not, due to join semantics; and (ii) a programmer needs to write lengthy StreamIt code to define pipelines, in contrast to Curracurrong's easier user interface for defining query pipelines. The difference between IBM's SPL and Curracurrong includes the size of the code to define programs and constructs. SPL has 'functor' similar to 'filter' in Curracurrong.

6.1.2 Stream Processing Engines

Since 2000, several stream processing engines (SPEs) have been developed [15, 94]. Among the existing SPEs, a few are of particular interest because of their innovations. These SPEs include Cougar [69], NiagaraCQ [127], TelegraphCQ [61], STREAM [128], Borealis [28]. We give an overview of them as follows.

- *Cougar* – The Cornell Cougar research project [69] proposes an SPE that relies on a database engine. The focus is on sensor database systems used to maintain long-running queries over the data from the sensors. Such systems usually maintain stored and sensor data. The traditional centralised approach to process sensors data is in two steps: data is initially collected from all sensors into a centralised database, and queried to extract the desired information. Such a system does not scale for two reasons: a huge amount of data has to be collected, and part of these data might be not included in any query. The Cougar proposes a distributed processing of sensor data. Sensors are queried only to extract information required by queries. Depending on each query, results are evaluated at the front-end server or at the sensor network.

  Cougar distributed processing introduces a key aspect of distributed SPEs: when running distributed queries, data that has moved across nodes should be minimised by transferring only the useful information and removing the unnecessary data. This concept turns out to be important for attaining a higher throughput and more energy efficiency in distributed SPEs. Like Cougar, Curracurrong provides distributed processing, with queries evaluated in the sensor network. Unlike

Cougar, Curracurrong is not limited to the SQL aggregation type of queries. Another difference between the two systems is the way they deploy the query in the network: the Curracurrong system has an energy-efficient way of deciding operator placement.

- *NiagaraCQ* – NiagaraCQ [127] is one of the earliest research projects to attempt to overcome the limitation of database-based technology with respect to data streaming applications. The Niagara project at the University of Winsconsin, Madison, developed the eponymous data management system. NiagaraCQ uses an XML-based query language, proposing several heuristics for combining selection and join predicates that could be useful for mapping algorithms. The system proposes a novel approach to attain a scalable system for managing a large number of continuous queries simultaneously. The idea is to group queries so that overlapping computation is shared among them and redundant computation overhead is reduced.

  NiagaraCQ exploits existing querying languages to ease the programming of continuous queries. The authors introduce a command language that is an extension of ordinary XML-QL language. The command language allows the user to add queries at runtime and to specify the frequency with which data is emitted. The two main differences between NiagaraCQ and Curracurrong are: (i) language to define queries – Curracurrong has its own way to define query pipeline and does not require knowledge of XML; and (ii) NiagaraCQ groups queries to overlap computation and therefore reduces overhead, whereas Curracurrong decides energy-efficient operator placement by solving multiway cut problem with heuristics and dynamic programming.

- *TelegraphCQ* – Like NiagaraCQ, TelegraphCQ [61] has been designed and developed to address the limitation of database-supported solutions. TelegraphCQ is a continuous query processing system from UC Berkeley. The core goal of the project is to make the system highly adaptable to a changing environment. TelegraphCQ architecture is based on individual modules – units that produce and consume data – which communicate using the *Fjord* API [129]. Different types of modules include: Query processing modules, which transform incoming data tuples to output data tuples; adaptive routing modules, which distribute tuples to other modules based on some route criteria and measure system output quality

to adapt routing policies accordingly; and Ingress and Caching modules, which provide tuples to the system from external applications.

Unlike TelegraphCQ, Curracurrong decides routing policies by minimising computation and communication cost at the server; it does not have caching modules. Curracurrong modules communicate over UDP different from Fjord. Curracurrong, however, provides delay information, a feature that is unavailable in Cougar, NiagaraCQ, and TelegraphCQ.

- *STREAM* – Another stream processing system, called STREAM (the STanford stREam datA Manager) [128] has been developed at Stanford. One of the requirements of data streaming applications is to create a bridge from database solutions to data streaming solutions, providing an easy way to define queries. For this requirement, STREAM introduces a declarative language to specify queries – Continuous Query Language (CQL). CQL's relational query language is derived from that of SQL. Some of the constructs of CQL extract windows of data tuples from a stream and manage them as relations. The scheduling protocol defined in STREAM considers main memory consumptions as the primary criteria to decide which operator and how many tuples a node should process.

  CQL's query execution plans are built by defining chains of consecutive operators that effectively reduce the runtime memory; this is different from Curracurrong, which decides operator placement on the basis of minimising energy costs. Two other important differences between the two systems are: (i) language – STREAM introduces declarative CQL based on SQL, whereas Curracurrong introduces more flexible command language; and (ii) delay measurement scheme – STREAM uses wall clocks and Curracurrong employs concepts of events and causality. The STREAM project is not active since 2006.

- *Borealis* – The Borealis project [28] defines one of the first distributed SPEs. It was developed by Brandeis University, Brown University, and MIT. Borealis was built on two existing SPEs: Aurora [27] and Medusa [26]. The Aurora project was one of the first centralised SPEs, where queries are deployed on a single instance in charge of processing all of the input tuples and producing all of the output results. The project focused on the weakness of the existing database-related solutions with respect to emerging data streaming applications. Aurora introduced

a boxes-and-arrows model to define queries, providing a set of around 10 operators. A *scheduler* decides which operator should run and how many tuples it should process at any point in time. The scheduler cooperates with *QoS Monitor* and *Load Shedder*. The Qos Monitor tries to maximise the quality of the outputs produced by each query and considers aspects such as response time, tuple drops, and values produced. Load shedding is applied at any time the resource of the instance cannot cope with the incoming load.

Using Aurora, two more distributed SPEs were developed, Aurora* and Medusa [130]. Aurora* allows the creation of distributed networks of Aurora instances in the same administrative domain. As nodes belong to the same domain, there are no operational restrictions on how query operators can be distributed. Medusa was developed to connect autonomous participants. Each participant represents a set of computing devices of an entity that can contribute to running queries in several ways, such as providing data sources, providing computational resources that can be used to deploy query operators, and consuming query results.

The Borealis project came out of Aurora* and Medusa. It addresses new aspects such as dynamic revision of query results and dynamic query modification. Dynamic revision of query results provides the possibility of corrections to results produced by the query. Dynamic query modification gives the user the possibility of changing, at execution time, the parameters that define how data should be processed and which data should be forwarded to the application. The Borealis project has been inactive since 2008.

