
Liquid Stream Processing on the Web: a JavaScript Framework

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Masiar Babazadeh

under the supervision of
Prof. Dr. Cesare Pautasso

October 2017

Dissertation Committee

Prof. Dr. Marc Langheinrich Università della Svizzera Italiana, Lugano, Switzerland
Prof. Dr. Robert Soulé Università della Svizzera Italiana, Lugano, Switzerland
Prof. Dr. Gustavo Alonso Eidgenössische Technische Hochschule Zürich, Switzerland
Prof. Dr. Tommi Mikkonen University of Helsinki, Finland

Dissertation accepted on 25 October 2017

Research Advisor

Prof. Dr. Cesare Pautasso

PhD Program Directors

Prof. Dr. Igor Pivkin, Prof. Dr. Stefan Wolf

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Masiar Babazadeh
Lugano, 25 October 2017

To the ones I love the most

*Nullius addictus iurare in verba
magistri,
quo me cumque rapit tempestas,
deferor hospes.*

Horace

Abstract

The Web is rapidly becoming a mature platform to host distributed applications. Pervasive computing application running on the Web are now common in the era of the Web of Things, which has made it increasingly simple to integrate sensors and microcontrollers in our everyday life. Such devices are of great interest to Makers with basic Web development skills. With them, Makers are able to build small smart stream processing applications with sensors and actuators without spending a fortune and without knowing much about the technologies they use. Thanks to ongoing Web technology trends enabling real-time peer-to-peer communication between Web-enabled devices, Web browsers and server-side JavaScript runtimes, developers are able to implement pervasive Web applications using a single programming language. These can take advantage of direct and continuous communication channels going beyond what was possible in the early stages of the Web to push data in real-time.

Despite these recent advances, building stream processing applications on the Web of Things remains a challenging task. On the one hand, Web-enabled devices of different nature still have to communicate with different protocols. On the other hand, dealing with a dynamic, heterogeneous, and volatile environment like the Web requires developers to face issues like disconnections, unpredictable workload fluctuations, and device overload.

To help developers deal with such issues, in this dissertation we present the Web Liquid Streams (WLS) framework, a novel streaming framework for JavaScript. Developers implement streaming operators written in JavaScript and may interactively and dynamically define a streaming topology. The framework takes care of deploying the user-defined operators on the available devices and connecting them using the appropriate data channel, removing the burden of dealing with different deployment environments from the developers. Changes in the semantic of the application and in its execution environment may be applied at runtime without stopping the stream flow.

Like a liquid adapts its shape to the one of its container, the Web Liquid Streams framework makes streaming topologies flow across multiple heteroge-

neous devices, enabling dynamic operator migration without disrupting the data flow. By constantly monitoring the execution of the topology with a hierarchical controller infrastructure, WLS takes care of parallelising the operator execution across multiple devices in case of bottlenecks and of recovering the execution of the streaming topology in case one or more devices disconnect, by restarting lost operators on other available devices.

Acknowledgements

The Ph.D. grind has been a long and illuminating journey with many ups and few downs.

First and foremost I would like to thank Prof. Cesare Pautasso for having spent a good amount of the past five years working with me. Our success and failures had taught me a lot – on an academic and personal level. I grew up a lot as a researcher during this journey thanks to his precise and sharp observations, and his dedication in our work increased its quality, teaching me how good research is done.

I'd like to thank Prof. Marc Langheinrich and Prof. Robert Soulé from Lugano, Prof. Gustavo Alonso from Zürich, and Prof. Tommi Mikkonen from Helsinki for being part of this journey of mine, and for their strong feedback and support. Their invaluable comments had helped the development of this dissertation.

This amazing journey wouldn't have been the same without my first travel companion, Vassilis, and all the old and new team: Achille, Ana, Daniele, Marcin, Saeed, and Vincenzo, and all the amazing people I've met at USI. Their support through these Ph.D. years has been amazing. Last but not least, I'd like to give big credits to Andrea with whom I shared great trips and amazing moments, while squatting his place every morning for a cup of coffee.

The biggest shoutout of all goes to my family and their infinite support, and all my friends that walked with me through this big journey of mine. Finally, I'd like to credit Lisa, the most amazing person I've ever met. Her help and support throughout my bachelor, master, and Ph.D. years made me grow in many ways, making me a better person.

Contents

| | |
|--|------------|
| Contents | xi |
| List of Figures | xv |
| List of Tables | xix |
| | |
| I Prologue | 1 |
| | |
| 1 Introduction | 3 |
| 1.1 The Web of Things | 4 |
| 1.2 Data Streams and the Web | 4 |
| 1.3 Liquid Software | 5 |
| 1.4 Motivation | 6 |
| 1.4.1 Thesis Statement | 8 |
| 1.5 Contributions | 8 |
| 1.6 Outline | 9 |
| | |
| 2 Related Work | 11 |
| 2.1 The Stream Connector | 11 |
| 2.1.1 Defining the Stream Software Connector | 11 |
| 2.1.2 Stream Processing Systems and Languages Surveyed | 13 |
| 2.1.3 Survey Methodology | 17 |
| 2.1.4 Design-Time | 18 |
| 2.1.5 Run-Time | 28 |
| 2.1.6 Outlook | 38 |
| 2.1.7 Stream Optimisations | 38 |
| 2.2 Streaming on the Web of Things | 39 |
| 2.3 Wireless Sensor Networks | 40 |
| 2.4 Liquid Software Architecture | 41 |

| | | |
|-----------|--|-----------|
| 2.5 | Mobile/Cloud | 42 |
| II | Web Liquid Streams | 45 |
| 3 | The Web Liquid Streams Framework | 47 |
| 3.1 | Introduction | 47 |
| 3.1.1 | The Liquid Software Metaphor in WLS | 49 |
| 3.2 | System Model | 52 |
| 3.3 | Developing Operator Scripts | 53 |
| 3.4 | Deploying a Streaming Topology | 62 |
| 3.4.1 | Command Line Interface | 62 |
| 3.4.2 | Topology Description File | 65 |
| 3.5 | RESTful API | 68 |
| 3.5.1 | Resources | 68 |
| 3.5.2 | Uniform Interface | 70 |
| 3.5.3 | Representations | 73 |
| 3.6 | Graphical User Interface | 75 |
| 3.7 | Use Cases | 76 |
| 3.7.1 | New Peer Joins the Network | 76 |
| 3.7.2 | Setting up a Topology with a Topology Description File | 76 |
| 3.7.3 | Using Web Browsers to run Operators | 76 |
| 3.7.4 | Perform a new Binding | 77 |
| 3.7.5 | Load Balancing through the REST API | 77 |
| 4 | The Web Liquid Streams Runtime | 79 |
| 4.1 | The WLS Communication Layers | 80 |
| 4.1.1 | Command Layer | 80 |
| 4.1.2 | Stream Layer | 84 |
| 4.2 | Peer Infrastructure | 85 |
| 4.2.1 | Web Server Peer | 85 |
| 4.2.2 | Minified Web Server Peer | 88 |
| 4.2.3 | Web Browser Peer | 90 |
| 4.3 | Operator Infrastructure | 91 |
| 4.3.1 | Web Server Operator Pool | 92 |
| 4.3.2 | Web Browser Operator Pool | 94 |
| 4.3.3 | Web Server Worker Pool | 98 |
| 4.3.4 | Web Browser Worker Pool | 99 |
| 4.4 | Topology Creation and Dynamic Evolution | 100 |

| | | |
|------------|---|------------|
| 4.5 | Stateful Operators | 101 |
| 4.5.1 | Overview | 101 |
| 4.5.2 | Redis | 102 |
| 4.5.3 | Implementation | 104 |
| 4.6 | Summary | 105 |
| 5 | The Control Infrastructure | 107 |
| 5.1 | Controller Use Cases | 107 |
| 5.1.1 | Operator Migration | 107 |
| 5.1.2 | Peer Failure | 108 |
| 5.1.3 | Root Peer Failure | 108 |
| 5.1.4 | Lack of Resources for Parallelisation | 108 |
| 5.1.5 | Lack of Peers for Deployment | 108 |
| 5.2 | The Controller Tasks and Constraints | 109 |
| 5.2.1 | Automatic Deployment | 110 |
| 5.2.2 | Load Balancing | 110 |
| 5.2.3 | Operator Migration | 111 |
| 5.2.4 | Disconnection Handling | 112 |
| 5.3 | Implementation | 113 |
| 5.3.1 | Global Controller Implementation | 113 |
| 5.3.2 | Web Server Local Controller Implementation | 113 |
| 5.3.3 | Web Browser Local Controller Implementation | 116 |
| 5.3.4 | Ranking Function | 118 |
| III | Evaluation | 121 |
| 6 | Application Case Studies | 123 |
| 6.1 | Study Week in Informatics | 123 |
| 6.1.1 | Lessons learned | 124 |
| 6.2 | Inforte Seminar on Software Technologies and Development for Multi-Device Environments | 124 |
| 6.2.1 | Lessons learned | 126 |
| 6.3 | WLS as a Mashup Tool | 126 |
| 6.3.1 | Lessons learned | 129 |
| 6.4 | Software Atelier 3: The Web – Home Automation System Project | 129 |
| 6.4.1 | Lessons learned | 131 |
| 6.5 | Experimentelle Evaluation des Web Liquid Streams-Framework | 133 |
| 6.5.1 | Lessons Learned | 138 |

| | | |
|-----------|---|------------|
| 7 | Performance Evaluation | 139 |
| 7.1 | Overview | 139 |
| 7.2 | Metrics | 141 |
| 7.3 | Operator Migration and Disconnection Recovery | 142 |
| 7.4 | Elastic Parallelisation Experiment | 145 |
| 7.5 | Global Controller Algorithm | 149 |
| 7.6 | Web Browser Local Controller and Fine-tuning | 151 |
| 7.7 | 1 Failure | 156 |
| 7.8 | N-Failures | 157 |
| 7.9 | Summary | 160 |
| | | |
| IV | Epilogue | 165 |
| | | |
| 8 | Conclusion | 167 |
| 8.1 | Future Work | 170 |
| | | |
| | Bibliography | 173 |

Figures

| | | |
|------|--|-----|
| 3.1 | Example of a distributed streaming topology running on different peers. Peer 1 and peer 2 in this particular case host both the computation of operator 2. | 49 |
| 3.2 | The Web Liquid Streams features in the liquid software feature model introduced in [GPM ⁺ 17]. | 50 |
| 3.3 | WLS in the maturity model for liquid Web software applications [GP17]. | 52 |
| 3.4 | Logical view of the topology we illustrate in this Chapter. | 59 |
| 3.5 | Hypermedia navigation map, showing the resources that can be discovered from each GET request. | 69 |
| 3.6 | Web-based Graphical User Interface. | 75 |
| 4.1 | WLS Runtime and the three different deployment implementations. | 80 |
| 4.2 | RPC and IPC Interactions among distributed and heterogeneous peers. | 82 |
| 4.3 | Closeup of the RPC interaction among two RPC modules, one acting as a client while the other acting as a server. | 83 |
| 4.4 | Logical view of the command layer of the Web server peer. Streaming channels are not shown. | 89 |
| 4.5 | Minified Web server peer infrastructure. | 90 |
| 4.6 | Logical view of a Web browser peer. | 92 |
| 4.7 | Logical view of the Web browser operator | 98 |
| 4.8 | The worker communication infrastructure for server-to-server and server-to-browser communication. | 99 |
| 4.9 | Step-by-step setup of a topology. | 100 |
| 4.10 | Redis communication abstraction with WLS | 104 |
| 4.11 | Interaction with the Redis module. | 105 |
| 5.1 | Step-by-step migration of an operator. | 111 |
| 5.2 | Visual representation of the Web server local controller cycle. | 114 |
| 5.3 | Local Web server controller behaviour. | 115 |

| | | |
|------|--|-----|
| 6.1 | Topology implemented in the study week in informatics. | 124 |
| 6.2 | Topology implemented in the workshop. | 125 |
| 6.3 | The mashup topology. | 128 |
| 6.4 | Physical deployment of the operators and data flow in the mashup topology [GBP16]. | 128 |
| 6.5 | Screenshot of the mashup running on a Web browser [GBP16]. | 129 |
| 6.6 | Topology implemented during the Software Atelier 3 project | 130 |
| 6.7 | Tessels and microphone modules built by the students. | 131 |
| 6.8 | Screenshot of the application running on a Web browser. | 132 |
| 6.9 | Topology for the Karlsruhe example | 133 |
| 6.10 | Screenshot of the consumer Web page on the running application [Fus16]. | 137 |
| 7.1 | DES Encryption topology. | 142 |
| 7.2 | Operator Migration and Recovery Impact on Throughput. Vertical lines indicate Low Battery and Disconnection events. | 144 |
| 7.3 | Throughput distribution during the operator migration and recovery experiments with low battery and disconnection. | 145 |
| 7.4 | Number of workers throughout the three scenarios. | 146 |
| 7.5 | Parallelisation of the execution as the workload mutates every 5000 messages (slow). | 147 |
| 7.6 | Parallelisation of the execution as the workload mutates every 500 messages (fast). | 148 |
| 7.7 | Throughputs for the two different workloads in the two experiments. | 149 |
| 7.8 | Liquid deployment: comparison of random vs. ranked resource allocations and their median end-to-end latency. | 150 |
| 7.9 | Face detection and decoration topology employed for the experiment. | 151 |
| 7.10 | Message latency and queue size distributions per peer running on different controller configuration with 6 messages per second. | 153 |
| 7.11 | Message latency and queue size distributions per peer running on different controller configuration with 10 messages per second. | 154 |
| 7.12 | Message latency and queue size distributions per peer running on different controller configuration with 13 messages per second. | 155 |
| 7.13 | Throughput of the topologies in the three experiments with the four controller configurations. | 156 |
| 7.14 | Oscillation in the number of workers throughout the experiment with only four tablets as filter deployment. | 157 |
| 7.15 | End-to-end latency for the four tablets. | 158 |

| | |
|--|-----|
| 7.16 Queue size shown per peer. | 159 |
| 7.17 Oscillation in the number of workers throughout the experiment. | 160 |
| 7.18 End-to-end latency shown per peer. | 161 |
| 7.19 Queue size shown per peer. | 162 |

Tables

| | | |
|-----|---|-----|
| 2.1 | Summary of the design decisions (+) over the stream connector design space. | 36 |
| 3.1 | List of WLS command line interface commands. | 64 |
| 4.1 | Data channels for different deployments. | 85 |
| 4.2 | Messages scopes within the RPC communication. | 93 |
| 7.1 | Machines used during the WLS evaluation. | 140 |
| 7.2 | Controller configuration parameters | 152 |

Part I
Prologue

Chapter 1

Introduction

The Web is becoming a mature platform to host distributed applications. Thanks to standard protocols like WebSockets [FM11] and WebRTC [BBJN12] (Web Real-Time Communication) developers are able to connect machines of different nature, all running a Web browser, and share their resources for distributed computations. In the era of the Web of Things [GT16], pervasive computing applications [DR07, PdABL⁺13] are one example of a class of distributed application that recently started to use the Web as a platform. Microcontrollers and single-board PCs have become interesting platforms to develop such applications, offering powerful multicore CPUs¹ while remaining cheap and small in size. These development platforms have recently become of interest by Makers [And12], people that often do not have a background in computer science, but enjoy tinkering and hacking, and in general like to follow the principles of the Do It Yourself (DIY) [McK98] culture.

Makers that want to build a small-sized pervasive Web application (for example, to make their homes a smart environment [DW16], or just simplify their everyday life through ambient assisted living [SFBS12]) have to deal with different development deployment environments (microcontrollers or single-board PCs, home desktop PCs, home servers, smartphones, tablets, ...) which often means different programming languages and communication protocols. Besides, failures in the execution of the streaming topologies and bottlenecks during the execution have to be dealt by the developer of the application, with ad hoc solutions.

In this dissertation we present Web Liquid Streams (WLS), a streaming framework that lets developers build complex distributed streaming Web applications by coding the functionalities of the streaming operators and describing the topol-

¹<https://www.raspberrypi.org/magpi/raspberrypi-3-specs-benchmarks/>

ogy in plain JavaScript, without worrying about the communication channels, the code distribution, node failures, and bottlenecks in the execution of streaming topologies.

1.1 The Web of Things

As more and more sensors and smart devices are getting connected to the Internet [TM17, GBMP13], an interest has grown in exploring the use of the World Wide Web as a platform for such devices [GTW10]. The Web of Things (WoT) is a set of architectural styles and programming patterns that help integrating these smart devices and, more in general, real-world objects into the World Wide Web. Frameworks (i.e., EVERTHING [Gui11]), protocols (i.e., CoAP [Kov13]) and best practices [GTMW11] have been proposed to bridge the gap between the real-world and the Web.

The Web of Things offers a playground for Makers and developers that want to build their own systems using single-board PCs, sensors, and actuators. Developers can make their smart objects exchange data by using the Web and its well-known standards. While using the Web as the common platform to send and receive data may be trivial, developing an application that deals with smart objects of different nature is more difficult. Dealing with small and faulty devices means manually restarting the application or updating code on different devices to solve bugs.