In the Borealis system, a user writes XML files to define a query. This contrasts with Curracurrong, which specifies much shorter queries. Some of the other complexity issues in writing a Borealis query are: choice of the predefined box type, definition of input-output streams, manual type inference over I/O streams, manual definition of connection points between boxes, and the requirement of a separate XML file for query deployment. The latest version of the Borealis system (2008) supports a graphical editor to define queries. However, the literature reports that frequent users of a programming system prefer a traditional textual interface over the graphical [73]. Curracurrong offers adequate flexibility

in defining domain-specific queries, using built-in and user-defined query operators. Unlike Borealis, Curracurrong provides an automatic type inference feature that reduces the burden of the application developer. The query deployment in Borealis requires the user to write a separate XML file, while Curracurrong decides deployment automatically.

6.2 Stream Data Processing in WSN

Several sensor programming models have been developed with varying capabilities for use in sensor networking applications. One of the key challenges in designing a programming environment for WSNs is finding a trade-off between ease of use, efficiency, and providing adequate flexibility for expressing powerful distributed computations. Each WSN programming model addresses these problems differently.

6.2.1 Productivity

Node-centric abstraction provides programming model at the level of individual nodes. The approach is typically supported by a system-level programming language, for example, nesC [55]. nesC is the most widely used programming language for developing WSNs. Both nesC and its extension galsC [66] provide the flexibility to write complex applications; but they suffer from the common problem of system-level programming in which the programmer has to deal with low-level aspects of WSN such as messaging, data caches and routing protocols. This requires application developers to have embedded system programming skills.

Some of the network-centric models are query processing systems, such as TinyDB [47], Cougar [69], and MaD-WiSe [30], which are database-inspired declarative query languages and provide an abstraction in terms of SQL-like queries. The other network-centric models include Regiment [56] and Kairos [67]. Regiment uses the concept of functional reactive programming (FRP) for high-level abstraction. Users write their program in Haskell-like functional programming manner, whereas Kairos' programming model specifies the global behaviour of a WSN computation using a centralised approach. In contrast, Curracurrong has a high-level programming model that uses stream graph to represent queries offering development productivity to the end

user. Curracurrong queries are expressed succinctly through the textual and visual concepts in the stream graph.

6.2.2 Flexibility

TinyDB [47] and Cougar [69] are pioneering approaches in WSN query languages. They are designed for the union of query results processed in parallel by the sensor nodes. All nodes locally execute the same query, and the results of each query are merged as they flow from the sensor node to the sink. However, TinyDB lacks flexibility: it cannot relate and compare the in-network data acquired by different sensors – for example, checking if the temperature reading from one sensor node is lower than from some other sensor node reading. To implement a new query operator or feature in TinyDB requires substantial changes in the language parser and query engine. Curracurrong proposes a query language by using abstraction provided in terms of stream operators. The stream operators define only the application logic along with input and output data types. The stream-based programming model allows the user to use built-in operators and extend functionality by writing customised operators for complex queries. To extend the functionality and expressiveness of the applications, the user needs to focus only on defining operators. Curracurrong query processing system and runtime environment take care of all system-related non-functional requirements like query parsing, the use of the placement algorithm, scheduling, and energy optimisation – this offers significant flexibility to the user.

The Curracurrong query processing system uses the concept of stream for data communication between query operators. [15] reports the results of a stream processing data model survey. Aurora [27] is a centralised stream processor that models a stream as an append-only sequence of data tuples. There is no direct control on the ordering and regularity of data arrival from multiple sources. This model was extended in Aurora* [26] and Medusa [26] with distributed stream processing features. Borealis [28] is a generalised second-generation stream processing engine that leverages and extends the Aurora and Medusa designs. The Borealis engine, with the extended and complex stream model supports features like revision of data tuple and dynamic query modification. Another distributed stream management for WSN, MaD-WiSe [30], was recently introduced. However, it uses SQL-based querying, which limits stream management to aggregation and does not support flexible application design. Approaches discussed

in Aurora* and Borealis address issues for generic data stream management, whereas Curracurrong considers particular aspects of WSN, such as flexibility in writing complex application and resource-efficiency. For resource-constrained WSN applications, Curracurrong offers a simpler programming system that achieves the level of flexibility of the complex streams in Borealis.

6.2.3 Efficiency

Other related work in macro-programming and query processing includes the in-network data processing for achieving energy efficiency. Instead of directly pushing the raw sensor data to the sink node (forwarding), aggregation is performed at the sensor nodes before sending the results back to the sink. The idea of in-network aggregation in a sensor network to minimise the communication was first proposed in Tiny AGgregation (TAG) [77]. Later approaches include Regiment [56], Kairos [67], [131], MaD-WiSe [30], and [50]. The in-network processing approach requires a good trade-off between data quality and energy consumption for long-running query. The task mapping problem in Regiment is solved during compilation by performing code normalisation, analysis, and optimisation, in many stages. In Kairos (later known as Pleiades [24]), the authors address the problem of how actions are distributed onto nodes, by partitioning the program's control flow graph (CFG) into *nodecuts* before placing them on individual nodes. They apply a reaching definition-based compilation technique to partition the CFG and heuristic to minimise the total number of edges in a program's CFG that cross from one nodecut to another. Other task allocation approaches include [131], which has an objective to find a task allocation with a balanced energy consumption for WSN applications. They model the communication over channels as constraints of integer linear programming (ILP) and use heuristic algorithm to solve it. The optimisations in Regiment, Kairos, and [131] concern the running time and do not consider energy efficiency. Different from these three systems, Curracurrong proposes an operator placement approach that aims to minimise the energy spent on each sensor, based on local computation and communication costs. This non-trivial placement supports energy-efficient long-running queries. Curracurrong's language abstraction as a stream operator allows us to develop the placement approach as a uniform optimisation strategy that remains unaffected with the extension of operator library.

MaD-WiSe uses an algebraic optimisation approach based on transformation rules

applied to a query execution plan to produce a semantically equivalent plan with lower cost. Their optimiser applies a greedy approach to choose the operator ordering, based on various criteria that give optimal performance. Another task mapping approach is [50], where the authors provide mathematical formulations for the task-mapping problem for both energy balance and total energy spent. They suggest mixed integer programming (MIP) formulations, which give optimal results with long run-time. They provide a task-mapping approach with greedy heuristics that has shorter run-time than MIP. In contrast with MaD-WiSe and [50], we formalise the operator placement problem as a multiway cut and solve it with an isolating cut heuristic [82], which gives nearly optimal solutions with better worst-case time complexity than the MIP approach proposed in [50].

6.3 Stream Data Processing in Cloud

Application data, particularly application or web-server logs are generated at very high rates and volumes. The typical approach to dealing with these large amounts of data is to collect them into files in a BLOB store like Amazon Simple Storage Service (S3) or in a large distributed file system like Hadoop HDFS for future processing. Some solutions may choose to store files in a more structured format such as a column store like HBase, Accumulo, or Cassandra. Once these data are stored in the file system, they are then processed using a batch-oriented processing mechanism like Hadoop MapReduce.

The problem with this approach is that log and application data are most often unstructured or semi-structured, and further processing and cleansing is needed to extract useful statistics. Transporting and storing data logs into a data store or file system adds significant latency to the data processing and further delays downstream processing. Nevertheless, the batch-oriented, collect-store-and-process approach is the predominant processing pattern in use across most organisations large and small. Some popular and commercially used cloud/distributed stream processing systems are explained below.

- Twitter's Storm – The Storm [31] project focuses on the distribution, parallelisation, and fault tolerance guarantees while leaving the specification of how to process tuples to the final user. In Storm, queries can be expressed using the boxes-and-arrows model; the system will take care of distributing such operators among the available instances, but will leave the task of defining how to process

data to the final user. With Storm, queries are expressed via two kind of objects, Spouts and Bolts. Spout nodes are responsible for generating the system input streams, while Bolts are in charge of processing those streams and generate output results. The Storm project can be seen as the data streaming alternative to the map-reduce [92] paradigm. As for the map-reduce paradigm, the user task is to define the functions that are used to read and generate data, and to process it in parallel while the system is in charge of managing the several multi-threaded instances. Storm relies on Zookeeper servers to maintain the state of distributed setups. Storm was later acquired by Twitter in 2013.

- IBM Infosphere – Infosphere SPE [132] represents another alternative parallel-distributed SPE. Two interesting aspects of Infosphere are its capability for processing several input formats such as XML, text, voice, and video; and its query language SPADE [133]. SPADE is a declarative language that allows the user to define queries along with how to distribute or parallelise the query operators. The language allows user-defined operators for complex functions. Infosphere provides an enterprise-class foundation for information-intensive projects, providing the performance, scalability, reliability and acceleration needed to simplify difficult challenges and deliver trusted information to your business faster.

- Yahoo S4 – The S4 [33] SPE represents a comprehensive free data streaming solution that provides distributed processing and fault tolerance capabilities. It is a Java-based solution that, similarly to STORM, relies on the user for the definition of classes for processing and producing streams tuples. As well, S4 relies on Zookeeper to maintain the state of a distributed setup. S4 allows the parallel execution of data streaming operators referred as symmetrical deployment. S4 provides a fault tolerance protocol based on state check-pointing and leads to minimal state loss.

- LinkedIn Samza – Apache Samza [32] is a stream processing framework developed by LinkedIn. Samza relies on Apache Yarn for distributed resource allocation and scheduling. Samza uses Apache Kafka for distributed message brokering, and provides an API for creating and running stream tasks on a cluster managed by Yarn. The system is optimised for handling large messages, and

provides file system persistence for messages. Samza provides message delivery guarantees using upstream backup techniques for handling message failures. When a node fails and new node takes over, the new node starts reading from the upstream broker stream from the maker that the failed node set. Because of this, Samza achieves repeating recovery for deterministic networks and divergent recovery for non-deterministic networks. Samza tasks are written with Java programming language. The code specifies how to process the messages.

6.4 Real-time Stream processing

The problems of timeliness and QoS guarantee in WSN applications have drawn the attention of researchers for many years. The recent survey of real-time data management in WSN is presented in [134]. The article identifies main specifications of real-time data management in WSN and presents the available solutions for the same. Another article [135] outlines eight requirements that a system should meet for real-time stream data processing applications. In database, the term real-time means that the transaction must run in a fixed time interval [136]. To satisfy time constraints, the data structure should include: timestamp, the time when data was generated or observed; and an absolute validity interval (AVI – the time interval during which the data are considered valid) [137]. In distributed environments, the challenge is to measure the time interval as end-to-end delay.

Existing approaches to determine end-to-end delay include probabilistic distribution [104, 105]. In [104], the authors propose a comprehensive analytical model that characterises end-to-end delay distribution in WSN for both deterministic and random network topologies. They highlight relationships between network parameters and delay distribution in multi-hop WSNs. [105] introduces a probabilistic analysis of delay for broadcast networks considering medium access control protocol. Other related work in timeliness includes probabilistic delay bounds presented in [138, 139]. In these approaches, the worst-case performance bounds are analysed, which have limited applicability in WSN due to randomness in communication. Work on real-time queuing theory [107] has introduced a stochastic model for unreliable networks that combines real-time theory and queuing theory. This approach uses scheduling policy and deadlines associated with each packet, in relation to two scheduling policies: EDF and FIFO.

Technique/ System	Stream processing system	Delay measurement	Method	Ahead of time
Wang 2012 [104]	No	Yes	Comprehensive analytical model	Yes
Netus 2005 [105]	No	Yes	Probabilistic analysis	Yes
Burchard 2006 [138]	No	Yes	Probabilistic delay bounds	Yes
Yeung 2001 [107]	No	Yes	Real-time queuing theory	Yes
Cougar	Yes	No	–	–
NiagaraCQ	Yes	No	–	–
TelegraphCQ	Yes	No	–	–
STREAM	Yes	Yes	Uses wall clock and skew bound	No
Borealis	Yes	Yes	Uses wall clock and timestamps	No
Storm	Yes	Yes	Complete and Processing Latency	No
Curracurrong	Yes	Yes	Event Causality	Yes

Table 6.1: Comparison of Curracurrong with related systems

Unlike existing approaches to measuring end-to-end delays, our approach uses the concept of event causality. The concept of events, causality, event hierarchies, and event patterns and rules are explained in in [140]. The event causality in actor network was also explained in the article [109], which provides a unique formal modelling of communicating actors. The model concerns *causality* – that is, the propagation of events across the system. An actor has a semantics called its *causality* interface, which relates events (a pair of time and data values) of the input with events of the output. The system semantics are algebraically defined in terms of the semantics of each component, connected appropriately by edges of the graph. The graph's semantics are defined by a smallest set of time-stamped events that can occur in the system. Since the approach measures delay for each individual data token that reaches the sink operator, it provides precise time information; whereas other existing approaches in [104, 105, 138] provide probabilistic delay distribution or delay bounds. Hence, our approach can be used to determine point-in-time data freshness/staleness during processing.