In this dissertation we show how we are able to offer to the developers a framework to build streaming applications for the Web of Things by using JavaScript and exploiting Node.js and the Web browsers as our execution environment. Our framework is able to keep track of all the connected devices, autonomically orchestrating the execution on streaming operators on the available devices, and automatically dealing with faults by migrating the execution on other devices with the same capabilities.

1.2 Data Streams and the Web

During the early stages of the Web, Web browsers could only interact with a Web server through synchronous request-response interactions [FTE⁺17]. By downloading HTML code, users could navigate through a website by following links and loading new pages: the logic and the model of a website were stored in the server, while the browser acted as a view. More recently, rich Web applica-

tions [CGM14] allowed the execution of parts of the logic on the Web browser by means of JavaScript, enhancing the user experience. Users could interact with the logic of the Web application on the webpage, modifying its model in the server. With the advent of Web technologies such as the Comet [CM08] Web application model or WebSockets, Web applications became real-time by allowing Web servers update Web browsers without the Web browsers explicitly requesting it. The recently proposed WebRTC API enhances the real time data exchange by enabling direct browser-to-browser streams. WebRTC is an API born to support browser-to-browser applications such as voice calls, video conferencing but also peer-to-peer chat rooms and file sharing natively [RCTSR12, VWS13, VJWS13, DJL15], or more recently sensor data [APC15]. WebRTC allowed exchanging parts of the logic or the model of a Web application to other devices and users without necessarily passing through the Web server. By pushing the logic and the model of a Web application directly on Web browsers, we obtain peer-to-peer Web applications in which the server only acts as discovery for the Web browsers, where the model and the logic is stored.

This Web architecture evolution suggest a paradigm shift from the client-server request-response pattern to a more decentralised peer-to-peer stream of data, where Web server and Web browsers exchange data continuously. While opening a socket and streaming data has become easier, building complex streaming applications that run on Web browsers without necessarily running through a Web server still needs effort. The Web is a volatile environment in which Web browsers join and leave websites in a non-predictable way, thus a robust streaming application should be able to deal with a dynamic and unpredictable environment.

In this dissertation we discuss how, by taking full advantage of novel Web technologies, the use of JavaScript, and the availability of a Web browser in almost any Web-enabled device, we are able to build Web-based streaming applications that run on Web browsers and Web servers, and are able to function in a dynamic environment by migrating and restarting streaming operators on available devices.

1.3 Liquid Software

The liquid software metaphor is used to describe applications that can operate seamlessly across multiple devices owned by one or more users [TMS14, GPM⁺17, BP11]. Such applications can take advantage of computing power, storage, and communication resources on all available devices owned by the

user. Like a liquid can flow from one container to another, liquid software applications can dynamically migrate from one device to another following the user's needs and hiding the effort of dealing with multiple devices of different nature [GP16a, MSP15]. Cloud-based systems such as Apple's Continuity², or Samsung Flow³ already offer a similar approach, however they are tied to devices supporting a similar native ecosystem locking users to the same vendor. While paving the way for automatically synchronised multi-device experience, these system do not yet provide a seamless transition across heterogeneous devices.

The liquid software approach comes in handy when dealing with Web-enabled devices, which are volatile by nature. Users join and leave Web pages, Web-enabled sensors can be turned on and off; by adopting the liquid software concept, developers can "pour" the application code from one device to another as users change environment without disrupting the underlying application runtime.

In this dissertation we make use of the liquid software metaphor to describe the behaviour of a streaming framework that lets developers interoperate with devices of different nature they own by offering a common development interface and a control system that is able to autonomously deal with deployment, parallelisation, and migration of streaming operators, and disconnection recovery. We believe there is great potential in this model as nowadays almost every hardware is able to run a Web browser, thus it can be part of a peer-to-peer system where Web browsers and Web servers are interconnected and share resources to run distributed stream processing applications.

1.4 Motivation

Nowadays the number of Web-enabled device per user is increasing rapidly. Most of these devices own enough computational power to run a Web browser or a Web server. Small single-board computers and microcontrollers have become popular and their number is rapidly increasing [DGHN16]. These devices are used by developers to implement sensor-based applications, nonetheless while it is simple to implement small toy-examples it becomes nontrivial to implement big applications running across multiple heterogeneous devices. Buying off-the-shelf solutions solves implementation issues, but lacks flexibility in terms of programmable

²<https://www.apple.com/macOS/continuity/>

³<http://www.samsung.com/global/galaxy/apps/samsung-flow/>

interfaces (i.e., *Mother by sen.se*⁴, *WallyHome*⁵). Microcontrollers may also lack resources to handle computations [IJHS14], forcing users to buy new hardware if the computations require more effort or to run it on the Cloud [GBMP13]. Running applications that deal with private data, such as sensor data gathered at home, may pose a threat in the privacy of the users [Lan01, LCBRO2, Pea09], which may not want to send their personal data to a server located in the Cloud, but would prefer executing the application on the devices they own.

Current systems also tend to provide solutions for a restricted subset of appliance categories. Thus, if customers want to control multiple appliance types, they need to integrate systems from multiple companies and deal with multiple systems in their houses, or create custom solutions and deal with the lack of interoperability among most of the available sensors and devices [BSM⁺06, TCE⁺10]. Building ad hoc solutions to deal with sensors and microcontrollers may be difficult even for more seasoned developers, which should deal with different communication protocols for devices of different nature, and deal with a dynamic heterogeneous environment.

How would it be possible to exploit the users' hardware in a peer-to-peer fashion to run distributed streaming applications making use of sensors data without the need of installing additional software? Can we make this approach completely dynamic, autonomic, and self-healing, once the stream processing topology has been started?

The framework we propose lets developers build streaming applications on the Web, offering a framework that can be run on most of the heterogeneous Web-enabled devices. It is liquid because streaming Web operators can be migrated around the available devices seamlessly. Developers can process their private sensor data across a personal Cloud of devices they own, without the need to deploy parts of the execution on the public Cloud. By using Web browsers and, more in general, users' devices, we target low-bandwidth applications in which small data packets are sent through home or office connections and processed with relatively low-end processors. Our aim is to make developers' life simpler by dealing with disconnections, autonomous operator deployment, workload fluctuations, as well as offering a flexible framework to build dynamic streaming topologies on the Web.

⁴<https://sen.se/store/mother>

⁵<https://www.wallyhome.com>

1.4.1 Thesis Statement

The goal of this research is to provide developers and Makers with a framework to build distributed streaming topologies that are able to run on the Web, and are able to self heal failures and deal with bottlenecks during the execution. Our research covers the areas of the Web of Things, Web engineering, data streaming, software architecture, and autonomic computing, with the aim of providing a robust framework to build peer-to-peer Web-based streaming topologies.

- *Thanks to emerging Web technologies, it is possible to abstract the complexity of the hardware of Web-enabled devices and offer a JavaScript data stream processing framework with autonomic failure recovery and distributed execution across heterogeneous hardware.*

The Web can be used as a platform to deploy and run streaming applications. Thanks to the newly available technologies, Web browsers and Web servers can interchangeably use the same code to run parts of a streaming topology. Developers may use their own Web-enabled hardware to run personal streaming topologies without the need of Cloud services. An autonomous control system integrated in the liquid streaming framework is able to deploy streaming operators on the available machines, parallelise the execution of computationally intensive streaming operators, migrate the execution on different machines, and recover them from nonpermanent disconnections and failures.

1.5 Contributions

This dissertation gives the following contributions:

- A novel Web-based data stream processing framework that offers developers the possibility to build streaming applications on the Web using JavaScript. Developers can use their own devices and sensors to build streaming topologies with the help of the WLS development API, which offers stateful operators and a Web browser-based HTML5 actuator display API.
- A runtime that, through the use of novel Web technologies such as Node.js, WebSockets, and WebRTC, is able to exploit single-board PCs and micro-controllers, Web servers, and Web browsers to offer a homogeneous experience in a heterogeneous environment. The different environments and different programming styles of Web browsers and Web servers are abstracted by WLS whose runtime takes streaming operators written by the

developers, and seamlessly runs them in either environment without requiring to write ad hoc code. Differences in communication protocols are dealt with by the runtime and abstracted for the developers.

- The framework presents itself as one of the earliest streaming frameworks in the field of liquid software architecture. By implementing the liquid software paradigm in our framework we are able to offer a seamless experience to developers by means of the runtime that exploits connected devices sequentially or simultaneously to run streaming operators that roam from device to device as needed.
- A dynamic controller infrastructure able to autonomically deploy streaming operators and make decisions to solve runtime bottlenecks by parallelising the execution locally, or distributing the execution of the bottleneck operator on more than one device. The controller is also able to migrate the execution of streaming operators and recover from nonpermanent disconnections by restarting operators on the available devices.

1.6 Outline

The dissertation is structured as follows:

In Chapter 2 we will introduce the related work to the dissertation. The related work covers the fields of stream processing languages and frameworks, Web of Things, wireless sensor networks, liquid software, mobile/cloud computing, and streaming optimisations. We take inspiration from some of the stream processing frameworks presented in the literature and the Web of Things. The liquid software paradigm will help us define the procedures to follow to implement a liquid application that can be spread across multiple Web-enabled devices.

Chapters 3 and 4 will introduce the programmer API, RESTful API, and the runtime of the framework. Through realistic examples and the proposed API, we will give an idea of the capabilities of the framework, showing how developers can write operators. We will show how the runtime works, and how we are able to deal with different platforms and different communication channels under the hood. What we give to the developer is a communication abstraction, hiding all the complexity of dealing with different communication protocols to build a reliable stream channel. We will also describe how operators and peers work, showing how we achieve such flexibility at operator, peer, and topology level in the framework.

In Chapter 5 we will show how our implementation of the controller gives to WLS self healing capabilities over nonpermanent failures, and helps dealing with bottlenecks by parallelising operator execution on more than one device. The controller is able to start a migration procedure upon graceful shutdown, keeping the topology running when peers disconnect. We will show the differences in the two implementations of the controller (Web server, Web browser) and show the algorithms implemented.

Chapter 6 will show the application case studies that have been proposed using Web Liquid Streams. We proposed the framework to high-school and middle-school students, as well as to a workshop in the Inforte Summer School held in Tampere, Finland. A project at the University of Applied Sciences in Karlsruhe evaluated WLS by building a browser-to-browser streaming application based on traffic information. We will also show how WLS can be used as a mashup tool, illustrating our submission in the Rapid Mashup Challenge at ICWE 2015. Finally, we will show the results of a project in which USI students used WLS to build a noise and light monitor for the USI open space.

In Chapter 7 we will evaluate the framework, showing how the controller is able to pick the best and most appropriate peers for topology deployment among a set of heterogeneous peers, and how it deals with failures and battery shortages. The evaluation will also show how the controller is able to parallelise the execution of operators on a single peer (by increasing the number of workers) and distributing it on more peers (creating copies of the bottleneck operator). We will also evaluate the Web browser controller by stressing the topology, and show how the two controllers deal with permanent failures, and what happens when not enough peers are available for operator deployment at runtime.

Finally, in Chapter 8 we will wrap up the dissertation, giving an overview on the work done and perspectives on future research.

Chapter 2

Related Work

In the first part of this Chapter we analyse the stream software connector, presenting languages and framework of the past seventeen years that were researched and used to build data streaming applications. The Chapter then covers related work in the area of streaming optimisations, the Web of Things, wireless sensor networks, liquid software, and mobile/cloud computing.

2.1 The Stream Connector

The stream software connector offers a way to connect components in a software system enabling a continuous data feed from one endpoint to another. These connectors usually come in handy when dealing with real time applications, where connectors like remote procedure call (RPC [Sun88]) offering request-response interaction pattern do not fit the needs of such applications.

In this Chapter we survey the recent stream processing frameworks and domain specific languages that feature streaming support to illustrate the state-of-the-art design. The classification of the presented systems and languages is based on a taxonomy extracted from the features they offer (i.e., deployment configurations, load balancing, runtime flexibility, etc.). The result identifies the gaps in the design space and points at future research directions in the area of distributed stream processing.

2.1.1 Defining the Stream Software Connector

The concept of "stream" has recently become popular with the emergence of Web application streaming media on home computers [CVHDVF09]. Data is continuously generated on one endpoint and pushed towards receiver machines, which

collect the packets and feed the streamed content to the end users. The notion of "stream" in computer science, though, has a much older background and a broader scope.

The evolution of the stream software connector is a result of the increasingly large amount of data that organisations have to store, organise, and analyse [AGT14]. This trend forced developers to work on ways to process large-scale data aggregation/summarisation, without the ability to store it. The widespread use of distributed systems to process large-scale data and, more recently, real-time data, has encouraged the development of faster and better integrated data analysis, resulting in the rise of stream processing technologies. Nowadays, streaming applications are used in banking, financial sector, and pervasive computing (Web of Things [GTMW11]) as well as in increasingly popular social media, like Twitter ¹.

Stream processing systems are used to build relatively complex topologies formed by operators and streaming channels. An operator is the basic execution unit that executes a function over a stream of data. We classify operators in three categories: producers, filters, and consumers. Producers are found at the beginning of a topology and start the data stream. Packets pass then through filters, that upon receiving data, process it, and forward the result of the execution downstream. A topology may be composed by many concatenated filters. At the end of the topology, consumers receive the computed data and, for example, store it in a database or visualise it to the users.

Given that different systems use different notations and vocabulary to describe their primitives, we decided to stick with a common terminology, which will be used throughout the rest of this thesis. With the term "system", we address the streaming framework and/or programming language used to build a streaming topology. The term "host" defines the hardware where (part of) the streaming system is run. With the term "operator" we define the logical component of a topology. The term "worker" instead indicates the thread or process that processes the stream inside the operators. Each operator has at least one worker running, but can have more to parallelise the work if the host allows it. For scalability and reliability multiple workers running on a single logical operator can run on multiple hosts. Operators are logically connected by a data stream, whose elements are called "stream elements" or "packets".

¹<https://dev.twitter.com/streaming/firehose>

2.1.2 Stream Processing Systems and Languages Surveyed

We collected a representative sample of systems from both the software architecture and the database area in order to attempt to provide a perspective on the design space. We not only are interested in describing the design space of the stream connector, but we want to take a perspective on the evolution of the stream software connector.

First Generation Systems (2001-2004)

- InfoPipes [BHK⁺02] is a multimedia streaming system developed in 2001 and mainly focused on audio and video (media) streaming applications. InfoPipes topologies are built using a set of pre-defined components such as sources, sinks, buffers, and filters. The system also offers a basic control interface that allows dynamic monitoring and control of the topology and of the information flowing through it. Thanks to special connection interfaces, topologies can be connected at runtime, forming longer and broader topologies (that the authors call "InfoPipelines").
- StreamIt [TKA02] is a programming language and a compilation infrastructure developed in 2002. Operators in StreamIt come in four different flavours: Filter, Pipeline, SplitJoin, and FeedbackLoop. Developers can form a streaming topology by making use of such operators and binding them at code level, and then feed the code to the compilation infrastructure to run the streaming application.
- CQL [ABW06] (*Continuous Query Language*) is a programming language and execution engine developed in 2003. CQL is a declarative language that extends SQL [CB74] with the capability of querying windows over possibly infinite streams of data. CQL has been developed to run on top of the Stanford STREAM [MWA⁺03] DSMS runtime engine.
- Sawzall [PDGQ05] is a procedural domain-specific programming language developed in 2003 and built upon MapReduce [DG08]. It was developed by Google and was used to process large batches of log records. Each Sawzall script execution takes as input a single log record and processes it on operators deployed on multiple machines. The output is then emitted in tabular form.

Second Generation Systems (2005-2010)

- Borealis [AAB⁺05] is a distributed stream processing system that inherits the core functionalities from Aurora [CcC⁺02] and the networking infrastructure from Medusa [ZSC⁺03]. Aurora is a framework for monitoring applications; its system model is composed by streaming operators that send and receive continuous data streams executing ad hoc queries. Aurora* [CBB⁺03] is an extension of Aurora, and introduces distribution and scalability to the framework. Medusa, on the other hand, is a networking infrastructure that introduced a common networking infrastructure for machines to support the distributed deployment of the Aurora project. Borealis was developed in 2005 and implements all the functionalities inherited by the previously mentioned systems. Borealis is designed to support dynamic revision of query results, dynamic query modification, and to be flexible and highly scalable.
- Stream Processing Core (SPC) [AAB⁺06] is a middleware for distributed stream processing developed in 2006. SPC targets data mining applications, supporting information extraction from multiple digital input data streams. Streaming application topologies are composed by Processing Elements (operators) that implement user-defined operations and are connected by stream subscriptions.
- StreamFlex [SPGV07] is a programming model for high throughput stream processing in Java developed in 2007. StreamFlex extends the Java Virtual Machine (JVM) with type-safe allocation and transactional channels while providing a stricter typing discipline on the stream components of the code.
- DryadLINQ [YIF⁺08] is a system and a set of language extensions developed in 2008 that enable a new programming model for distributed computing on large scale by supporting general-purpose declarative and imperative operations on data sets through a high-level programming language. A DryadLINQ program is composed by LINQ [Mei11] expressions which are compiled by the runtime into a distributed execution plan (called "execution graph", a topology) passed to the Dryad [IBY⁺07] execution platform.
- The Simple Scalable Streaming System (S4) [NRNK10] has been developed in 2010 by Yahoo! and is a general-purpose distributed platform similar to Storm and inspired by MapReduce and the Actor [Agh86] model. With S4 programmers can build topologies that can process possibly unbounded

data streams out of Processing Nodes (PNs) that hosts Processing Elements (PEs, operators). Each PE is associated with a procedure and with the type of even that it is able to consume.