Some related stream processing systems like Cougar [69], NiagaraCQ [127], and TelegraphCQ [61] do not provide the notion of delay measurements. STREAM [29] and

the latest version of Borealis [28] use the wall clock time at source and sink to measure data packet latencies, and requires skew bounds for all network nodes. Another recent stream processing concept, Storm [31], provides a user interface that tabulates complete latency and processing latency when a data packet reaches the sink. Unlike these stream processing systems, our approach uses timestamps and the novel concept of causality to provide delay measures, not only at the sink but at any operator in a stream graph. Table 6.1 compares the Curracurrong system with related systems.

Chapter 7

Conclusion and Future Work

The thesis has presented Curracurrong, a stream processing system for distributed environments, with easy-to-use query language, energy-efficient operator placement, and timeliness. Previous research into stream data processing over the past decade has attempted to design a programming model that achieves combination of productivity, flexibility, and energy-efficiency. We have responded to the challenge of finding a good trade-off between these three by developing Curracurrong – a stream processing system, for distributed environments. The system is easy to use because of high-level query language and automatic operator placement scheme. An extendable stream operator library provides flexibility by supporting a wide range of application systems. Efficient operator placement algorithms offer reductions in energy consumption and support the dynamic nature of queries in WSN and cloud. For time-critical stream processing applications, the system provides timeliness by measuring end-to-end delays. We conclude the work by explaining the lessons learnt by addressing the challenges and achieving our goals.

(L1): Designing a query language for a wide range of distributed systems is very challenging

To offer a user-friendly language and thus increasing user productivity, we defined a high-level query language for a wide range of stream processing applications. During the language design, we faced software engineering and computer science challenges. To address these, we employed the SDF programming model with restrictions. Curracurrong query language is represented by compositional/structured stream graphs, with vertices as stream operators and edges as

unidirectional communication FIFO channels. Each stream graph is a composite of five types of stream operators: sense, sink, filter, split, and join. The query language uses the notion of declarative languages and has no loops. To provide flexibility to users in developing wide range of applications, we built a stream operator library comprising various aggregator and data mining operators; the library is easily extendible when required by customised applications. We compared our query language with that of existing systems and confirmed that users of Curracurrong query language write queries more efficiently than when writing lines of code or XML files.

(**L2**): **Energy-efficient operator placement is an NP-hard problem and requires sophisticated algorithms to solve in polynomial time**

For WSN applications, it is important that the system provide enough energy efficiency for network longevity. Similarly, in cloud applications, users pay for their use of resources like communication bandwidth; therefore, the challenge was to develop a system that offers the efficient use of such resources. We introduced a novel algorithm that minimises computation and communication costs with optimal query operator placement in the network. We modelled the placement problem as multiway cut, and proved that the placement problem is NP-hard. We used Dahlhaus' algorithm, which provides a simple combinatorial isolation heuristic with a 2-approximate solution. Existing systems like Borealis and Apache Storm require the user to define deployment while writing a query; in contrast, we have provided an automated operator placement with energy efficiency. To confirm the efficiency of our algorithm, we used SunSPOT sensors and an external power supply, and compared the energy consumption readings with those of standard data-forwarding placement. We compared the running time of our heuristic algorithm with the running time of ILP solver; the results showed that ILP solver is much slower than the heuristic.

The dynamic nature of the distributed environment was another challenge while designing our approach to operator placement. The system required uninterrupted query execution when a network node is unavailable or one of the queries has stopped its execution. To find an energy-efficient placement for migrating operators is again a NP-hard problem. We explained the graph composites and defined what we called the migrating operator placement problem (MOPP). To

solve MOPP, we developed a sophisticated algorithm that builds compositional trees for the running queries and finds the optimal operator placement (at regular intervals) using dynamic programming. We wrote a simulator to evaluate the algorithm, and our findings show that our approach is more efficient than the default operator placement scheme, data forwarding. To improve the running time of our algorithm, we developed and applied a locality heuristic that resulted in nearly optimal placement with shorter running time.

(**L3**): **Measuring end-to-end delays in continuous query processing is complex and requires detailed analysis of the system model**

Real-time data processing is essential in many time-critical stream-based applications, including disaster area monitoring, health monitoring, and intrusion detection. Some existing approaches use probability distribution and first-order statistics to measure end-to-end delays in stream data processing. In contrast, our aim was to find a comprehensive way that statically determines precise delays for each data token. The challenging part was that delays are not the same for each data token that reaches the sink; this was due to the design of system model, where sense operators may have different periodicity and filter operators have non-uniform data rates. We employed the concept of event causality to devise an algorithm that statically measures delays in the stream graph. The algorithm finds an interval over timeline – time steady state – and simulates the first such interval to measure delays. We defined denotational semantics for stream processing, and used abstract interpretation technique during the simulation phase in algorithm. We evaluated our approach with a set of experiments in which we computed time steady state with strategically varied filter data rates and sense periodicities. The results showed an association between (i) longer time steady state; (ii) non-uniform filter data rates; and (iii) sense periodicities that are prime to each other. We measured the effect of periodicity scaling over time steady state, confirming the association between shorter time steady state and less precision in delays.

(**L4**): **The system requires minimal changes to run under different technologies**

To confirm the adaptability of the Curracurrong system, we made changes in system design and communication technology so that it would run on a cloud-based

cluster of nodes. We developed a simple cluster monitoring application to demonstrate the technology and used parts to evaluate the performance characteristics of the system. This confirms that the system requires minimal adaptation for use with different technologies.

7.1 Opportunities for Future Work

Curracurrong is now open-source under the Apache 2.0 licence, available in the GitHub repository. It is ready to run lightweight applications on a cloud cluster. Although the system presented here is capable of running lightweight applications in WSN and cloud, a limitation of the current version is the messaging system, which uses UDP; the centralised mechanism of the system therefore suffers from unreliable communication with large networks (see Integration with other systems and technologies, below). The system could be further extended in the following research directions.

- **Improved language expressiveness** – At the moment Curracurrong queries are static; for example, if there is a requirement to change the threshold value in disaster monitoring applications, the user stops the running query and changes the threshold parameter. This requires redundant placement computation and redeployment of the operators. There is significant research potential for improving the language so that a user writes a query that modifies itself dynamically. The system would modify the specific query parameters while the previous version of the query was still running. The changes would require new query commands, the extension of query parser, and a set of new administrative commands.
- **Failure tolerance** – Another future direction would be to exploit the failure tolerance and estimate the data stream during the failure of node or unavailability of a communication channel. The unavailability of node or network interrupts the stream of data required for computation. In another case, the nodes may generate erroneous data; such missing or incorrect data will end up in inaccurate results. Using the concept of probability distribution and machine learning would enable us to deal with these uncertain data, producing results with possible minimum deviation. This would be achieved by extending the type system, where newly introduced uncertain data type would enable a model to be built using probabilistic

distributions. When expected data are missing in the stream, the model would insert the new data so that the computations would generate results with guaranteed minimal deviation.