Third Generation Systems (2011-2017)

- Storm [sto11] is a free and open source distributed real-time computational environment originally developed in 2011 by Twitter. Storm works with custom created "spouts" (operators that produce a data stream) and "bolts" (operators that receive streaming elements), which can be arranged into a streaming topology and distributed on one or more hosting machines. Storm takes inspiration from MapReduce as well, where the stream is mapped to many operators and reduced at the end of the computation, with the only exception that in its case operators can theoretically run forever.
- Web Real-Time Communication (WebRTC) [BBJN12] is an API drafted by the World Wide Web Consortium (W3C) in 2011. It enables browser-to-browser connectivity for applications such as video chat, voice calls, and peer-to-peer file sharing. This stream connector bypasses the communication with the server, enabling a direct connection between the Web browsers involved. The Web server's only purpose is to help Web browsers' discovery.
- XTream [DA11, DRAT11] is a platform developed in 2011 that supports the design of data processing and dissemination engines at Internet scale. Topologies are composed by *slets* (operators) and come in three flavours: α -slets (only have an output channel, data producers), ω -slets (only have an input channel, data consumers) and π -slets (both input and output channel, data filters). Slets can be added and removed at runtime, granting high flexibility to the topologies running on top of the platform. XTream is also able to run other kind of topologies (i.e., a CQL topology) within its *slets*.
- Discretized Streams (D-Streams) [ZDL⁺12] is a stream processing engine developed in 2012 whose key idea is to treat the stream as a series of deterministic batch computations on very small intervals. In this way, D-Streams is able to reuse the fault tolerance mechanism for batch processing, leveraging MapReduce-style recovery.

- IBM Streams Processing Language [HAG⁺13, HAG⁺09] (SPL) is a programming language developed for the IBM InfoSphere Streams platform to analyse continuous data streams. SPL abstracts the complexity of dealing with a distributed system by exposing a graph-of-operators view to the users. It provides a code-generation interface to C++ and Java for performance and code reuse. SPL offers static check while providing a strong type system and user-defined operators models. We did not include SPL in the survey table as we were not able to determine its run-time alternatives.
- TimeStream [QHS⁺13] is a distributed system developed in 2013 and designed for continuous processing of big streaming data on large computing clusters. TimeStream's topologies are fault tolerant and are able to dynamically reconfigure themselves to face load changes at runtime. It runs on top of the StreamInsight [AGR⁺09] platform (used to develop and deploy complex event processing applications), and it is designed to preserve its programming model. This way, it can scale any StreamInsight application to larger clusters without modification.
- MillWheel [ABB⁺13] is a framework developed in 2013 by Google for building low-latency data-processing application at large-scale. It allows users to focus entirely on the application logic without the need to think about its distribution. MillWheel offers fault tolerance and guaranteed delivery by checkpointing the state and using the concepts of strong and weak production. Strong production is a concept by which a computation checkpoints its state and forwards the message. When an ACK is received from the receiver, the checkpoint is garbage-collected. Weak production follows the same principles, by which the events may be sent before checkpointing the state; this is only possible if the processing of an event is idempotent with respect to the persistent storage and event production.
- Samza [sam14] was presented in 2013 and is a stream processing framework for writing scalable stream processing applications. Like MapReduce for batch processing, it takes care of running message-processing code on a distributed environment. It offers fault tolerance, scalability and stateful operators to manage the state. It is currently in production use at LinkedIn.
- Curracurrong Cloud [KDFS14] is a light-weight stream processing system for the cloud proposed in 2014. It is designed to be deployed in large distributed clusters hosted on cloud computing infrastructure. It features an algebraic-style description of the processing topology which is automati-

cally deployed on the available computing hosts. This system is based on Curracurrong [KAS⁺14], a similar stream query processing engine suited for Wireless Sensor Networks (WSN) that focus to provide a good trade-off between productivity, flexibility, and energy efficiency.

- StreamScope (StreamS) [LQX⁺16] is a reliable distributed stream computation engine presented in 2016 that provides a continuous temporal stream model that allows users to express complex stream processing logic in a declarative way. StreamS introduces two abstractions, rVertex and rStreams, to manage the complexity in distributed stream processing and deal with failure recovery and allow efficient and flexible distributed execution, facilitating debugging and deployment of complex multi-stage streaming applications.
- Web Liquid Streams (WLS) is the streaming systems we introduce in this dissertation.

2.1.3 Survey Methodology

We decided to focus on the functional characteristics of the proposed set of representative stream processing systems (frameworks and languages) to gather architectural decisions on the software connector. We use these functional characteristics to categorise, classify, and compare the surveyed systems. The architectural knowledge [KLvV06] is organised following the Issue-Based Information Systems (IBIS) meta-model [KR70]. By reconstructing the set of principal design issues of the stream processing systems we surveyed, we present the architectural alternatives proposed in the frameworks and languages we analysed.

We decided to divide the issues we found in two categories: design-time and run-time. On the one hand, design-time issues regard what has to be taken into account for a stream processing system before running it, for example where it can be deployed, or what kind of topology is supported. Run-time issues on the other hand take care of describing what should be decided when the streaming topology is running. For example, if the system is able to balance the load and how it does it, or if it is able to tolerate faults.

We took inspiration by [JBA08] to recover the architectural decisions we gathered in the survey. We based our analysis on the available documentation of the systems, as well as testing them where possible. We decided to include all the issues that resulted having more than one alternative in the state-of-the-art systems. We introduce each alternative, showing benefits and challenges, and including examples taken from the literature.

The criteria followed to choose the systems to include in the survey include the support for distributed stream processing and the availability of prototypes (or implementations) with real-world use case scenarios. On the other hand we omitted similar or closely-related systems as well as previous versions or predecessors of state-of-the-art systems. The final sample of surveyed systems results to be very diverse, but also representative of most design space variants of the stream software connector.

2.1.4 Design-Time

Topology: Linear/Parallel Flows, DAG, Arbitrary

The topology of a data stream infrastructure describes how the streaming operators are interconnected and in which order they process the stream. This design issue impacts the expressiveness of the resulting stream processing infrastructure. We denoted three different alternatives for this issue: linear, directed acyclic graph, and arbitrary.

- **Alternative: Linear** A linear topology is a simple pipeline of operators that execute operations on the data received in linear fashion, much like the shell does when chaining input processes. It is the simplest shape a topology may assume for a data stream to exist, the most basic example being a topology composed by a producer operator streaming data to a consumer operator. This alternative keeps into account both the possibility to build linear topologies and parallel linear topologies, where producers branch the input stream into more linear parallel flows, all executing the same processes. All the stream processing systems we analysed are able to setup a linear topology.

The benefit of this approach lies in the simplicity to design a streaming application. The applications that can be implemented using a simple pipeline have to contain all branching decisions within the operators without having alternative data flows based on intermediate results. Moreover, with the possibility to only branch the producer, having parallel flows can help to cope with workload fluctuations, for example by exploiting the cloud or multicore architecture [GTA06].

This approach brings the inherent challenge of limited expressiveness. Not having the possibility to create branches limits the number of applications that could be developed. Depending on the semantics of the application,

the parallel flow approach may be challenging to implement while maintaining the order of the stream elements when the flow are merged into the consumer. The possibility to split (map operation) the work on many parallel linear topologies may also imply a challenge in joining the streams afterwards (reduce operation).

- **Alternative: DAG** This alternative let operators branch on more than one downstream operator. The data flow can be dictated by a routing algorithm, or by the result of the computation as a branching condition. Not having cycles in the topology is the only constraint this alternative poses.

The higher degree of expressiveness gave by this alternative let developers implement a wider range of streaming applications, with respect to the linear alternative. The possibility to branch the work on different downstream operators is an improvement of the previous alternative and is beneficial for most of the applications that cannot be implemented with a simple linear topology.

The possibility to create acyclic topologies intrinsically increases the complexity in the system to be built as well as in the topologies created by the developers. The system should give the possibility to developers to implement different routing strategies, for example heuristic-based routing, or branching conditions. The possibility to two or more data streams in one operator should be also taken into account (more on this in the appropriate architectural issue).

- **Alternative: Arbitrary** With this alternative, any kind of topology can be set up: with respect to the DAG alternative, this alternative lets developer implement topologies with cycles. This possibility has to be handled carefully, as data may flow indefinitely many times through the topology passing by the same operators.

Besides the expressive power, this alternative gives the possibility to build recursive data streams topologies, useful when dealing with, for example, sensor data in control loops, or for recursive computations (i.e., genetic algorithm [AT99]).

Cycles may create deadlocks of operators that may end up being in a state of waiting their own results as input. In general, cycles have to be handled with care, otherwise streaming packets may end up cycling in the topology forever. If this alternative is chosen, both these challenges have to be addressed.

Topology Representation: Textual Declarative, Textual Imperative

Operators have to be defined and linked together to form and run a topology. We found many ways with different levels of abstractions to define and represent a topology, but decided to categorise them in two main alternatives: declarative languages (i.e., rule-based) and imperative languages. We omitted visual representations as they are usually a high-level abstraction of one of the two representation alternatives we found.

- **Alternative: Textual, Declarative** Declarative programming is a style of programming that minimises, or eliminates, side effects by characterising what the program should compute and not how to compute it. The same approach can be used to describe a topology, letting developers describe them by building the structure of the system without describing the data flow.

Using declarative programming helps focusing on the specification of the result rather than the underlying implementation (that is, how things should be computed). The runtime operations can be guided and optimised through a declarative specification of the topology. It can be beneficial to reuse existing declarative languages and extend them with the notion of stream (i.e., stream-to-relation mappings).

Despite the illustrated benefits, dealing with the high abstraction level intrinsic to the declarative languages may impact the visibility into the actual topology being executed. Moreover, state-of-the-art shows that it may be difficult to deal with stateful operators applied over an infinite data stream. These operators have in fact to deal with the notion of windows or limited-size buffers.

This alternative is being used by Borealis, CQL, Curracurrong Cloud, DryadLINQ, TimeStream, and StreamScope. As the name suggests, DryadLINQ is based on top of LINQ (Language-Integrated Query) ², which is a programming model developed by Microsoft that introduces formal query capabilities into the .NET-based programming languages. TimeStream on the other hand adopts the programming model used by the StreamInsight [ACGS11] framework (again, a LINQ dialect). Likewise, Borealis uses the same model used in its predecessor Aurora, while StreamScope is designed and implemented as a streaming extension of the SCOPE [ZBW⁺12] batch-processing system. CQL is based on SQL (Structured Programming Language), to

²<http://msdn.microsoft.com/en-us/library/bb397926.aspx>

which it adds the capability to deal with a continuous set of queries, and works on top of STREAM [ABB⁺03]. The following snippet shows a simple CQL query returning a relation containing the set of recent orders, that is the ones that have been submitted in the last 30 seconds, for an online store.

```
1 Select Distinct OrderId
2   From OrdersStr [Range 30 Seconds]
```

Listing 2.1. Simple CQL query with a sliding window.

The query shown in Listing 2.1 uses a stream-to-relation operator, the sliding window, that defines the historical snapshot of a finite portion of the stream. In this case the sliding window returns the orders performed in the last 30 seconds in the OrderStr stream. The rest of the query is a simple relation-to-relation operation derived from SQL, which performs projection and duplicate elimination.

- **Alternative: Textual, Imperative** Traditional imperative languages can also be used to implement a stream processing system. Streaming constructs may have to be imported through libraries, or implemented ad hoc in order to include stream processing in an imperative programming language.

The beneficial aspect of this approach is the possibility to reuse the same programming language used throughout an application to implement the streaming operators as well as to configure the topology. Existing code that once was used for a different purpose may be recycled and made as a streaming operator.

The challenge of this approach lies in the abstraction of a streaming topology infrastructure with respect to the programming language: the topology may not be as easy to grasp as its corresponding visual representation. The separation of concerns between the composition and the component may end up being not so clear anymore.

Most of the systems we surveyed implement this alternative. StreamFlex, Storm, S4, and XStream support operator implementation in Java. D-Streams makes use of Scala, while WebRTC and WLS use JavaScript. While we mentioned TimeStream adopting the StreamInsight declarative programming model, users may also use an imperative language to program user-defined operators. Listing 2.2 shows a StreamIt program that performs a moving average over a set of streamed items.

```
1 void->void pipeline MovingAverage {
2     add IntSource();
3     add Averager(10);
4     add IntPrinter ();
5 }
6 void->int filter IntSource {
7     int x;
8     init {
9         x = 0;
10    }
11    work push 1 {
12        push(x++);
13    }
14 }
15 int->int filter Averager(int n) {
16     work pop 1 push 1 peek n {
17         int sum = 0;
18         for ( int i = 0; i < n; i++)
19             sum += peek(i);
20         push(sum/n);
21         pop();
22     }
23 }
24 int->void filter IntPrinter {
25     work pop 1 {
26         print(pop());
27     }
28 }
```

Listing 2.2. StreamIt application that performs a moving average over a set of streamed items.

StreamIt embeds the topology with the `pipeline` and `add` constructs, and defines the operators with the `filter` keyword. `push` and `pop` are used respectively to push an item in the output queue or to fetch an item from the input queue. The topology, in this case a pipeline, shows a producer sending numbers downstream to an `averager` filter which peeks the first 10 items from the input queue and measures the average, pushing it downstream and removing the last item from the input queue. The consumer simply takes the last element from the input queue and prints it.

Deployment Environment: Cluster, Cloud, Pervasive, Web browser

The deployment environment issue aims to define where a stream processing system is able to run. We filtered out four different possible deployment scenarios

from the systems we surveyed: cluster, cloud, pervasive, and browser-based. Every system we surveyed is also able to run on a single machine, besides, we only surveyed systems with the possibility of a distributed deployment (with the exception of CQL). Unlike a single machine, where the parallelisation is bound by the processor cores, a distributed deployment should adopt good load balancing strategies in order to parallelise correctly the work.

- **Alternative: Cluster** A cluster of machines is the standard deployment for a stream processing system that requires high processing power, memory, and storage capacity.

A cluster of machine is one of the alternatives to obtain a significant amount of computing power. The benefit of this approach lies in the advantage of using an optimised local network that reduces the latency and increases the available bandwidth.

Implementing the cluster deployment alternative poses the challenge of the development of a distributed deployment infrastructure, which is shared among all the alternatives proposed in the survey for the deployment issue. The implementation may be thought to be ad hoc for a cluster of machines, or may be generalised to be able to run on a cloud of machines, which can either be nearby or topologically distant. We discuss the latter in the cloud alternative.

This alternative is implicit if the cloud deployment alternative is implemented. Every system we surveyed, with the exception of CQL and WebRTC, support a cluster deployment. CQL only runs on a single machine, while WebRTC needs the support of a cloud infrastructure, either a signalling server or a STUN server, to create the peer-to-peer channels, thus it may not be fit for a cluster deployment.

- **Alternative: Cloud** This alternative is very similar to the previous one, with the exception that the stream in this case runs across a distributed cluster of virtual machines.

With this alternative, producer operators may be moved closed to the data origin, while consumers may be directly served to the data consumers. Adding new resources to this alternative can be as easy as renting a new virtual machine and set up the stream processing system on it, with respect to physically wiring a new machine to a rack of clusters in the cluster alternative.

Despite the benefit of setting up a relatively big infrastructure within a small time frame, the developer has limited control over the topological displacement of the virtual machines in the cloud. This implies less guarantees on the actual network condition, which may result in increased latencies and increased challenges to maintain a given Quality of Service for the resulting stream.

Most of the systems we surveyed are able to run in a cloud environment, with the exceptions of CQL, DryadLINQ, Sawzall, StreamFlex, StreamIt, and WebRTC. All the systems able to run on the cloud implement operators with socket connections which are able to create a data stream to remote virtual machines.

- **Alternative: Pervasive** The pervasive alternative may be seen as a subset of the cloud alternative, taking the distributed deployment to the extreme by enabling the stream processing on sensors and microcontrollers. Given the small processing power and the physical architecture of the smart devices, the cloud alternative may not be suited for such deployment, where a more ad hoc implementation is needed.

This alternative benefits applications that are fed with sensor data, or that feed actuators (i.e., microcontrollers wired to servo motors) as consumers of the topology. Deploying an operator on a sensor acting as a producer may, on one side, avoid the inefficient polling of producers towards sensors, and on the other side optimise the flow of input data by, for example, only inserting interesting data samples.

The challenges to implement such alternative lie mostly in the inherent small infrastructure offered by nowadays sensors and microcontrollers. Although hardware is constantly improving, some of these ubiquitous devices may not be suited yet to handle part of a stream processing infrastructure, thus a general solution is not always possible.

We found Samza, WLS, and XTream to support pervasive deployment. XTream is able to do so with the concepts of α -slets and ω -slets, which are able to incorporate pervasive devices. Also Aurora, the predecessor of Borealis, was developed to access and stream the data produced by sensors.