- **Robustness** – Another possibility would be to make the system robust in two steps. The first step would be to create a multi-server system. In the current version, a single server is used to manage query deployment, send out administrative commands, and collect results. In case of server failure, entire system will fail as well. To tackle this issue, the system could introduce multiple servers that run independently and take the query operators' workload from the nearest failed server. In the second step, backup and recovery techniques could be developed for each participating node. In case of node failure, a backup node would replace the failed node.

- **Integration with other systems and technologies** – The current version of the system uses the Java serialisation format; this could be extended to support serialisation formats like Apache Avro, Google Protocol Buffers, and Apache Thrift. This extension would enable users to define custom serialisation formats so that Curracurrong would seamlessly interact with large web-scale systems like Apache Hadoop. In future versions we intend to support not just UDP, but configuration-driven messaging systems like ZeroMQ; this would enable easier integration with legacy and heterogeneous system deployments. Messaging systems like ZeroMQ-, AMQP- and JMS-compliant systems are easy to deploy, maintain, and administer. In addition, they support store and forward mechanisms and guaranteed message delivery modes. Another research direction could be to develop the system so that it can be used for mobile, internet of things, and wearable technologies – made possible by the flexibility and adaptability of Curracurrong's programming model.

Bibliography

[1] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh, "Monitoring volcanic eruptions with a wireless sensor network," in *Proc. Second European Workshop on Wireless Sensor Networks (EWSN '05)*, Citeseer, 2005.

[2] Z. Chaczko and F. Ahmad, "Wireless sensor network based system for fire endangered areas," in *Information Technology and Applications, 2005. ICITA 2005. Third International Conference on*, vol. 2, pp. 203–207, IEEE, 2005.

[3] R. Jafari, A. Encarnacao, A. Zahoory, F. Dabiri, H. Noshadi, and M. Sarrafzadeh, "Wireless sensor networks for health monitoring," in *Mobile and Ubiquitous Systems: Networking and Services, 2005. MobiQuitous 2005. The Second Annual International Conference on*, pp. 479–481, 2005.

[4] Y. Zhu and D. Shasha, "Statstream: Statistical monitoring of thousands of data streams in real time," in *Proceedings of the 28th International Conference on Very Large Data Bases*, pp. 358–369, VLDB Endowment, 2002.

[5] S. Babu, L. Subramanian, and J. Widom, "A data stream management system for network traffic management," in *In Proceedings of Workshop on Network-Related Data Management (NRDM 2001)*, 2001.

[6] Z. Barney, "Counting the terrible cost of a state burning," *Fairfax Media (Melbourne: The Age)*, 2009.

[7] "Research - Bushfire Cooperative Research Centre." http://www.bushfirecrc.com/research, 2013.

[8] P. Johnston, G. Milne, and B. Chu, "Coupling bushfire spread with a sensor network detection system model."

http://www.bushfirecrc.com/managed/resource/paul-johnston-george-milne-bobby-chu.pdf.

[9] B. Lo, S. Thiemjarus, R. King, and G.-Z. Yang, "Body sensor network-a wireless sensor platform for pervasive healthcare monitoring," in *The 3rd International Conference on Pervasive Computing*, vol. 13, pp. 77–80, 2005.

[10] E. Jovanov, D. Raskovic, J. Price, A. Krishnamurthy, J. Chapman, and A. Moore, "Patient monitoring using personal area networks of wireless intelligent sensors," *Biomedical Sciences Instrumentation*, vol. 37, pp. 373–378, 2001.

[11] G. Werner-Allen, K. Lorincz, M. Welsh, O. Marcillo, J. Johnson, M. Ruiz, and J. Lees, "Deploying a wireless sensor network on an active volcano," *IEEE Internet Computing*, vol. 10, no. 2, pp. 18–25, 2006.

[12] "Amazon cloudwatch." http://aws.amazon.com/cloudwatch/, 2013.

[13] "AzureWatch - Monitoring and Autoscaling of Azure." http://www.paraleap.com/azurewatch, 2014.

[14] "Nagios Is The Industry Standard In IT Infrastructure Monitoring." http://www.nagios.org/, 2013.

[15] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, no. 7, pp. 491–541, 1997.

[16] G. Kahn, "The semantics of a simple language for parallel programming," *Information processing*, vol. 74, pp. 471–475, 1974.

[17] E. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on computers*, vol. 36, no. 1, pp. 24–35, 1987.

[18] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: Determinacy, termination, queueing," *SIAM Journal on Applied Mathematics*, vol. 14, no. 6, pp. 1390–1411, 1966.

[19] W. W. Wadge and E. A. Ashcroft, "Lucid, the dataflow programming language1," 1985.

[20] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, "A report on the sisal language project," *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, pp. 349–366, 1990.

[21] D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre: A declarative language for programming synchronous systems," in *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, vol. 178, p. 188, 1987.

[22] G. Berry and G. Gonthier, "The esterel synchronous programming language: Design, semantics, implementation," *Science of computer programming*, vol. 19, no. 2, pp. 87–152, 1992.

[23] A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous programming with events and relations: the signal language and its semantics," *Science of computer programming*, vol. 16, no. 2, pp. 103–149, 1991.

[24] D. May, "Occam 2 language definition," *Inmos Ltd*, vol. 72, 1987.

[25] J. Armstrong, *Programming Erlang: software for a concurrent world*. Pragmatic Bookshelf, 2007.

[26] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik, "Scalable distributed stream processing," in *First Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA*, 2003.

[27] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik, "Aurora: a new model and architecture for data stream management," *Internation Journal on Very Large Data Bases (VLDB)*, vol. 12, no. 2, pp. 120–139, 2003.

[28] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik, "The design of the borealis stream processing engine," in *Second Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA*, pp. 277–289, 2005.

[29] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The stanford data stream management system," Technical Report 2004-20, Stanford InfoLab, 2004.

[30] G. Amato, S. Chessa, and C. Vairo, "MaD-WiSe: a distributed stream management system for wireless sensor networks," *Software: Practice and Experience*, vol. 40, pp. 431–451, Apr. 2010.

[31] "Storm - Distributed and fault-tolerant realtime computation." http://storm-project.net/, 2013.

[32] "Apache Samza Project." http://samza.incubator.apache.org/, 2013.

[33] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pp. 170–177, IEEE, 2010.

[34] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pp. 10–10, USENIX Association, 2012.

[35] J. Beutel, O. Kasten, and M. Ringwald, "Btnodes–a distributed platform for sensor nodes," in *Proceedings of the 1st International Conference on Embedded networked sensor systems*, pp. 292–293, ACM, 2003.

[36] P. Dutta, "Epic: An open mote platform for application-driven design." http://www.eecs.berkeley.edu/ prabal/projects/epic/, Oct 2010.

[37] MEMSIC, Inc., "IRIS: Wireless Measurement System." http://www.memsic.com/userfiles/files/Datasheets/WSN/IRIS_Datasheet.pdf.

[38] C. Technology, "Mica2: Wireless Measurement System," 2004.

[39] H. University, "Shimmer getting started guide." http://www.eecs.harvard.edu/ konrad/projects/shimmer/SHIMMER-GettingStartedGuide.html, 2006.

[40] Sun Labs, *Sun SPOT Programmer's Manual: Release v6.0 (Yellow)*, Nov. 2010. Document Revision 2.0.

[41] M. Corp., "Tmote Sky : Low Power Wireless Sensor Module." http://www.eecs.harvard.edu/ konrad/projects/shimmer/references/tmote-sky-datasheet.pdf, 2006.

[42] A. Rowe, R. Mangharam, and R. Rajkumar, "Firefly: A time synchronized real-time sensor networking platform," *Wireless Ad Hoc Networking: Personal-Area, Local-Area, and the Sensory-Area Networks, CRC Press Book Chapter*, 2006.

[43] Libelium Comunicaciones Distribuidas S.L., "Waspmote data sheet." http://web.univ-pau.fr/ cpham/ENSEIGNEMENT/PAU-UPPA/RESA-M2/DOC/waspmote-datasheet_eng.pdf, 2010.

[44] "Wisense – Remote Sensor Monitoring System." http://www.ewisense.com.

[45] G. Werner-Allen, K. Lorincz, M. Ruiz, O. Marcillo, J. Johnson, J. Lees, and M. Welsh, "Deploying a wireless sensor network on an active volcano," *Internet Computing, IEEE*, vol. 10, no. 2, pp. 18–25, 2006.

[46] N. Zhang and A. Pisano, "Harsh environment temperature sensor based on 4h-silicon carbide pn diode," in *Solid-State Sensors, Actuators and Microsystems (TRANSDUCERS EUROSENSORS XXVII), 2013 Transducers Eurosensors XXVII: The 17th International Conference on*, pp. 1016–1019, June 2013.

[47] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, "TinyDB: An acquisitional query processing system for sensor networks," *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 122–173, 2005.

[48] B. Scholz, M. M. Gaber, T. Dawborn, R. Khoury, and E. Tse, "Efficient time triggered query processing in wireless sensor networks," in *ICESS*, vol. 4523 of *Lecture Notes on Computer Science*, pp. 391–402, Springer, 2007.

[49] R. Khoury, T. Dawborn, B. Gafurov, G. Pink, E. Tse, Q. Tse, K. Almi'Ani, M. M. Gaber, U. Röhm, and B. Scholz, "Corona: Energy-efficient multi-query processing in wireless sensor networks," in *DASFAA (2)*, vol. 5982 of *Lecture Notes on Computer Science*, pp. 416–419, Springer, 2010.

[50] A. Pathak and V. K. Prasanna, "Energy-efficient task mapping for data-driven sensor network macroprogramming," *IEEE Trans. Computers*, vol. 59, no. 7, pp. 955–968, 2010.

[51] A. Venuturumilli and A. Minai, "Obtaining robust wireless sensor networks through self-organization of heterogeneous connectivity," in *Unifying Themes in Complex Systems*, pp. 487–494, Springer, 2008.

[52] X. Huang, H. Zhai, and Y. Fang, "Robust cooperative routing protocol in mobile wireless sensor networks," *Wireless Communications, IEEE Transactions on*, vol. 7, no. 12, pp. 5278–5285, 2008.

[53] R. Kumar, V. Tsiatsis, and M. B. Srivastava, "Computation hierarchy for in-network processing," in *Proceedings of the 2nd ACM International Conference on Wireless sensor networks and applications*, pp. 68–77, ACM, 2003.

[54] L. Alazzawi, A. Elkateeb, and A. Ramesh, "Scalability analysis for wireless sensor networks routing protocols," in *Advanced Information Networking and Applications - Workshops, 2008. AINAW 2008. 22nd International Conference on*, pp. 139–144, March 2008.

[55] D. Gay, P. Levis, J. R. von Behren, M. Welsh, E. A. Brewer, and D. E. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA* (R. Cytron and R. Gupta, eds.), pp. 1–11, ACM, 2003.

[56] R. Newton, G. Morrisett, and M. Welsh, "The Regiment macroprogramming system," in *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN)*, pp. 489–498, ACM, 2007.

[57] R. Harms and M. Yamartino, "The economics of the cloud," *Microsoft whitepaper, Microsoft Corporation*, 2010.

[58] Amazon, Inc., "Elastic Compute Cloud (Amazon EC2)." http://aws.amazon.com/ec2/, July 2013.

[59] Microsoft, Inc., "Microsoft Windows Azure." http://www.windowsazure.com/en-us/documentation/, Nov. 2013.

[60] Google, Inc., "Google Compute Engine." https://cloud.google.com/products/compute-engine, Nov. 2013.

[61] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, "Telegraphcq: continuous dataflow processing," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of data*, pp. 668–668, ACM, 2003.

[62] G. Călinescu, H. Karloff, and Y. Rabani, "An improved approximation algorithm for multiway cut," *Journal of Computer and System Sciences*, vol. 60, no. 3, pp. 564–574, 2000.

[63] R. Szewczyk, E. Osterweil, J. Polastre, M. Hamilton, A. Mainwaring, and D. Estrin, "Habitat monitoring with sensor networks," *Communications of ACM*, vol. 47, no. 6, pp. 34–40, 2004.

[64] F. Ingelrest, G. Barrenetxea, G. Schaefer, M. Vetterli, O. Couach, and M. Parlange, "Sensorscope: Application-specific sensor network for environmental monitoring," *ACM Transactions On Sensor Networks (TOSN)*, vol. 6, no. 2, 2010.