- **Alternative: Web Browser** The Web is rapidly adopting real-time communication infrastructures in order to speedup the communication between clients and servers, with respect to the synchronous request-response interaction we are used to with HTTP. Thanks to WebSockets [FM11] we are

able to smoothen the connection between clients and servers, and nowadays thanks to WebRTC we are able to connect Web browsers to other Web browsers in a peer-to-peer fashion, effectively creating a stream of data.

Being able to establish browser-to-browser direct connections gives life to a new era of real-time Web applications without the support of a server infrastructure, especially considering that the latest hardware (mobile phone, televisions, gaming consoles) is able to run a Web browser. By performing a GET request to a URL, the Web browser automatically downloads all the associated JavaScript scripts, effectively setting up the infrastructure of the application with a simple page load. No installation on the client machine is required besides the Web browser.

Some interactions with the server should nonetheless be kept into account to organise and establish the bindings with other Web browsers, for example. In fact, a browser is not able to discover other browsers by itself, it needs the support of a server that exchanges the Interactive Connectivity Establishment (ICE) candidates for the browsers involved in the communication, effectively establishing the infrastructure.

The examples that we found implementing this alternative are WLS and WebRTC. WebRTC requires the Web browser to create an ID and request a Channel token from the server. The server in turn requires the token from a STUN server, and sends it back to the clients to setup the direct stream connection. WLS makes use of WebRTC to implement browser-to-browser communication in its topologies.

Join Operator: No implementation, Forward Join, Predicate Join, Function Join

In relational databases, the join clause combines columns from one or more tables and outputs a set that can be saved as a table that contains the entries of interest for the user. In a data streaming system, the join operation happens inside an operator and defines how two or more data streams have to be joined in order to form a single output data stream. Joining two or more streams can be offered as a special kind of operator with a different programming interface from others operators, or may not be specified, leaving to the user the implementation of such functionality. Not all the streaming systems we surveyed specified how the join functionality was implemented. Among those which implemented it, we filtered out the forward join alternative, the user-defined predicate-based join alternative, and the user-defined functional-based join.

- **Alternative: No Implementation** Leaving no implementation for the join operator means having users implement their own join infrastructure. If the system allows it, users may have to deal with coding operators with multiple input streams and a single output stream.

From a development point of view, this alternative may be suited for systems that do not take into account deployment scenarios involving topologies with joins (that is, either multiple consumers or pipelines), thus there is no need to implement it. On the other hand, developers may want to give the possibility to the users to implement their own join infrastructure, thus developing a flexible enough operator component.

If the developers decide not to implement a join infrastructure, the resulting topologies will not be able to perform join operations on the data streams. Leaving the join operation as user-defined implementation may increase the complexity when interoperating with the system.

This alternative is embraced by Sawzall, and WebRTC. The developers of said infrastructures left the join operation implementation to the users.

- **Alternative: Forward (or Echo) Join** The simplest form of join takes two or more input data streams and serialises them in one single output data stream by forwarding the data received in a FIFO order. Each time a message is received, it is immediately forwarded downstream.

This approach simplifies the join operation by transforming multiple data streams in a single one.

The data in the resulting stream may be arriving from completely different sources, thus resulting in tuples with different structure. It is on the user's side to deal with this heterogeneity in the join operator (if possible) by outputting messages formatted the same way.

This alternative is implemented by InfoPipes, StreamIt, and Curracurrong Cloud. StreamIt implements an operator called SplitJoin which is able to parallelise the work by splitting it in one point of the topology and joining it afterwards. The join happens in a round-robin fashion, where streamed data is taken from each input stream in turn. Input queues are implemented with the first come first served policy. The implementation of the Curracurrong Cloud join operator reads the input from one or more input channels and writes it out to an output channel, effectively pipelining what once was parallel.

- **Alternative: User-defined Predicate Join** This alternative is strictly related to the topology representation issue. If the alternative chosen is "Textual, Declarative", then the implementation of a join operator is bound to be predicate-based.

Benefits and challenges of this approach reflect the ones presented for the "Textual, Declarative" alternative.

This alternative is implemented by the systems that implement the "Textual, Declarative" alternative: Borealis, CQL, DryadLINQ, StreamScope, and TimeStream. CQL's join is very similar to SQL's, the only difference being the sliding time window over the streamed elements. Borealis Join is based on Aurora's Join, which is a binary operator with the form $\text{Join}(P, \text{Size } s, \text{Left Assuming } O1, \text{Right Assuming } O2) (S1, S2)$

where P is a predicate over the pairs of tuple from input streams $S1, S2$ (i.e., $P(x, y) \Leftrightarrow x.\text{pos} = y.\text{pos}$); s is an integer and $O1, O2$ are specification of ordering assumed for $S1$ and $S2$ respectively. TimeStream's join is based on StreamInsight, with a slight difference to handle real-time events. Listing 2.3 shows an example of a StreamInsight join, taken from the StreamInsight guide [KG10].

```

1 var innerJoin = from left in outerJoin_L
2   from right in outerJoin_R
3   where left.LicensePlate == right.LicensePlate
4   select new TollOuterJoin
5   {
6     LicensePlate = left.LicensePlate,
7     Make = left.Make,
8     Model = left.Model,
9     Toll = right.Toll,
10    TollId = right.TollId,
11  };

```

Listing 2.3. Example of a StreamInsight join.

The join implementation takes two input streams and, where the licence plates are the same, it creates a new tuple to be forwarded downstream.

- **Alternative: User-defined Function Join**

Like the previous alternative, this alternative is strictly related to the topology representation issue. In this case, if the alternative chosen is "Textual, Imperative", the implementation will be bound to be functional-based.

Benefits and challenges of this approach reflect the ones presented for the "Textual, Imperative" alternative.

Systems implementing this alternative are D-Streams, DryadLINQ, S4, Samza, Storm, WLS, and XTream. In all the studied examples users are able to implement their own join behaviour using the streaming system's programming language. S4 let users subclass the `ProcessingElement` class, effectively creating a join PE. Likewise happens for Storm's bolts and for XTream's π -*slets* that can be custom-created to deal with data joins.

The same approach is taken by DryadLINQ which offers an infrastructure which only need the specification on how the aggregation must be computed. Listing 2.4 shows a join example of PageRank score update producing a list of `<target, score>` pairs starting from `<source, target>` pairs (edges) and a list of current scores (rank) that will be joined together. The example is taken from a list of sample DryadLINQ programs ³.

```

1 public IQueryable<Rank>
2 PageRank(IQueryable<Edge> edges, IQueryable<Rank> ranks)
3 {
4     return edges.Join(ranks,
5         edge => edge.source,
6         rank => rank.source,
7         (edge, rank) => new Rank(edge.target, rank.value)).
8     GroupBy(rank => rank.source).
9     Select(group => new Rank(group.Key,
10        group.Select(rank => rank.value).Sum()));
11 }

```

Listing 2.4. Example of a DryadLINQ user-defined function join.

The result is then grouped using the `GroupBy` function on the first field.

2.1.5 Run-Time

Dynamic Adaptation: Static, Dynamic Operator, Dynamic Topology

This issue determines how flexible the topologies can be. A static topology offers no room for flexibility: once it has been run, it cannot change its configuration nor the load passing through the streaming operators; a change in one of those dimensions implies stopping the topology and re-running it. Dynamic topologies may come in two orthogonal alternatives: whether the topology can be changed

³<http://research.microsoft.com/pubs/66811/programming-dryadlinq-dec2009.pdf>

at runtime (i.e., by re-arranging the streaming operators), or by changing the operator configuration (i.e., by increasing or decreasing operator resource allocation to face changes in the load).

- **Alternative: Static** Once a static topology is deployed, it cannot change at runtime. To change the deployment configuration at runtime, the topology has to be fully stopped and re-run with a new configuration. The static alternative includes systems that cannot change in both the orthogonal dynamic alternatives we analysed.

Picking the static alternative simplifies the runtime of the system, since it does not have to deal with the support for runtime update of the topology.

To achieve a dynamic behaviour (i.e., reordering or removal of operators), developers may use branching operators, effectively skipping parts of the topology. This solution however requires effort from the developer, as all the changes that may happen in the topology at runtime have to be planned in advance.

Among the streaming systems and languages we surveyed, CQL (STREAM), Curracurrong Cloud, D-Streams, Samza, StreamFlex and StreamIt cannot modify the topology configuration after the stream starts flowing through it.

- **Alternative: Dynamic at Operator level**

This alternative proposes the possibility to adapt the operator configuration at runtime. The configuration includes the parallelism level within the operator (i.e., number of threads dedicated to the operator), the physical location of the operator as well as the routing configuration.

Such approach is beneficial when the streaming infrastructure has to deal with load fluctuations that impact the computational effort of the operators, by being able to increase or decrease the footprint on the machine (that is, elastically scale or shrink). This approach is also useful when developers have to deal with a dynamic environment: machine that join and leave the system should be able to host and migrate one or more streaming operators while they are connected to the infrastructure. This alternative may also imply runtime update of the operator code, effectively enabling hotfixing of the operators at runtime.

An infrastructure that has to offer such flexibility needs to be able to support operator migration (possibly without data loss). The system should

also be able to support some kind of dynamic binding infrastructure as the route of the data stream is not fixed, thus increasing the complexity and overhead of the stream connector.

Borealis, InfoPipes, S4, Sawzall, Storm, StreamScope, TimeStream, and WLS use this approach to handle failures. For example, TimeStream, through the concept of resilient substitution, is able to initiate a new operator to replace a failed one and continue the execution (possibly on a different machine). This can also be used to balance the load, by creating more copies of the same operator on multiple machines. Likewise, Storm through the Nimbus daemon is able to restart at runtime streaming operators from one (faulty) machine to another one.

- **Alternative: Dynamic at Topology level**

A fully dynamic topology lets developers add and remove operators from the topology at runtime, as well as rearranging them, effectively changing its semantics.

The high degree of dinamicity given by this alternative gives developers the possibility to alter the semantics of the topology at runtime. This turns out to be especially useful when dealing with a home sensing and automating environment, where the users of the system may want to add one or more sensing agents to their topologies at runtime, without stopping the currently running topology.

This alternative brings intrinsic challenges from the system developer side as well as from the user's side. On the one hand, an infrastructure that offers such degree of flexibility needs to take care of the addition and removal of streaming operators at runtime, as well as re-routing of the operators which should avoid data loss. On the other hand, the user should be aware that removing and adding operators may influence the data sent and received from and to other operators. This could imply inconsistencies (and thus, failures) in the resulting topology.

The systems that we surveyed that apply this alternative are DryadLINQ, InfoPipes, Millwheel, SPC, StreamScope, TimeStream, WebRTC, WLS, and XTream. WebRTC does not impose any constrain on when and how Web browsers should be connected, thus even when a connection is already instantiated, more browsers can connect at runtime. SPC And XTream support the connection of new operators at runtime, the topology can then be extended as the stream flows. The Job Manager component in DryadLINQ

is able to modify the topology at runtime according to user-supplied policies, while TimeStream supports such alternative only if the new configuration is an equivalent substitution of a sub-DAGs of the previous topology.

Operator Disconnection Recovery: Replication, Reconfiguration

The fault of a single streaming operator may compromise the execution of a whole topology. Faults may be caused by the user's code that caused a crash in the execution of the operator, as well as some other hosting machine-related factor (i.e., system-level crashes or disconnections). Some systems like CQL, InfoPipes, and StreamIt do not address the problem, some others rely on the underlying infrastructure – like Sawzall does with MapReduce. Borealis uses the concept of replication to face faults, while S4, Storm, and TimeStream use ad hoc reconfiguration methods. XStream instead exploits the intrinsic dynamicity of its topology to cope with faults.

- **Alternative: Replication**

The concept of replication involves instantiating one or more copies of the same entity through information sharing to improve accessibility, reliability, and fault tolerance. In particular, in streaming systems this means replicating streaming operator to bypass the failure of any of them.

In presence of faults, the topology is able to quickly re-route the data stream from the faulty operator to one of the replicas. Lightweight topologies take the most out of this approach, where the costs of replicating operators and the cost of re-routing remains limited.

This alternative, in addition to the extra resource consumption needed to maintain the replicas, suffers from additional problems that mainly involve maintaining consistency of the replicated state [GHOS96]. The failure of an operator and the subsequent recovery protocol should not invalidate the consistency of all the other replicas.

We mentioned Borealis as an example system that supports this alternative. When a failure is detected, the Borealis infrastructure tries to find an alternative upstream replica to continue processing the data stream. This forces the upstream replicas to be consistent with each other, and this is achieved thanks to the *SUnion* operator which takes as input multiple streams and joins them, outputting a single stream with tuples ordered in a deterministic way. In this way, all the replicas process exactly the same input. Support for this alternative has also been mentioned in the future works of SPC, we

are not aware if the alternative has eventually been developed. Samza and StreamScope also support this alternative.

- **Alternative: Reconfiguration**

With the reconfiguration alternative, topologies are able to reconfigure themselves in order to face operator failures. Faulty operators are immediately substituted or re-launched by the runtime.

Thanks to this approach, there is no footprint caused by replicas of the operators, since the runtime is able to detect the failure and deal with it by restarting or substituting the operator.

As a drawback, the automatic process of reconfiguring the topology needs a reliable monitoring component which constantly checks the operators involved in the topology and detects faults. This can be a single point of failure, even though the system may still be implemented to run without such reconfiguring infrastructure.

We previously mentioned the Nimbus daemon in Storm, which is also in charge of restarting failed operators. If the operator keeps failing, the Nimbus daemon will reassign the execution to another host. A similar approach is taken by WLS with its control infrastructure. D-Streams is able to recompute the lost stream elements while the operator is being reinitialised, thus speeding up the recovery of the topology. DryadLINQ also supports the re-computation of lost elements, as they are re-executed by the Job Manager Component. Once again, TimeStream uses the already mentioned Resilient Substitution principle to replace failed operators, possibly restarting them on another available machine. S4 restarts the failed operators on the remaining available execution resources.

Load Balancing: Load Shedding, Reinitialisation, Adaptive Control

A well-balanced topology can process the data stream at a regular throughput by making good use of the available resources offered by the hosting machines to run the streaming operators. Within some streaming applications it may become difficult to predict the computational effort of each operator, which is based not only on its internal implementation by the user, but also by the values of the streamed elements. The input throughput of the streamed elements should also be taken into account, which may as well fluctuate leading to operator overloading. Different alternatives have been proposed by different systems to deal with

the balance of the load, ranging from dropping specific streamed elements (load shedding) to more adaptive or dynamic alternatives.

- **Alternative: Load Shedding**

Load Shedding is an alternative to reduce the load of streaming operators. The first approaches of this alternative dropped random streaming elements when input queue grew past a certain threshold. More recently, smarter approaches have been proposed which are able to drop elements that are considered to be less important, keeping only the most important ones following the quality of service standards given by the user [GWYL05]. This approach results in a best-effort streaming system, where no guarantees are made on the data reaching the end of the streaming topology, even though some guarantees can be given on the overall throughput as the saturation may be reached without overloading the system.

Load Shedding can be a very easy alternative to implement, given the fact that the underlying application can deal with the loss of stream elements, randomly dropped where topology bottlenecks occur.

More smarter approaches may be implemented in order to avoid dropping the important streaming elements, but this implies that the connector has to know more information about the application semantics of the stream to decide which elements to drop.

Borealis and its predecessor Aurora implement two different Load Shedding alternatives based on quality of service guarantees. Aurora randomly drops tuples that are in the input queue of a streaming operator whose output can tolerate data loss. By dropping randomly selected tuples in strategic points of the topology, this approach effectively reduces the workload of the Aurora topology. Borealis instead uses an approach called Semantic Load Shedding, which drops the least important tuples. Importance once again is determined by a metric called utility interval, computed by observing the quality of service of the application.

- **Alternative: Dynamic Reinitialisation**

The topology is reinitialised through a special token which is forwarded through the whole topology, forcing the streaming operators to flush the remaining stream elements and to reinitialise themselves allocating more resources to face bottlenecks.

The data loss is avoided by stopping the input throughput and flushing downstream the remaining streaming elements. The operators are then

reinitialised allocating more resources, effectively dealing with load balancing issues with the only drawback of a temporary flush of the topology.

The challenging part of this approach lies in the stream protocol, which should be able not only to deal with the streamed data, but also with the special control messages that reinitialise the topology. This control messages may either be triggered manually, or by a control infrastructure which can trigger it without human intervention. If that is the case, a solid control infrastructure has to be implemented to check the status of the queues.

StreamIt and StreamScope are the only systems we surveyed that planned to use this alternative in their future work. In StreamIt, when the topology needs to be modified to face load issues, an `init` message is sent. When the message is received by an operator, it re-executes the initialisation procedure and adjust the resources allocated to run. Even though this was presented as theoretical work, we decided to keep it as feature offered by the system. StreamScope is able to dynamically move to a new configuration with increased degrees of parallelism without interruptions by using checkpoints available at operator level through `rStreams` and `rVertices`.

- **Alternative: Dynamic Adaptive Control**

This alternative proposes the most flexible and adaptive solution, by introducing a control infrastructure which balances the load along the topology at runtime. Bottlenecks are immediately resolved by, for example, allocating more resources to the operator, or by offloading the operator on a more powerful machine.

Thanks to this approach, the topology does not have to stop to balance the load across the available machines. More resources are automatically allocated to the needs of the topology, trading off resources against the performance of the stream.

Real-time monitoring of the system is required, and the controller may need tuning before obtaining good performances. The dynamic reconfiguration should also be safe, by maintaining the semantics of the topology and, depending on the quality of service, avoiding any data loss.

The literature offers many approaches to this alternative. Storm allows to modify the number of workers inside an operator (that is, elastically scale the computation) by the means of a controller, or a GUI/command line administration tool. TimeStream uses once again its Resilient Substitution procedure to automatically change the number of hosts on which a

given operator of the topology is deployed, and thus solving bottlenecks. DryadLINQ exploits hooks in the Dryad API to aggregate operators, effectively reducing I/O operations and improving the overall performance. WLS takes inspiration from Storm, letting the controller modify the number of workers inside an operator, and creating copies of the operator in case the hosting machine used up all of its resources.

Overview

Table 2.1 shows the surveyed systems, highlighting the design decision taken (+). We decided to divide the analysed systems in three different generations based on their age and features, from left to right.

First generation systems (InfoPipes, StreamIt, CQL, Sawzall) share the following commonalities. Topologies are (parallel) linear or DAG, while we have both declarative and imperative representations of the topology. The only target deployment is the cluster of machines (with the exception of CQL). Given their age, it's not surprising that no cloud deployment option has been implemented. The join alternatives offered by the systems are simple, with the exception of CQL that offers user-defined predicate-based join. These systems do not present a flexible topology, with the exception of InfoPipes presenting a primitive dynamic infrastructure, and no system addresses disconnection recovery. As for load balancing, CQL uses load shedding, while Sawzall bases its own on MapReduce; StreamIt uses a lossless dynamic initialisation.

Borealis defines itself in the literature a "second-generation distributed stream processing engine" [AAB⁺05]. We included in this category SPC, StreamFlex, DryadLINQ, and S4 as well. Topologies are more complex, being able to be deployed on the cloud as well as on sensors (pervasive deployment). Join operations results to be totally user-defined, while systems are more flexible at operator level (Borealis, S4) as well as at a topology level (SPC, DryadLINQ). In these systems, disconnection recovery is introduced with the concepts of replication and reconfiguration. Balancing the load shifts from the load shedding approach towards a more dynamic approach with a controller.

The latest state-of-the-art streaming frameworks show a heterogeneous sample by including Storm, WebRTC, XTream, D-Streams, TimeStream, MillWheel, Samza, and Curracurrong Cloud. The topology deployment again includes DAGs, with the exception of D-Streams, while WebRTC also supports arbitrary topologies. The representation has narrowed to the imperative alternative, while the distribution is more in favour of cluster of machines and the cloud (with the exception of XTream and Curracurrong Cloud which can run on a pervasive en-

| | Issue, Alternative | Infopipes (2001) | StreamIt (2002) | CQL (2003) | Sawzall (2003) | Borealis (2005) | SPC (2006) | StreamFlex (2007) | DryadLINQ (2008) | S4 (2010) | Storm (2011) | WebRTC (2011) | XStream (2011) | D-Streams (2012) | TimeStream (2013) | MillWheel (2013) | Samza (2013) | Curraurong Cloud (2014) | StreamScope (2016) | Web Liquid Streams (2017) |
|--|--------------------------------|------------------|-----------------|------------|----------------|-----------------|------------|-------------------|------------------|-----------|--------------|---------------|----------------|------------------|-------------------|------------------|--------------|-------------------------|--------------------|---------------------------|
| | Topology | | | | | | | | | | | | | | | | | | | |
| | Linear | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |
| | Parallel Flows | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |
| | DAG | + | + | + | | + | + | + | + | + | + | + | + | | + | + | + | + | + | + |
| | Arbitrary | | | | | | + | | | | | + | | | | | | | | + |
| | Topology Representation | | | | | | | | | | | | | | | | | | | |
| | Textual, Declarative | | | + | | + | | | + | | | | | | + | | | + | + | |
| | Textual, Imperative | + | + | | + | | + | + | + | + | + | + | + | + | + | + | + | | | + |
| | Deployment | | | | | | | | | | | | | | | | | | | |
| | Cluster | + | + | | + | + | + | + | + | + | + | | + | + | + | + | + | + | + | + |
| | Cloud | + | | | | + | + | | | + | + | | + | | + | + | + | + | + | + |
| | Pervasive | | | | | + | | | | | | | + | | | | | + | + | + |
| | Web Browser | | | | | | | | | | | + | | | | | | | | + |
| | Join Operator | | | | | | | | | | | | | | | | | | | |
| | No Implementation | | | | + | | | | | | | + | | | | | | | | |
| | Forward Join | + | + | | | | | | | | | | | | | | | + | | |
| | User-defined Predicate | | | + | | + | | | + | | | | | | + | | | | + | |
| | User-defined Function | | | | | | | | + | + | + | | + | + | | | + | | | + |
| | Dyn. Adaptation | | | | | | | | | | | | | | | | | | | |
| | Static | | + | + | | | | + | | | | | | + | | | + | + | | |
| | Dynamic: Operator | + | | | + | + | | | + | + | + | | | | + | | | | + | + |
| | Dynamic: Topology | + | | | | | + | | + | | | + | + | | + | + | | | | + |
| | Disc. Recovery | | | | | | | | | | | | | | | | | | | |
| | Replication | | | | | + | + | | | | | | | | | | + | | + | |
| | Reconfiguration | | | | | | | | + | + | + | | | + | + | + | | | | + |
| | Load Balancing | | | | | | | | | | | | | | | | | | | |
| | Load Shedding | | | + | | + | | + | | | | | | | | | | | | |
| | Reinitialisation | | + | | | | | | | | | | | | | | | | + | |
| | Adaptive Control | | | | | | | | + | + | + | | | | + | + | + | | | + |

Table 2.1. Summary of the design decisions (+) over the stream connector design space.

vironment as well, and D-Streams which only runs on a cluster of machines). WebRTC and WLS are the only presented systems able to run on Web browsers. Systems appear to be dynamically adaptable at operator level (Storm, TimeStream, WLS) or at topology level (WebRTC, TimeStream, XTream, MillWheel, WLS). D-Streams is the only exception because of its MapReduce nature. Disconnection recovery is dealt with reconfiguration, and load balancing is done by the means of a controller as standard solution. We can define three different groups: Storm, XTream, MillWheel, Samza, Curracurrong Cloud, and TimeStream leverage the work proposed by StreamIt. D-Streams is an evolution of MapReduce, while WebRTC is a new primitive technology to exploit streams on a Web browser.

The overall view shows an initial trend where the target hardware architectures were fixed (cluster of machines), and barely any topology flexibility. Disconnection recovery was mostly not supported, while load balancing, if implemented, was achieved by relying on the system's underlying runtime platforms. The trend takes a shift over time, by offering a more flexible infrastructure, more deployment options, and disconnection recovery, and finally shifting towards dynamic topologies, dynamic reconfiguration both for load balancing and fault recovery issues, while maintaining unchanged deployment options. It is interesting to observe that this trend towards higher runtime flexibility is obtained while the representation alternative is constrained towards the use of imperative languages.

Web Liquid Streams

Table 2.1 shows the Web Liquid Streams (WLS) framework and the design decisions that were adopted during the development of our streaming framework. The results of the survey encouraged the design decisions we made. We decided to let developers the freedom of implementing an arbitrary topology with JavaScript, an imperative programming language. Our target users include Makers, thus we included Web browsers and the pervasive environment as we expect Makers to use their personal devices to run distributed topologies. The cloud and cluster environment deployment are given by the underlying Node.js infrastructure, able to run on big powerful cluster of local machines, or in the Cloud. To make the system even more flexible, we decided to let the developers implement their own join operator giving a basic infrastructure to handle multiple input messages.

As for what concerns run time issues, we implemented a dynamic environment by letting developers modify, at runtime, topologies and operators, by either rewiring, adding, removing, or updating operators as the topology is run-

ning. WLS deals with disconnection recovery by autonomously reconfiguring the topology at runtime, running the lost operators on available machines. WLS balances the workload on the operators by using a dynamic adaptive controller which is able to distribute the execution of streaming operators on multiple machines at runtime to deal with bottlenecks.

2.1.6 Outlook

The overview of the stream connector evolution, studied through frameworks and design space, highlights different trends. We noticed the rise of architectures and frameworks [JAF⁺06, KCF15, FMG⁺16] targeting microcontrollers and sensor data; we expect this interest to increase in the future, as we are witnessing the rise of hardware like Arduino, RaspberryPi, BeagleBone, or Tessel, which are becoming more and more common and powerful. The capability to run more powerful software may lead to an interest in using microcontrollers as development platforms for more complex applications. This holds for personal computers, home servers, and personal smart devices in general as well, which can be used as platforms to run personal clouds in which sensitive data can be processed and stored. We should also keep into account that the number of devices able to run a Web browser is dramatically increasing, thus we expect to see more and more applications running WebRTC – effectively increasing the number of streaming applications running over the Web browser.

Disconnection recovery (and fault tolerance in general) is likely to become even more important with the latest trend that makes use of a centralised controller, which also balances the load (by increasing or decreasing the computational effort on the hosting machine). We expect future systems to be as dynamic as the latest examples we surveyed (i.e., TimeStream, MillWheel, Samza, StreamScope).

2.1.7 Stream Optimisations

Besides our survey in the stream connector, we also analysed some of the related work on the optimisations for streaming frameworks and systems. During the implementation of WLS (Chapter 4) and its controller (Chapter 5) we took into account the list of optimisation on different streaming systems presented in [HSS⁺14]. Some of the optimisations could be implemented (fission, load balancing, state sharing when possible), while some were more suited for systems that include a compilation phase in which the streaming system could be optimised.

The self-adaptation of streaming application is an important related work for WLS, which controller works without human intervention. In [KCS05] the authors derive an utility-function to compute a measure of business-value given a streaming topology, which aggregates metrics like latency or bandwidth at a higher level of abstraction. A self-adapting optimiser has been presented in [ABQ13] where the authors introduced an online and offline scheduler for Storm. Another optimisation for Storm has been proposed in [SCCH13], where the authors present an optimisation algorithm that finds the optimal values for the batch sizes and the degree of parallelism for each Storm node in the dataflow. The algorithm automatically finds the best configuration for a topology, avoiding manual tuning. Our work is very similar, yet the premises of having fully transparent peers are not met, as WLS is unable to access the complete hardware specifications of a machine from the existing Web browser HTML5 APIs (described in Chapter 4).

2.2 Streaming on the Web of Things

The Web of Things (WoT) is a set of programming patterns and architectural styles that integrate real-world objects into the World Wide Web. WoT reuses most of the existing and well-known Web standards, ranging from the programmable Web (e.g., HTTP communication, RESTful architectures, JSON), the semantic Web (e.g., JSON-LD to link data, Microdata for metadata) and the real-time Web (e.g., WebSockets communication).

During the reminder of this dissertation, we show how WLS is able to abstract the complexity of the underlying hardware by using Web standards. A similar approach is taken by Stream Feeds [DLLW08], which propose an abstraction for the sensor Web which combines the advantages of Web feeds (XML documents that contain dynamically changing sequence of content items) and multimedia stream paradigm. Stream Feeds can be fused, generated, and filtered to create new feeds. Feeds can be accessed through a RESTful API which offers filtering, and historical and incoming data access.

Another approach in the literature that offers a Web abstraction and lets the developer build streaming pipelines is taken by Node-Red [nod13], a browser-based editor that lets developer wire together devices, APIs, and online services. The created dataflow can be deployed on the Node.js runtime and can also be saved in JSON format for future use. Nodes may implement JavaScript functions that are executed on the received data and sent downstream. Unlike WLS, Node-Red does not offer fault detection, flexibility in terms of modifying the topology at runtime, nor supports heterogeneous hardware.

The ECCE toolkit [BAD14] (Entities, Components, Couplings and Ecology) allows users to setup device topologies through XML programming. Like WLS, ECCE reduces the burden of deploying applications over an heterogeneous set of resources. Devices can be bound together in a similar way WLS does (shown in Chapters 3 and 4), while ECCE is more oriented towards I/O events with respect to the data stream WLS processes in its topologies.

The work presented in [RCH⁺04] introduces a component model for sensors and devices around the house. Through a tablet interface, users are able to create very simple pipelines of interconnected devices. It is not clear if more complex scenarios can be built, what are the imposed constraint on the devices, and how the orchestration is performed. WLS treats operators as components to build topologies, and takes care of the orchestration of the sensors and devices.

2.3 Wireless Sensor Networks

The idea of streaming data through a wireless sensor network is not new [MRX08]. While being at a lower abstraction level than the Web of Things, the field of wireless sensor networks proposes related work to Web Liquid Streams.

IrisNet [NDK⁺03] (Internet-scale Resource-Intensive Sensor Network services) is a general-purpose software platform that supports central tasks common to sensing services: collecting, filtering and combining sensor feeds, and performing distributed queries within a reasonable response time. IrisNet is composed by Sensing Agents (SA) which gather sensor data (e.g., webcams) and Organising Agents (OA) which are hosts that keep a distributed database of sensor data and route the queries towards SAs. While being a system at much larger scale than WLS, it proposes an inflexible infrastructure which is difficult to update (e.g., by adding or removing sensors).

Global Sensor Network [AHS06] (2006) is a middleware that supports discovery and flexible integration of sensor networks. Thanks to its virtual sensor abstraction, GSN enables the user to specify XML-based deployment and descriptors with the possibility to integrate sensor data through SQL over local and remote sensor data sources, overcoming the problem of deploying on heterogeneous hardware. In the next Chapters this dissertation will show how our approach in WLS is similar in the way we use the JSON topology and the constraints given by the users, but we do not make use of SQL queries over the data stream.

SwissQM [MAK07] (2007) is an architecture for data acquisition in sensor networks. By separating the gateway and sensor nodes, and implementing a virtual machine at the sensor nodes, SwissQM proposes a richer functionality

in sensor networks and language independence by gathering sensor data at the gateway, and letting users and devices query the gateway, rather than querying the sensors directly. Our approach uses a similar way to abstract the hardware of microcontrollers and single-board PCs by using Node.js, while SwissQM uses a small subset of the Java Virtual Machine. SwissQM is a key element of the XTream platform we presented in our survey.

Previously, in this Chapter we described Curracurrong [KAS⁺14] and Curracurrong Cloud [KDFS14], that let developers deploy part of streaming topologies on sensors.

The concept of wireless sensor networks and the Web of Things can be tightly coupled with Complex Event Processing [CM12, CFS⁺14], a computational approach that takes events from multiple sources and, through analysis, infers patterns or events and responds as quickly as possible. While in this dissertation we often focus on the home automation system example, WLS can also be used to build CEP topologies that gather data from different sources in a Maker's house and infer events that can be dealt with by using actuators.

2.4 Liquid Software Architecture

The liquid software metaphor describes how Web Liquid Streams can operate in a seamless way across multiple heterogeneous devices owned by one or multiple users [GPM⁺17, GPI⁺16, GP17].

The concept of liquid software has been introduced in the Liquid Software Manifesto [TMS14], where the authors use it to represent a seamless multi-device user experience where software can effortlessly flow from one device to another.

Likewise, in [BP11] authors describe an architectural style for liquid Web services, which can elastically scale taking advantage of heterogeneous computing resources. Joust [HPB⁺98] presented an earlier attempt to characterise liquid software, by defining it as mobile code within the context of communication-oriented systems. It presents a complete re-implementation of the Java VM, running on the Scout operating system resulting in a configurable, high-performance platform for running liquid software. In all of the above cases the liquid quality is applied to the deployment of software components.

XD-MVC [HN15] is a more recent example of a framework that lets developer create cross-device interfaces. It implements migration at the application level, simulating the migration of the application through clipping parts of the view. Views are annotated with rules that describe how they should adapt to the pool of available devices. Similarly, the Liquid.js framework [GP16a, GP16b] is one of

the latest examples of a liquid software framework. It helps developer build Web applications taking advantage of multiple heterogeneous devices by extending Web components standards to implement the liquid user experience. Any device can be used in a sequential or concurrent way with applications roaming from device to device without requiring effort by the users.

2.5 Mobile/Cloud

As WLS makes use of mobile devices, we studied how to deal with battery shortages (shown in Chapter 5). While our solution takes only into account the battery level, a lot of effort has been put in studying how to improve the energy efficiency of mobile devices and smart phones, given their rapid growth both in terms of users adoption and computing power. The efforts targeted mostly how to deal with applications consuming a great deal of the smart phone batteries and how offloading computation could save them [KL10]. A high level approach has been proposed in [KSV13] where authors describe MECCA, a scheduler that decides at runtime which applications to run on the cloud and which on the mobile device. This approach differs from ours, in which we use any kind of device *unless* the battery level is too low.

Approaches like MAUI [CBC⁺10] have been proposed regarding mobile phones applications, where programmers could annotate part of the code which could be offloaded. The MAUI runtime is then able to tell if the offloading of the offloadable parts could save energy or not. If that is the case, the server side would execute the method call(s) and send back the result. A similar approach has been taken by CloneCloud [CIM⁺11], which clones the OS running on the smartphone on a cloud VM. While applications are running, CloneCloud can decide which part of the code is convenient to be executed on the cloud replica. With this approach also native calls to the underlying OS can be offloaded.