[65] T. He, S. Krishnamurthy, L. Luo, T. Yan, L. Gu, R. Stoleru, G. Zhou, Q. Cao, P. Vicaire, J. A. Stankovic, T. F. Abdelzaher, J. Hui, B. Krogh, T. U. E. S. Krishnamurthy, L. Luo, T. Yan, L. Gu, R. Stoleru, G. Zhou, and Q. Cao, "Vigilnet: An integrated sensor network system for energy-efficient surveillance," *ACM Transactions on Sensor Networks (TOSN)*, vol. 2, pp. 1–38, 2006.

[66] E. Cheong and J. Liu, "galsC: A language for event-driven embedded systems," in *Proceedings of the conference on Design, Automation and Test*, vol. 2, pp. 1050–1055, IEEE Computer Society, 2005.

[67] R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming wireless sensor networks using *Kairos*," in *First IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS), Marina del Rey, CA, USA*, vol. 3560 of *Lecture Notes on Computer Science*, pp. 126–140, Springer, 2005.

[68] G. Mainland, G. Morrisett, and M. Welsh, "Flask: Staged functional programming for sensor networks," in *Proceeding of the 13th ACM SIGPLAN International Conference on Functional programming (ICFP), Victoria, BC, Canada*, pp. 335–346, ACM, 2008.

[69] Y. Yao and J. Gehrke, “The Cougar approach to in-network query processing in sensor networks,” *ACM SIGMOD Record*, vol. 31, no. 3, pp. 9–18, 2002.

[70] V. Kakkad, S. Attar, A. E. Santosa, A. Fekete, and B. Scholz, “Curracurrong: a stream programming environment for wireless sensor networks,” *Software: Practice and Experience*, vol. 44, no. 2, pp. 175–199, 2014.

[71] W. Thies, M. Karczmarek, and S. P. Amarasinghe, “Streamit: A language for streaming applications,” in *11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS), Grenoble, France*, vol. 2304 of *Lecture Notes on Computer Science*, pp. 179–196, Springer, 2002.

[72] “Borealis: Distributed Stream Processing Engine.” http://cs.brown.edu/research/borealis/public/#software, 2013.

[73] T. R. G. Green and M. Petre, “When Visual Programs are Harder to Read than Textual Programs,” in *Human-Computer Interaction: Tasks and Organisation, Proceedings of 6th European Conference Cognitive Ergonomics (ECCE)*, 1992.

[74] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[75] R. Smith, “SPOTWorld and the Sun SPOT,” in *Proceedings of the 6th International Conference on Information processing in sensor networks (IPSN)*, pp. 565–566, ACM, 2007.

[76] R. Sedgewick, *Algorithms in C, Part 5: graph algorithms*. Addison-Wesley Professional, 2001.

[77] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, “Tag: A tiny aggregation service for ad-hoc sensor networks,” in *5th Symposium on Operating System Design and Implementation (OSDI), Boston, Massachusetts, USA* (D. E. Culler and P. Druschel, eds.), USENIX Association, 2002.

[78] K. Bachour, E. Baykan, W. Galuba, and A. Salehi, “Citation network partitioning,” tech. rep., École Polytechnique Fédérale de Lausanne, 2005.

[79] H. S. Stone, “Multiprocessor scheduling with the aid of network flow algorithms,” *IEEE Transactions on Software Enggineering*, vol. 3, no. 1, pp. 85–93, 1977.

[80] V. Nikolyenko, “Instruction scheduling with convex optimisation.” Bachelor’s thesis. School of Information Technologies, University of Sydney, Australia, June 2011.

[81] L. R. Ford and D. R. Fulkerson, “Maximal flow through network,” *Canadian Journal of Mathematics*, vol. 8, pp. 399–404, 1956.

[82] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis, “The complexity of multiterminal cuts,” *SIAM Journal on Computing*, vol. 23, no. 4, pp. 864–894, 1994.

[83] G. Calinescu, H. Karloff, and Y. Rabani, “An improved approximation algorithm for multiway cut,” in *Proceedings of the 30th Annual ACM Symposium on the Theory of Computing (SOTC), Dallas, Texas, USA* (J. S. Vitter, ed.), pp. 48–52, ACM, 1998.

[84] D. R. Karger, P. N. Klein, C. Stein, M. Thorup, and N. E. Young, “Rounding algorithms for a geometric embedding of minimum multiway cut.,” in *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (SOTC), Atlanta, Georgia, USA* (J. S. Vitter, L. L. Larmore, and F. T. Leighton, eds.), pp. 668–678, ACM, 1999.

[85] Y.-C. Wu, Q. Chaudhari, and E. Serpedin, “Clock synchronization of wireless sensor networks,” *Signal Processing Magazine, IEEE*, vol. 28, no. 1, pp. 124–138, 2011.

[86] Q. Li and D. Rus, “Global clock synchronization in sensor networks,” *Computers, IEEE Transactions on*, vol. 55, no. 2, pp. 214–226, 2006.

[87] “GLPK (GNU linear programming kit).” http://www.gnu.org/s/glpk/, 2008.

[88] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*. Brooks/Cole, 2003. 2nd Edition.

[89] B. Hayes, “Cloud computing,” *Commun. ACM*, vol. 51, pp. 9–11, July 2008.

[90] G. Garrison, S. Kim, and R. L. Wakefield, "Success factors for deploying cloud computing," *Communications of the ACM*, vol. 55, no. 9, pp. 62–68, 2012.

[91] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI '04*, pp. 137–150, 2004.

[92] J. Dean and S. Ghemawat, "Mapreduce: a flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[93] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–10, 2010.

[94] L. Golab and M. T. Özsu, eds., *Data Stream Management*. Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2010.

[95] P. Angin, B. K. Bhargava, and S. Helal, "A mobile-cloud collaborative traffic lights detector for blind navigation," in *11th MDM* (T. Hara, C. S. Jensen, V. Kumar, S. Madria, and D. Zeinalipour-Yazti, eds.), pp. 396–401, IEEE Computer Society, 2010.

[96] X. Li, H. Zhang, and Y. Zhang, "Deploying mobile computation in cloud service," in *1st CloudCom* (M. G. Jaatun, G. Zhao, and C. Rong, eds.), vol. 5931 of *Lecture Notes on Computer Science*, pp. 301–311, Springer, 2009.

[97] U. Srivastava, K. Munagala, and J. Widom, "Operator placement for in-network stream query processing," in *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 250–258, ACM, 2005.

[98] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Computer Survey*, vol. 36, pp. 1–34, March 2004.