ThinkAir [KAH⁺12] improves the lack of scalability of MAUI with the same approach of CloneCloud, but removes the restrictions on applications/inputs/environment that the latter includes by adopting an online method offloading. Likewise, the COMET [GJM⁺12] runtime environment is able to offload computations on a cloud VM through a Distributed Shared Memory (DSM) technique and VM-synchronisation operation to keep endpoints in a consistent state. Native calls to the underlying OS cannot be offloaded. Cuckoo [KPKB12] is another framework for computation offloading which integrates with the Eclipse development tool and provides a programming model to write applications that are able to offload part of the computations on a server running the remote service implementation

of the application. A more formal approach is taken in [ZWW13] where authors formulate a constrained stochastic shortest path problem over an acyclic graph for the task scheduling problem.

The next Chapters introduce the Web Liquid Streams framework, its runtime, and its control infrastructure.

Part II

Web Liquid Streams

Chapter 3

The Web Liquid Streams Framework

The Web Liquid Streams (WLS) framework helps developers create stream processing topologies and run them across a personal peer-to-peer Cloud [BMT12] of connected heterogeneous devices. Developers can either install the framework on their devices and run WLS, or they can connect with a Web browser to an existing WLS running instance to execute streaming operators on their browsers. WLS is able to also run on some microcontrollers and single-board PCs, exploiting the I/O functionalities offered and having direct access to sensor data. Thanks to the use of JavaScript, WLS lets developer write streaming operators once and deploy them on any kind of device supported by the runtime, abstracting the underlying hardware.

This Chapter gives a general overview of the framework we developed, as well as how to use it. The purpose of this Chapter is to give the reader a basic understanding of the functionalities of the framework and how developers can use it to develop and deploy data streaming applications through realistic examples.

3.1 Introduction

Web Liquid Streams is a framework that is able to support the execution of distributed streaming topologies. WLS offers the deployment of stateful streaming operators on Web servers (Cloud), Web browsers, and some off-the-shelf microcontrollers and single-board PCs. This is possible thanks to the underlying runtime entirely developed in JavaScript, the lingua franca of the Web. By exploiting the Node.js¹ framework (a JavaScript runtime built on Chrome's V8 engine that

¹<https://nodejs.org/>

lets developer execute JavaScript code outside of a Web browser) we are able to run parts of a streaming topology on Web servers and on microcontrollers and single-board PCs that either support JavaScript (i.e., Tessel²) or that can be installed on them (Arduino's Noduino³, RaspberryPi through Node.js). Most of the available Web browsers natively run JavaScript and thus can be used to offload part of the computation.

To build and deploy a streaming topology, the developer must become familiar with the following WLS primitives.

- **Peers** are physical hosts where the data processing happens. The peer abstraction holds in the physical view of the topology, where one or more computational entities connects to the cloud of computing devices and offers its own resources. Any Web-enabled device that can run a Node.js Web server or a Web browser can become a Peer in the Web Liquid Streams framework. A Peer may host one or more streaming operators.
- **Operators** receive incoming data stream elements, process them, and forward the results downstream. Each operator is associated to a JavaScript file, which describes the stream element processing logic. Operators are hosted in peers, and are part of the logical view of a topology.
- **Workers** are single processes spawned within operators. They are used to parallelise the work of a single operator. Operators are stateful, and start by default with a single worker, this number can elastically change to accommodate additional resource demand by the execution of the script associated with the operator.
- **Topologies** describe how operators are interconnected in the data stream. They define an arbitrary graph of operators using data flow bindings (edges of the topology). The structure of the topology can dynamically change while the stream is running.

Figure 3.1 shows how peers, operators, and workers can be organised in a topology. The first peer is hosting two operators, which stream data to a single central operator, which is deployed on two different peers. The arrows represent the logical flow of the stream (from operator to operator) and not the physical wiring (from peer to peer). The central operator forwards the stream to a peer hosting the final operator.

²<https://tessel.io/>

³<https://sbstjn.com/noduino/>

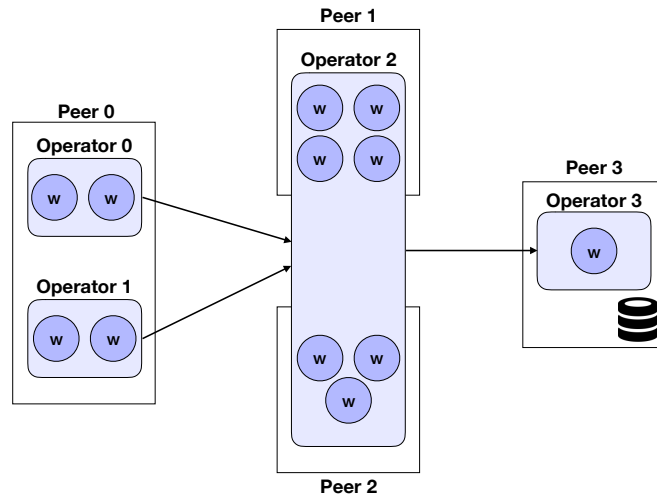


Figure 3.1. Example of a distributed streaming topology running on different peers. Peer 1 and peer 2 in this particular case host both the computation of operator 2.

Developers must first connect peers to the system in order to use them to offload operator execution. Each JavaScript script implements the functionality of an operator, which starts by default with one single worker executing the script. More workers may be started for a single operator during the startup phase, otherwise the control infrastructure will add them at runtime if bottlenecks in the execution are found. Workers can in fact be added or removed at runtime to face load changes and solve bottlenecks caused by a slow operator execution.

3.1.1 The Liquid Software Metaphor in WLS

The metaphor of liquid software is used to illustrate the user experience during the interaction with software deployed across more than one device. Like a liquid adapts its shape to the one of its container, WLS is able to adapt the stream computation footprint on the pool of available resources. When the resource demand increases, new resources are allocated and de-allocated whenever the resource demand decreases. The computation is autonomously adapted to the workload on the set of devices being used, seamlessly splitting and migrating streaming operators when needed. WLS follows the principles of the liquid software metaphor [MSP15, TMS14] by adapting to the set of devices being (concurrently) used, seamlessly migrating running applications across devices and synchronising the state distributed across two or more devices (through stateful operators).

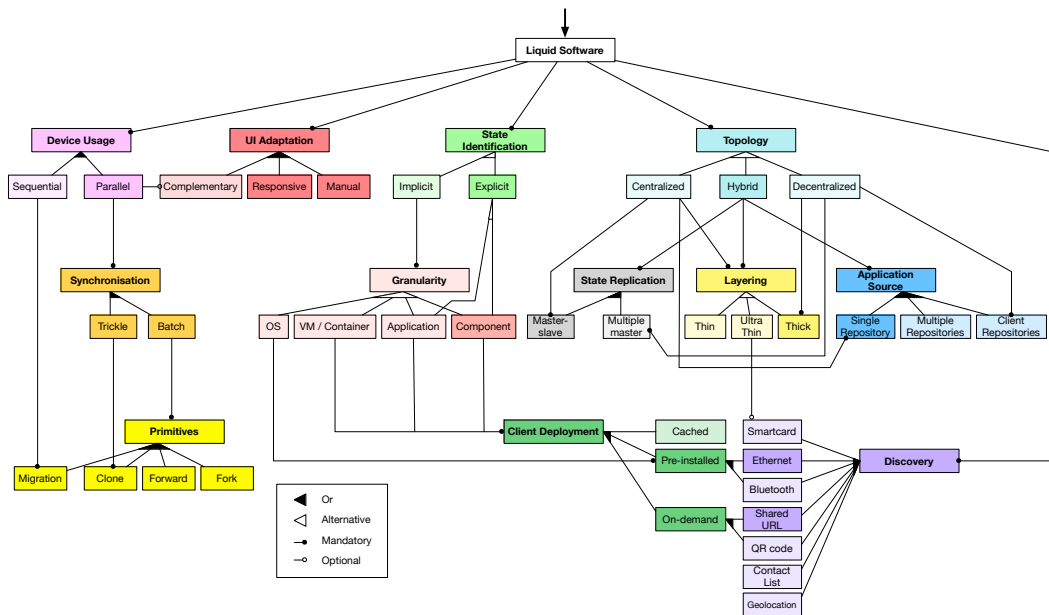


Figure 3.2. The Web Liquid Streams features in the liquid software feature model introduced in [GPM⁺17].

Figure 3.2 shows the feature model for the liquid software metaphor, introduced in [GPM⁺17]. The bright rectangles represents our design choices, while the blurred rectangles are the decisions that were not taken but are part of the liquid software domain. The model is slightly adapted for WLS, which may not include applications with a UI.

WLS uses a hybrid approach for the topology of the data stream. The starting point of the streaming topology is centralised, while the operators are spread across different devices, each one holding its own code (and state, in case of stateful operators). The state replication is in a master-slave form, implemented with Redis and described in Chapter 4. The layering adopted in WLS is considered thick, as each peer holds the operator code as well as the control infrastructure. The starting peer also holds data regarding the topology and the connected peers.

The discovery happens either through the pre-installed WLS runtime on Web server peers, or through shared, on-demand, URLs for Web browser peers. Web server peers are given the IP address of the starting peer to which they can connect through ethernet/WiFi and become part of the computation, while Web browser peers only need to connect to specific URLs in order to start the computation of an operator, or become idle peers. This design decision also impacts the client deployment, which is in fact both pre-installed (Web server) and on-demand (Web browser).

The state identification defines how the developer of the application identifies and specifies how the state is shared among the devices. WLS lets developers explicitly define what to share at operator level (that is, what to send downstream to other operators, or what to save on the database in the case of a stateful operator). For this reason the design choice is explicit, and is component based as we treat each operator as a component running on a different device.

The computational adaptation of WLS can be complementary, manual, and responsive. The controller decides how to spread the operators on different Web browsers, if it causes bottlenecks. The controller is also in charge to take decisions on how many resources to allocate on the operator at runtime, making it responsive. We also consider the manual design choice, as the user of the system may decide how to allocate resources on a given device.

The device usage is parallel, as peers host part of the application at the same time in a parallel way. The synchronisation of the operators happens in two different layers, depending on whether an operator is stateful or stateless. The synchronisation of the state for stateful operators is handled by the underlying database infrastructure. When a stateless operator is moved on a different peer, or forked to parallelise execution, the process batches the whole state of the operator (that is, number of workers running in the operator) and moves it on another peer. For stateful operators, we consider clone as the primitive – the cloning of the database structure handled by the runtime through master-server replication. The primitives for stateless operators are migration, to move an entire operator from one peer to another, and fork, to split the execution of an operator on more than one peer.

WLS fits in the maturity model for liquid software applications and frameworks introduced in [GP17]. The maturity model is determined by combining the deployment configuration of the liquid applications model view controller (MVC) layers across server-side and client-side. WLS fits the model proposed even if topologies do not always have a view.

Figure 3.3 shows the maturity model for Web architectures for centralised, decentralised and distributed model layer deployments. If we consider the starting peer of a topology as the "server" and other peers connected to it as the "clients", WLS can be seen as a decentralised hybrid Web application (level 4, highlighted in green). WLS is a hybrid Web application as it decentralises topology execution while maintaining a degree of centralisation for what concerns control of the topology. The topology model is bound to the starting peer, while every other peer connected and part of a topology holds part of the model as well as the state (in the case of stateful operators).

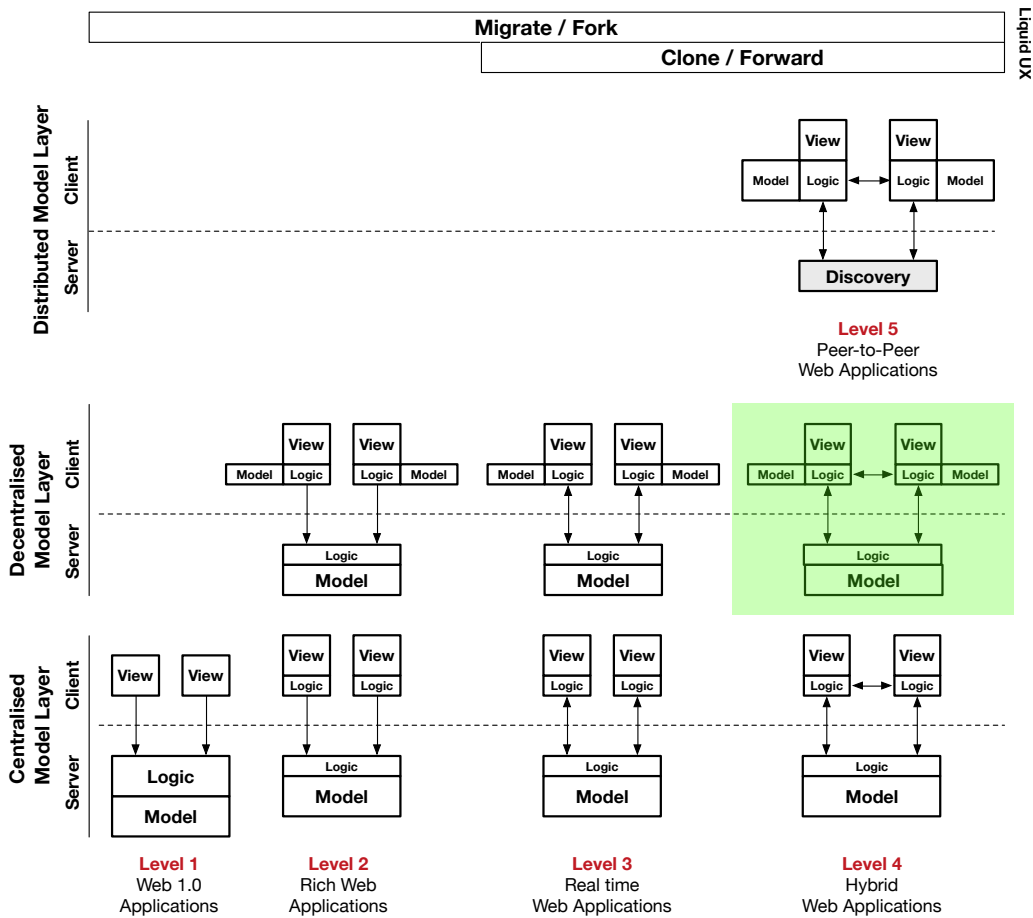


Figure 3.3. WLS in the maturity model for liquid Web software applications [GP17].

3.2 System Model

Our system model is composed by a set of networked peers owned by the users of the framework. Each peer can be of different nature: big Web servers with a great deal of storage and RAM, Web browsers running on personal computers or smart devices, and microcontrollers and single-board PCs. A single topology may be deployed on different heterogeneous peers, depending on the needs of the streaming application. WLS is able to handle the churn of connecting peers, notifying the appropriate peers of the new incoming connections. Likewise, if a peer has to leave the network, running topologies will be rewired – if needed – to deal with the disappearing peer by migrating operators elsewhere.

By uploading a topology description in the form of a JSON file, or by manually specifying the operators and bindings through the user interface, users are able

to configure and start a data stream topology. The system checks if the requests to run a given topology could be satisfied with the known resources. If that is the case, operators will be allocated on the available peers and the topology will start. More than one topology can coexist at the same time within the same set of peers. In fact, more than one operator (either from the same topology, or from different ones) can run on the same peer. Operator scripts can be updated at runtime by the user, if for example a bug is found, or if the functionality of an operator has to be updated to accommodate changes in the topology. In fact, topologies may also change semantics at runtime: WLS supports adding and removing operators while a stream is running, effectively changing the end result of the computation. This can prove to be useful when, for example, a new hardware with new sensors is added, and the topology has to be expanded to accommodate another producer operator.

When a peer starts a topology, the newly created topology refers to that peer as the starting peer, or root peer. The root peer deals with the churn of joining and leaving operators, fixing the physical wiring of the topology if needed. The command-line control interface, as well as the RESTful API for that topology are available only on the starting peer.

Operators are implemented by the users themselves, thus no quality of service is guaranteed; operators crashing resulting in application failures at runtime should be handled by the users running the topology. The WLS framework ensures cohesion among the heterogeneous set of peers that offer their resources as platform for stream processing.

3.3 Developing Operator Scripts

Operators are written in plain JavaScript and include the WLS library that provides all the functionalities needed for connecting operators into a streaming application. In this Section we show how to write operator scripts for a simple topology. We demonstrate how to interact with sensors, the Web browsers, and the database support. The following list of commands shows the basic API to interact with WLS. It describes the basic operations, topology operations, stateful operations, and Web browsers-only related methods. Users only need to `require` the WLS library in their scripts to access the API.

Basic WLS Method Calls

- **`createWorker(Function callback(Object msg))`** Procedure call to set up the functionality of a single worker. The callback function is executed each

time the worker is assigned to process a message by the operator. The callback function takes a `Object` as input parameter, which is the deserialised version of the message received.

- **createJoin(Function callback(Array [msgs]))** Procedure call to set up the functionality of a single worker in the case of a join operator. The callback function is executed each time the worker is assigned to process a batch of messages received from every endpoint. The callback function takes an array of `Strings` as input parameter, which corresponds to the serialised version of the messages received.
- **send(Object msg)** Function used to send messages downstream. It accepts JavaScript objects as well as strings.
- **sendContentBased(Object msg, String dest)** Sends messages downstream to the specified operator. `dest` can either be the operator id or the alias (specified in the topology description file). If no destination is specified, the message will be sent to all the connected operators.
- **hasIncomingBinding()** Checks if the operator where the worker is running has an incoming binding from one or more operators. Returns `true` if there is at least a binding, otherwise it returns `false`.

Topology Operations Through Operator Functions

- **addOperator(Integer oid, String script, Integer workers, Bool automatic, String alias, Function callback())** Creates a new operator on the machine currently running the worker that called the method. The procedure needs an operator ID, a string representing the script to be run, the number of workers to be started, a boolean value that is used to tell the controller to either load balance this operator (`true`) or not (`false`), an alias (i.e., "noise_sensor"), and a callback function that is executed after the operator started running.
- **bindOperator(Integer/String from, Integer/String to, Function callback())** Binds operator `from` to operator `to` and calls the callback function when done. The two operators can be passed as IDs or aliases.
- **unbindOperator(Integer/String from, Integer/String to, Function callback())** Unbinds operator `from` and operator `to`. The callback function is called when the unbind has finished executing.

- **migrateOperator(Integer/String oID, Integer pID, Function callback())**
Migrates the execution of the operator `oID` (that can be addressed either with its ID or its alias) to the peer with ID `pID`. The callback function is executed when the migration is completed. In case of failure, an error is prompted on screen.
- **getWorkersNumber(Integer/String oID, Function callback(Integer result))**
Inquires for the number of workers currently running in a given operator with ID/alias `oID`. The callback function takes as single parameter the number of workers.

Stateful OperatorMethod Calls

- **stateful.plainSet(String key, Object value, Function callback(String result))**
Sets a value at a given key. If the key previously stored a value of another type (i.e., trying to save a String on top of an Integer), an error will appear in the `result` of the callback. Unlike the `stateful.set` function, this function does not automatically stringify the received value, thus if it is not a String or an Integer, an error will be returned. The callback function is called with the result of the set (i.e., if it succeeded or not).
- **stateful.plainGet(String key, Function callback(String/Integer result))**
Gets a value at a given key. The callback function is called with the result of the get, which is either a String (i.e., stringified Object) or an Integer.
- **stateful.set(String key, Object value, Function callback(String result))**
Sets a value at a given key. Unlike `stateful.plainSet`, this function automatically stringifies the key if it's an Object. If the key previously stored a value of another type (i.e., trying to save a String on top of an Integer), `result` will contain an error. The callback function is called with the result of the set (i.e., if it succeeded or not).
- **stateful.get(String key, Function callback(String/Integer result))**
Gets the value stored at a given key. The callback function is called with the result of the get as a String or Integer.
- **stateful.incr(String key, Function callback(String result))**
Increments the value stored at the key. If there is no value at the given key, `stateful.incr` will behave as if 0 was stored. If a value which is not an Integer is stored at the key, `result` will contain an error. If there is no error, the callback function is called with the incremented Integer.

- **stateful.decr(String key, Function callback(String result))** Decrements the value stored at the key. If there is no value at the given key, `stateful.decr` will behave as if 0 was stored. If a value which is not an Integer is stored at the key, `result` will contain an error. If there is no error, the callback function is called with the decremented Integer.
- **stateful.addToSortList(String setName, Array args, Function callback(String result))** Adds the specified members with the specified scores in a sorted set stored at key `setName`. The `args` array should contain a score with a member, for example `[1, "one", 2, "two"]` where the strings are the members and their preceding number is their score. This is stored in a sorted list at the given key which updates each time a new member with a new score is added. Objects to be added need to be stringified. The callback function returns the number of elements added to the sorted sets, not including elements already existing for which the score was updated.
- **stateful.getRangeSortList(String setName, Integer start, Integer end, Bool withScore, Function callback(Array result))** Returns the content of the sorted list stored with the `stateful.addToSortList` function. The `start` and `end` input values define the range of the inspected list (with `end` set to -1 all the elements are returned). If the scores are to be displayed (`withScore` value is true), the results are returned in a `[member, score, member, score, ...]` fashion ranging from the highest score to the lowest score in the callback function. If the scores are not to be displayed, the results will be returned with members only, showing from the highest scored member to the lowest scored member. If Objects were previously stored inside the sorted list, they will be displayed as Strings.
- **stateful.incrBySortList(String setName, Integer incr, String key, Function callback(Integer incrementedValue))** Increments by the given value `incr` the value associated with the given key `key` in the given sorted list named `setName`. The callback function is passed as a single parameter the Integer representing the new value after the increment is applied.
- **stateful.lpush(String key, String value, Function callback(Integer listLength))** Pushes `value` in the first place of the list `key`. If an Object is pushed, it has to be converted to String before passing it to the function or it will throw an error. The callback function is fired with an Integer parameter representing the length of the resulting list.

- **stateful.lrange(String key, Integer start, Integer stop, Function callback(Array resultList))** Reads the list named `key` from position `start` to position `stop`. The callback function takes as parameter the resulting list.
- **stateful.llen(String key, Function callback(Integer listLength))** Inquires the length of the list called `key` within the callback function as an Integer.
- **stateful.ltrim(String key, Integer start, Integer stop, Function callback(String result))** Trims the list stored at `key` from `start` to `stop`. The callback function is passed an input value representing the result of the execution on Redis (i.e., if it succeeded or not).
- **stateful.del(String key, Function callback(Integer result))** Removes the content stored at `key`. The callback function is passed an input value representing the number of elements removed.

Browser-related Method Calls

- **createHTML(String ID, String HTML)** Function used to add HTML content in the Web page running the worker. The `ID` is used to identify the block of HTML, to avoid parallel-executing worker to recreate the same block. The `HTML` input string represents the HTML to be added to the page, which is added sequentially to the page.
- **createCSS(String ID, String cssFileName)** Function used to add a CSS file to the Web browser worker execution. The `ID` is used to identify the added file, to avoid parallel-executing workers to add the same file. `cssFileName` represents the name of the file to be added to the page, which should be added beforehand to the `/public` directory of the WLS framework.
- **createScript(String ID, String scriptName)** Function used to add a JavaScript file to the Web browser worker execution. This is useful when dealing with external libraries (i.e., visualization libraries). The `ID` is used to identify the added script, to avoid parallel-executing workers to add the same script. `scriptName` represents the name of the script to be added to the page, which should be added beforehand to the `/public` directory of the WLS framework.

- **callFunction(String functionName, Array args, Function callback(Function returnValue))** Calls a function named `functionName` defined in a script imported with `createScript`. The array `args` contains the arguments of `functionName` in the correct order. At the end of the execution, the function `callback` is executed with the return value of the function execution.

We now show how to code a simple topology that reads data taken from sensors, stores it into a database, and visualised on a Web browser. Figure 3.4 shows a picture of the logical view of such linear topology, composed by a producer forwarding sensor data to a stateful filter, which in turn forwards computed results to a consumer.

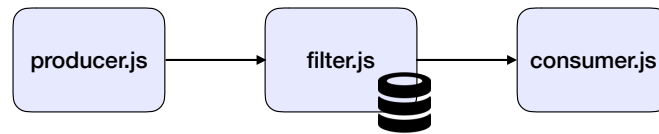


Figure 3.4. Logical view of the topology we illustrate in this Chapter.

```

1 var k = require('../k_global/WLS.js');
2 var sensorLib = require('node-dht-sensor');
3
4 var sensor = {
5   initialize : function() {
6     return sensorLib.initialise(22, 77);
7   },
8   read : function() {
9     var readOut = sensorLib.read();
10    k.send({
11      temperature : readOut.temperature.toFixed(2),
12      humidity : readOut.humidity.toFixed(2),
13      timestamp : new Date().getTime()
14    });
15    setTimeout(function() {
16      sensor.read();
17    }, 1000);
18  }
19 };
20
21 if(sensor.initialise()){
22   sensor.read();
23 } else {
24   console.warn("Failed to initialise sensor.");
25 }
  
```

Listing 3.1. Example producer operator.

Listing 3.1 shows a simple producer operator running on a Raspberry Pi. We wired a sensor that measures temperature and humidity to a Raspberry Pi. We want to read the sensor data every second, format it, and send it downstream.

On the first two lines we import our library in a variable called `k`, and a sensor library which makes inquiring the Digital Humidity and Temperature (DHT) sensor easy. We then create an object containing two functions, one to initialise the library and pointing it on the correct pin, and the other to deal with the read function. Each time we read the data from the sensor, we call the `k.send()` function to send the temperature and the humidity downstream. We also set a

timeout such that every 1000 milliseconds we call `sensor.read()`, effectively polling the DHT sensor every second. The final part of the script checks if the sensor library is initialised, otherwise a `console.warn` prints on screen.

```

1 var k = require('./../k_global/WLS.js');
2
3 k.createWorker(function(raw_data) {
4   k.stateful.lpush('sensorData', JSON.stringify(raw_data),
5     function(response) {
6     if(response) {
7       k.stateful.lrange('sensorData', 0, 60, function(
8         response) {
9         var hum_mean = 0;
10        var temp_mean = 0;
11        var light_mean = 0;
12        var date = new Date();
13        var time = date.getHours(date) + ":" + date.getMinutes(
14          date);
15
16        for(var i = 0; i < response.length; i++){
17          response[i] = JSON.parse(response[i]);
18          hum_mean += parseInt(response[i].humidity);
19          temp_mean += parseInt(response[i].temperature);
20        }
21
22        var to_send = {
23          "temperature_mean" : temp_mean/response.length,
24          "humidity_mean" : hum_mean/response.length,
25          "time" : time
26        }
27        k.send(to_send);
28      });
29    }
30    else {
31      console.warn('There was an error processing the stored
32        data.');
```

Listing 3.2. Example of a stateful filter operator.

Listing 3.2 shows the filter operator running on a Web server. We use this operator to store the data measured by the producer(s) and compute a mean of the last sixty measurements. We create an object containing the temperature and humidity means as well as a timestamp, and forward it downstream.

After importing the WLS library, by calling the `createOperator` function we create the filter operator. Its logic is described in the callback function passed to `createOperator`. This callback is executed each time a message is received; in this case we first call a database function, `k.stateful.lpush` and store data in our Redis infrastructure using a user-generated id in this case `'sensorData'`). If there are no errors, we call the `k.stateful.lrange` function to get the last sixty measured values. We use those to compute the humidity and temperature means after parsing the string we read. We then prepare the object to be sent downstream and subsequently send it.

```
1 var k = require('../k_globals/WLS.js')
2
3 k.createWorker(function(polished_temperature_data) {
4   k.callFunction("updateGraph", [polished_temperature_data.
5     temperature_mean+"", polished_temperature_data.
6     humidity_mean+"", polished_temperature_data.time]);
7 });
8
9 k.createHTML('canvas_temperature', '<h1>Temperature Chart</h1>
10 <br/><canvas id="canvas_temperature" width="900px" height
11   ="500px"></canvas>');
12
13 k.createHTML('canvas_humidity', '<h1>Humidity Chart</h1><br
14   /><canvas id="canvas_humidity" width="900px" height="500px
15   "></canvas>');
16
17 k.createScript('graph_functions', 'js/graph_functions.js');
```

Listing 3.3. Example consumer operator running on a Web browser.

Listing 3.3 shows the code of an example consumer. The consumer runs on a Web browser and shows the mean values computed in the filter operator on a graph. This gives a clear visual output to the user on the environment in his house/office. After requiring our library, we proceed to create a worker of the consumer operator. The operator executes the `callFunction` procedure, which – as the name suggests – calls a function called `updateGraph` and passes the computed means as well as the time received from upstream. The function we call is defined in another file, `graph_functions.js`, which is imported in the last line of the script by calling the `createScript` procedure and passing a string identifier for the script, and the path. The rest of the operator just creates the HTML infrastructure (a title and a canvas) needed by the visualisation to build the graph.

The `js/graph_functions.js` file contains all the functionalities needed to build a self-updating graph. We won't show the code as it goes beyond the scope of this example.

3.4 Deploying a Streaming Topology

3.4.1 Command Line Interface

After installing the WLS framework and starting it, users will be prompted to a command line interface. By typing `help`, WLS returns a list of available commands.

| | |
|--|--|
| <code>help</code> | show help |
| <code>ls</code> | Lists all peer resources |
| <code>ls peers</code> | Lists all server peers |
| <code>ls remote</code> | Lists all remote (Web browser) peers |
| <code>ls scripts</code> | Lists all the known operator scripts |
| <code>ls bindings</code> | Lists all the bindings |
| <code>ls processes</code> | Lists all running workers |
| <code>ls operators</code> | Lists all running operators |
| <code>ls connections</code> | Lists all the operators and their connections |
| <code>run {src}</code> | Runs the script <code>{src}</code> on a new worker on any peer |
| <code>run {src} [pID]</code> | Runs the script <code>{src}</code> in peer <code>[pID]</code> . If <code>[pID]</code> is not specified, it will be run on the first available peer |
| <code>runw {src} [pID]</code> | Runs the script <code>{src}</code> on a new worker in peer <code>[pID]</code> . If <code>[pID]</code> is not specified, it will be run on the first available peer |
| <code>runc {src} [pID] [num] [flag]</code> | Runs the script <code>{src}</code> in <code>[num]</code> copies in peer <code>[pID]</code> with flag <code>[flag]</code> . If <code>[pID]</code> is not specified, it will be run on the first available peer. If <code>num</code> is not specified, it defaults to 1. If no flag is specified, the program runs the operator as if it was run with the <code>-a</code> flag |

| | |
|---------------------------------|--|
| Flag : -a | Automatic control. The operator's number of workers is adjusted automatically by the controller (if started) |
| Flag : -m | Manual control. The operator's number of workers is not adjusted by the controller, but manually (by the user) |
| stop | Stops the topology |
| update_script {src} {oID} | Updates the script {src} in the operator with oID {oID} |
| bind {oID1} {oID2} | Connects operator {oID1} to operator {oID2} |
| bindw {oID} {wID} | Connects operator {oID} to worker {wID} |
| bindc_content_based {wID} {oID} | Connects worker {wID} to operator {oID} in a content_based sending algorithm (hash function to be definend in the koala.js file) |
| unbind {wID1} {wID2} | Disconnects worker {wID1} from worker {wID2} |
| unbindc {oID1} {oID2} | Disconnects all the workers running in the operator with id {oID1} from all the workers running in the operator with id {oID2} |
| addworker {src} {oID} {pID} | Adds a worker to an operator with id {oID} and running script {src} on peer {pID} |
| migrate {oID} {pID} | Migrates the operator {oID} to the new location peer {pID} |
| kill {wID} | Kills worker {wID} |
| killp {oID} | Kills a (random) worker in an operator {oID} |
| killall {pID} | Kills all workers of every operator in peer {pID} |
| killc {oID} | Kills an operator {oID} with all the workers running in it |
| start_controller | Starts the controller |
| sc | Starts the controller (shortcut version) |

| | |
|--------------|--|
| exec file.k | Executes the content of a .k file (list of manual inputs) |
| exec file.js | Executes the content of a .js file (topology description file) |

Table 3.1. List of WLS command line interface commands.

Table 3.1 shows the commands available to the command line interface (CLI). Once the operator scripts are ready, users can run and manage their topologies using the illustrated commands. Through the use of the CLI, we are able to set up the topology by running the operators on the appropriate peers.

```

1  run dht_producer.js 1
2  dht_producer.js is now running on peer 1 with operatorID 0
3  run dht_filter.js 0
4  dht_filter.js is now running on peer 0 with operatorID 1
5  run dht_consumer.js 2
6  dht_consumer.js is now running on peer 2 with operatorID 2
7  bind 0 1
8  operatorID 0 bound to operatorID 1
9  bind 1 2
10 operatorID 1 bound to operatorID 2

```

Listing 3.4. Command execution in the WLS CLI.

Listing 3.4 shows the commands to be executed on the CLI to setup a simple linear topology. We call the `run` commands specifying the scripts to be run and the `peerID`, that is the identifier of the peer where we want the script to be run. In our case, `peerID 0` represents a Raspberry Pi, `peerID 1` is a home Web server, while `peerID 2` is a Web browser (Peer IDs are assigned as the hardware connects in an increasing fashion). Once the scripts are running, we use the system-generated operatorIDs to specify the bindings of the topology. We bind `operatorID 0` to `operatorID 1`, that is `dht_producer.js` to `dht_filter.js`. Then we bind `dht_filter.js` to `dht_consumer.js`. The data stream will start automatically as the binding is performed.

The shown list of commands can be written in a topology file (with the `.k` extension) and passed to the WLS runtime with the `exec` command (i.e., `exec topology.k`). While writing the file, the user should keep into account the operatorIDs. Each command in the file is executed sequentially. For example, by writing the command to run the producer operator first on a Raspberry Pi, `run dht_producer.js 0`, the assigned operatorID to the producer would be 0 (zero), since it is the first one to be run.

3.4.2 Topology Description File

The command line interface can be easily used to quickly deploy small topologies, or to test operators before actually deploying them on a live application. However, for big and complex topologies, this approach may not be the most suitable one to adopt from the point of view of the user. Wiring and deploying by hand big DAG topologies can take time and can be error prone, given the fact that the user should keep in mind many different operatorIDs and peerIDs.