[99] T. D. Le, N. Ahmed, and S. Jha, "Location-free fault repair in hybrid sensor networks," in *1st InterSense* (I. Chlamtac, ed.), vol. 138 of *ACM International Conference Proceeding Series*, p. 23, ACM, 2006.

[100] P. Erdös and A. Rényi, "On the evolution of random graphs," in *Publication of the Mathematical Institute of the Hungarian Academy of Sciences*, pp. 17–61, 1960.

[101] W. Du, J. Deng, Y. S. Han, P. K. Varshney, J. Katz, and A. Khalili, "A pairwise key predistribution scheme for wireless sensor networks," *ACM Trans. Inf. Syst. Secur.*, vol. 8, pp. 228–258, May 2005.

[102] A. Hamlili, "Adaptive schemes for estimating random graph parameters in mobile wireless ad hoc networks' modeling," pp. 1–5, 2010.

[103] R. Serna Oliver and G. Fohler, "Timeliness in wireless sensor networks: Common misconceptions," in *Proceedings of the 9th International Workshop on Real-Time Networks RTN'2010*, (Brussels, Belgium), July 2010.

[104] Y. Wang, M. Vuran, and S. Goddard, "Cross-layer analysis of the end-to-end delay distribution in wireless sensor networks," *Networking, IEEE/ACM Transactions on*, vol. 20, pp. 305–318, feb. 2012.

[105] M. Neuts, J. Guo, M. Zukerman, and H. L. Vu, "The waiting time distribution for a tdma model with a finite buffer and state-dependent service," *Communications, IEEE Transactions on*, vol. 53, pp. 1522 – 1533, sept. 2005.

[106] T. Abdelzaher, S. Prabh, and R. Kiran, "On real-time capacity limits of multihop wireless sensor networks," in *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*, pp. 359 – 370, 2004.

[107] S.-N. Yeung and J. Lehoczky, "End-to-end delay analysis for real-time networks," in *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, pp. 299 – 309, dec. 2001.

[108] J. B. Dennis, "First version of a data flow procedure language," in *Symposium on Programming '74* (B. Robinet, ed.), vol. 19 of *Lecture Notes on Computer Science*, pp. 362–376, Springer, 1974.

[109] Y. Zhou and E. A. Lee, "Causality interfaces for actor networks," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 29:1–29:35, May 2008.

[110] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, pp. 1235–1245, Sept 1987.

[111] A. Jain, R. Bolle, and S. Pankanti, *Introduction to biometrics*. Springer, 1996.

[112] "Biometrics." http://en.wikipedia.org/wiki/Biometrics, 2014.

[113] A. Biem, B. Elmegreen, O. Verscheure, D. Turaga, H. Andrade, and T. Cornwell, "A streaming approach to radio astronomy imaging," in *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, pp. 1654–1657, IEEE, 2010.

[114] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," *SIGARCH Comput. Archit. News*, vol. 3, no. 4, pp. 126–132, 1974.

[115] J. Von Neumann, "First draft of a report on the edvac," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[116] E. A. Lee, "A denotational semantics for dataflow with firing," in *Technical Memorandum UCB/ERL M97/3, Electronics Research Laboratory, Berkeley*, (University of California, Berkeley), 1997.

[117] J. R. McGraw, "The val language: Description and analysis," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 1, pp. 44–82, 1982.

[118] I. Buck, *Brook: A streaming programming language*. Stanford University, 2001.

[119] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL: Revised OpenCL 1*. Newnes, 2012.

[120] C. Nvidia, "Compute unified device architecture programming guide," 2007.

[121] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, *et al.*, "Ibm streams processing language: Analyzing big data in motion," *IBM Journal of Research and Development*, vol. 57, no. 3/4, pp. 7–1, 2013.

[122] R. A. Finkel and S. N. Kamin, *Advanced programming language design*. Addison-Wesley Reading, 1996.

[123] J. R. Gurd, C. C. Kirkham, and I. Watson, “The manchester prototype dataflow computer,” *Communications of the ACM*, vol. 28, no. 1, pp. 34–52, 1985.

[124] B. Houssais, “The synchronous programming language signal: A tutorial,” *IRISA, April*, 2002.

[125] A. Kamil, “Fleet Programming Paradigms,” *Computer Science Division, University of California, Berkeley*, 2008.

[126] http://pic.dhe.ibm.com/infocenter/streams/v3r0/index.jsp, 2012.

[127] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, “Niagaracq: A scalable continuous query system for internet databases,” in *ACM SIGMOD Record*, vol. 29, pp. 379–390, ACM, 2000.

[128] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, “Stream: The stanford data stream management system,” *Book chapter*, 2004.

[129] “Fjord.” https://github.com/mozilla/fjord/blob/master/docs/api.rst, 2014.

[130] U. Cetintemel, “The aurora and medusa projects,” *Data Engineering*, vol. 51, no. 3, 2003.

[131] Y. Yu and V. K. Prasanna, “Energy-balanced task allocation for collaborative processing in wireless sensor networks,” *Mobile Networks and Applications (MONET)*, vol. 10, no. 1-2, pp. 115–131, 2005.

[132] “IBM Infosphere Stream.” http://www-01.ibm.com/software/data/infosphere/streams/.

[133] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, “Spade: the system s declarative stream processing engine,” in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of data*, pp. 1123–1134, ACM, 2008.

[134] O. Diallo, J. J. Rodrigues, and M. Sene, “Real-time data management on wireless sensor networks: A survey,” *Journal of Network and Computer Applications*, vol. 35, no. 3, pp. 1013–1021, 2012.

[135] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *SIGMOD Rec.*, vol. 34, pp. 42–47, Dec. 2005.

[136] K.-Y. Lam and T.-W. Kuo, *Real-time database systems: architecture and techniques*. Springer, 2001.

[137] K. Ramamritham, "Real-time databases," *Distributed and parallel databases*, vol. 1, no. 2, pp. 199–226, 1993.

[138] A. Burchard, J. Liebeherr, and S. Patek, "A min-plus calculus for end-to-end statistical service guarantees," *Information Theory, IEEE Transactions on*, vol. 52, pp. 4105–4114, sept. 2006.

[139] J. B. Schmitt, F. A. Zdarsky, and L. Thiele, "A comprehensive worst-case calculus for wireless sensor networks with in-network processing," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, RTSS '07, (Washington, DC, USA), pp. 193–202, IEEE Computer Society, 2007.

[140] D. C. Luckham, *The power of events*, vol. 204. Addison-Wesley Reading, 2002.