For this reason we decided to give the user the possibility to feed a JSON file describing the logical topology to our runtime. The idea is to use the structure JSON offers to organise operators and bindings as array of objects, each one describing the structure of the operator and binding to be created. The result is a human-readable topology configuration which becomes easier to write and to maintain. Listing 3.5 shows the topology schema that describes the JSON to be fed to the CLI when creating a new topology.

```
1 {
2   "title": "Topology Schema",
3   "type": "object",
4   "properties": {
5     "id": {
6       "type": "string"
7     },
8     "operators": {
9       "type": "array",
10      "minItems" : 1,
11      "items" : {
12        "type" : "object",
13        "properties" : {
14          "id" : { "type" : "string" },
15          "script" : { "type" : "string" },
16          "sensors" : {
17            "type": "array",
18            "minItems" : 1,
19            "items" : { "type" : "string"}
20          },
21          "workers" : { "type" : "integer" },
22          "browser" : {
23            "type" : "object",
24            "properties" : {
25              "path" : { "type" : "string"},
26              "only" : { "type" : "boolean"}
27            }
28          },
29          "max-workers" : { "type" : "integer" },
```

```
30     "min-workers" : { "type" : "integer" }
31     }
32   },
33   "bindings": {
34     "type": "string",
35     "minItems" : 0,
36     "items" : {
37       "type" : "object",
38       "properties" : {
39         "from" : { "type" : "string" },
40         "to" : { "type" : "string" }
41       }
42     }
43   }
44 }
45 }
46 }
```

Listing 3.5. Stream topology schema.

The `operators` key contains an array of objects describing which operator runs which script and the number of workers that should be started in the initial deployment configuration of the operator. If no number of workers is provided, the operator will be started with one worker by default. The `sensors` key specifies an array of sensors that may be needed by the operator to run. The `browser` key specifies whether browsers can connect to this operator and become part of the topology, by specifying the `path` to which users should connect to make their browsers interact with WLS and a boolean value specifying if the operator can only run on Web browsers (`only`). Web browsers can also connect to a more generic – and always available – URL, `/remote`, which doesn't wire them to a specific operator, rather pools the Web browser resources for future use (i.e., deployment of new topologies, target migration, etc.). The `max-workers` and `min-workers` keys define the maximum or minimum amount of workers that can be run for that operator. By default, the minimum is one and the maximum depends on the available resources on the hosting peer.

The value of the `bindings` key contains String describing the bindings, that is, how the data stream flows between operators. If an operator is bound to more than one operator downstream, it may forward data in a `round-robin` (default value) or `broadcast` fashion.


```
1
2 {
3   "topology": {
4     "id": "dht_topology",
5     "operators": [
6       {
7         "id": "producer",
8         "script": "dht_producer.js",
9         "sensors": ["temperature", "humidity"]
10      },
11      {
12        "id": "filter",
13        "script": "dht_filter.js"
14      },
15      {
16        "id": "consumer",
17        "script": "dht_consumer.js",
18        "browser": {
19          "path": "/consumer",
20          "only": true
21        }
22      }
23    ],
24    "bindings": [
25      {
26        "from": "producer",
27        "to": "filter"
28      },
29      {
30        "from": "filter",
31        "to": "consumer"
32      }
33    ]
34  }
35 }
```

Listing 3.6. Stream topology example.

Listing 3.6 shows the topology description file for the previously defined topology. After defining a string identifier for the topology, we identify an array for the operators. Each operator is described in an object containing the `id` and the `script` fields as mandatory values, all the other fields we left them with the default value. The producer object also specifies a list of sensors that needs to be available on the hosting machine to run the operator, in this case the temperature and the humidity sensors. The filter object doesn't have constraints,

while the consumer specifies `"only": true`, which forces its execution on a Web browser on the path `/consumer`.

The subsequent array describes the list of bindings for the topology. We connect the `producer` to the `filter` and the `filter` to the `consumer`.

The topology can be saved in a `.js` file and executed in the WLS runtime using the `exec` command. The runtime takes the input JSON file, analyses the requirements (i.e., if there are Web browsers available, or peers that have access to temperature and humidity sensors), and if all the requirements are met, it deploys the topology and starts it.

3.5 RESTful API

We implemented a RESTful API to deploy and control topologies. The API offers the same flexibility as the CLI, giving the possibility to the user to build applications on top of WLS. The implementation of such API comes naturally as Web server peers are basically Node.js servers which are able to serve HTTP requests. The API is in fact available on each Web server peer running WLS, accessible to the port specified when running the framework. The RESTful API as well as the graphical user interface were implemented by Mattia Candeloro's in his Master Thesis [Can14].

3.5.1 Resources

The resources exposed by the API represent the most important concepts used to manage the deployment and the execution of streaming topologies in WLS. We introduce the URI template and the informal semantics and hypermedia relationships between the resources. The API hypermedia graph is summarized in Figure 3.5, showing the hyperlink connections between representations returned by performing `GET` requests on the corresponding resource.

/

The peer root resource provides hyperlinks to the other top-level resources so that hypermedia can be used to dynamically discover what feature each of the WLS peer can offer.

/peers

This resource represents the collection of peers known by a peer and allows peers to discover and refer to each other.

/peers/:pid

This resource represents the information about a peer that is known by the peer

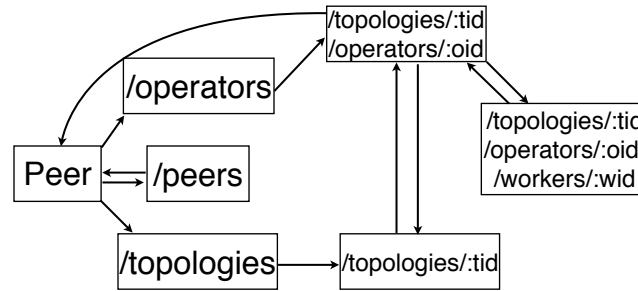


Figure 3.5. Hypermedia navigation map, showing the resources that can be discovered from each GET request.

from which it is retrieved. It also includes the hyperlink to the root of the REST API of the peer.

/operators

The operator resource collection represents the set of all operators (from all topologies) deployed on the peer.

/topologies

The topology resource collection represents the set of topologies known by the peer.

/topologies/:tid

A specific topology represents how operators are interconnected to form a data stream. It is used to deploy a new topology into the system as well as to dynamically modify it and control its state of execution. A topology representation contains hyperlinks to its operators, which are addressed as sub-resources for namespacing purposes.

/topologies/:tid/operators

The operators collection of a topology.

/topologies/:tid/operators/:oid

Each operator resource represents the individual processing step within a topology and contains the actual script to be executed. It also contains hyperlinks to connected operators that allow to follow the topology.

/topologies/:tid/operators/:oid/workers

Each operator resource allows to manage its worker collection (start, stop, migrate workers).

/topologies/:tid/operators/:oid/workers/:wid

The worker resource represents the execution state of an individual worker thread and allows the Controller to retrieve monitoring information about its performance.

3.5.2 Uniform Interface

For each of the previously introduced resources, we specify the semantics of applying one of the HTTP methods to it. If a method is not mentioned, a default 405 Method not allowed response can be expected.

Root

GET /

Retrieves a list of hyperlinks to the contents of the peer. There are three main links: `/peers`, `/topologies`, `/operators`. Additionally, some summary statistics about the peer performance are included.

Resource Management and Peer Discovery

GET /peers

Retrieves the collection of peers known to the peer inquired. The collection includes both relative hyperlinks to the local peer resource identifiers (`/peers/:pid`) as well as to the absolute hyperlinks to the REST API of the known peers (`http://ip:port/`). Finding a peer listed in the collection does not mean that the peer has established an actual connection to it for the purpose of streaming data, but only that the peer is known. If a connection has been established, the list contains the last CPU usage value seen on that peer.

POST /peers

A POST request on the `/peers` path with a payload referencing the address (IP:port) of the peer informs the receiver that a new peer exists on the network. The receiver stores the peer data in the collection and returns the updated list of known peers. This turns out to be useful when connecting a new resource to a root peer.

GET /peers/:pid

Retrieves the state of the peer and a hyperlink to its REST API.

DELETE /peers/:pid

Used to remove a peer with id `pid` from the list of known peers.

DELETE /peers/:IP:port

Used to remove the peer with address `IP:port` from the list of known peers.

Topology Management

GET /topologies

Retrieves the list of topologies started from the inquired peer with hyperlinks to their resource identifier in the form of `/topologies/:tid`.

GET /topologies/:tid

Retrieves the current execution state of the topology with id `tid`. The result shows the topology specifying which operators are running where and which script are they running. Hyperlinks to each operator are also included.

DELETE /topologies/:tid

This method call shuts down a topology.

POST /topologies

This method is used to create a new topology. The payload represents the structure of the topology to be implemented.

PUT /topologies/:tid

This method is used to create a new topology and associate it with the given identifier. The payload represents the the structure of the topology to be implemented.

Operator Configuration

GET /operators

Retrieves the list of all operators deployed on this peer with hyperlinks to their resource identifier.

GET /topologies/:tid/operators

Retrieves the list of operators running on this peer for topology with id `tid`.

GET /topologies/:tid/operators/:oid

Retrieves the representation of the operator with id `oid`. The representation includes the list of workers running (with hyperlinks to contact them), the script that they are running, the connections they have, as well as a hyperlink back to the topology it is part of and information about the overall performance (for example, the request/response rate or the CPU usage aggregated across all of its workers).

PUT /topologies/:tid/operators/:oid

Performing the request with a payload carrying a script and the bindings creates an operator named `oid` for the topology `tid`. This operation is not supported by `POST` as the name of the operator has to be know a priori in order to perform the bindings described in the topology. Workers created in this operator will run the script and performs the connections specified in the bindings. If the operator

identifier already exists, it is updated with the new information. This requires to stop the workers, update the script and the stream connections and then start the workers with the new script.

PATCH /topologies/:tid/operators/:oid/script

This request with a payload linking a new script updates at runtime the current script workers are running with a new version of it, without modifying the connections they have.

PATCH /topologies/:tid/operators/:oid/bindings

This request with a payload referencing new connections updates the bindings. In this case the overall topology is modified at runtime.

DELETE /topologies/:tid/operators/:oid

Stops and removes the operator with id `oid`.

Worker Configuration

POST /topologies/:tid/operators/:oid/workers

The request creates a new worker, the payload is not necessary as the operator already has all the information for its creation (that is, script to be run and connections to make).

GET /topologies/:tid/operators/:oid/workers

Retrieves the list of workers running on the operator with id `oid`. The list contains hyperlinks to contact every worker.

GET

/topologies/:tid/operators/:oid/workers/:wid

Retrieves the status of the worker with id `wid`. The result includes uptime, and information about the worker performance (for example, request/response ratio, or the throughput).

DELETE

/topologies/:tid/operators/:oid/workers/:wid

Deletes the worker with id `wid` from the operator by stopping it and removing its connections and deleting it.

POST /topologies/:tid/operators/:oid/browsers

Used to create a worker for operator with id `oid` on a browser. Returns a Web page and a script to be run. It only works if browser flag is specified in the topology description.

3.5.3 Representations

Operators

A collection of operators is either returned when performing a GET request on `/operators` or on `/topologies/example/operators`.

```
{
  "operators" : [
    {
      "topology" : "example",
      "id" : "a",
      "workers" : [...],
      "CPU usage" : "50%",
      "href" : "/topologies/example/operators/a",
      "peer" : "http://IP:port/",
      "replicas" : [
        "http://IP2:port2/topologies/example/operators/a"
      ]
    }
  ]
}
```

Listing 3.7. Example collection of operators.

Listing 3.7 shows an example collection of one operator. The JSON format is custom, as it is less verbose. Through content negotiation it is possible to retrieve different kind of representations. The representation is similar to the one we use to describe a topology: the array contains objects representing the operators, defining the id of the operator, the hyperlink to contact it and the name of the topology it is part of. A link back to the peer on which the operator is deployed is also provided. This allows to conveniently aggregate the operator configuration of multiple peers. Moreover, if the operator had to be replicated on other peers, due to overloading, an array with direct hyperlinks to the replicas is provided.

The topology description file, shown in Listing 3.5, can be fed through a PUT `/topologies/:tid` request. The operator IDs may be given by the user and are used to create the corresponding resource identifiers.

Operator Configuration

When executing a PATCH request to patch an operator, a JSON payload is sent containing the description of the change to apply. There are two kinds of patches: to modify which script is associated with the operator on the `.../script` sub-

resource or a modification of the bindings (that is, the topology) at runtime on `.../bindings`.

```
{
  "bindings" : {
    "from" : "/topologies/example/operators/c",
    "to" : "/topologies/example/operators/a"
  }
}
```

Listing 3.8. Updating the bindings of operator `b` at runtime.

Imagine a linear topology $a \rightarrow b \rightarrow c$. Listing 3.8 shows the JSON sent to update the bindings of operator `b`, reversing the flow of the topology. The object contains a `from` key and a `to` key whose values are different from the previous binding of the workers. Note that in this case, we can use the URI of the operators as yet another alias. It is mandatory to define at least one of the two in order to update a binding (the one not defined remains unmodified). If the operator is supposed to become a consumer, then it is sufficient to leave empty the string in the `to` key. Likewise, vice versa for operators that should be moved at the beginning of the topology (empty `from`). A similar JSON needs to be sent to the other operators `a` and `c` to update their connections as well.

Worker state monitoring

Listing 3.9 shows the result of a GET request on the `/topologies/example/operators/a/workers/0` path. It contains the id of the worker, information about how to contact it again and a hyperlink to its operator. Performance information includes how long has it been up, the total number of messages that have been processed since it was started, as well as its request/response ratio for the last second. The latter is very important for the controller in order to detect whether the operator is a bottleneck in the topology.

```
{
  "worker" :
  {
    "id" : "0",
    "href" : "/topologies/example/operators/a/workers/0",
    "operator" : "/topologies/example/operators/a",
    "uptime" : "3600",
    "messages" : 42,
    "req-res-ratio" : 1.5
  }
}
```

Listing 3.9. Worker state returned as JSON.

3.6 Graphical User Interface

Based on the RESTful API we built a Web Graphical User Interface (GUI) that shows in real time the running topology as a directed graph. Peers are shown as coloured overlays on top of the operators, which in turn contain the workers they are running. The GUI is able to do so by polling in real time the status of the topology from the RESTful API.

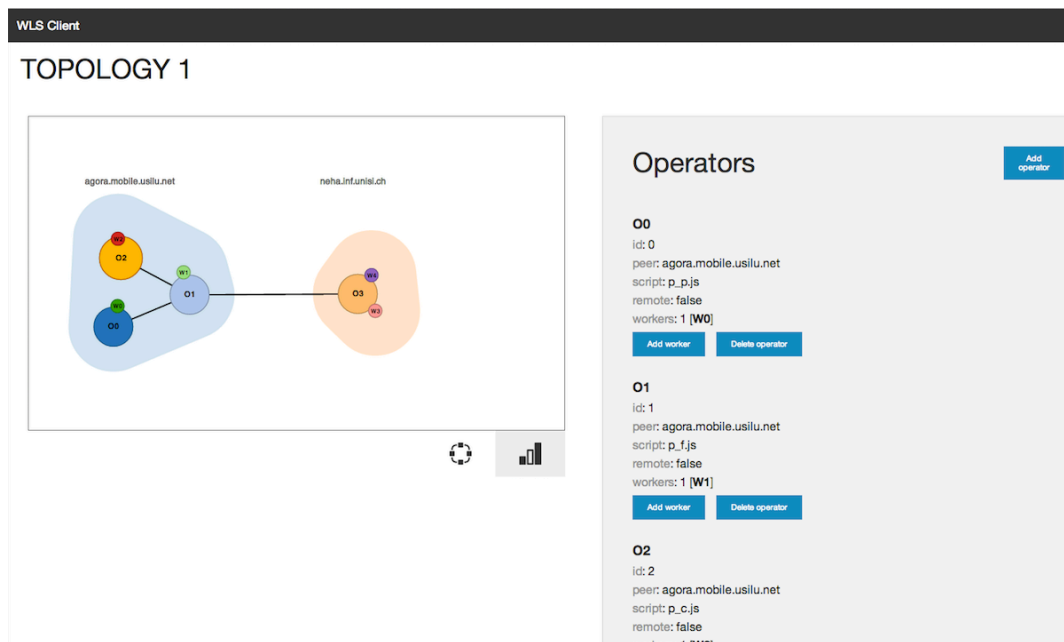


Figure 3.6. Web-based Graphical User Interface.

Figure 3.6 shows the GUI. The left part displays the information regarding the topology pulled from the RESTful API and fed to a D3⁴ visualisation. This visualisation gives the user, through keyboard shortcuts and mouse clicks, the ability to modify the configuration of the operators (by, for example, adding and removing workers) and of the topology (by adding and removing operators). On the right-hand side of the picture we show a textual representation of the information regarding currently running operators. The GUI offers the same capabilities of the command-line interface in a graphical form.

⁴<https://d3js.org/>

3.7 Use Cases

We now show use case scenarios that define some of the most common interactions with WLS from the developer's perspective. We show how they are carried out both with the CLI and with the REST API.

3.7.1 New Peer Joins the Network

When the developer wants to add a peer *P* to the network of devices connected in WLS, he/she needs to give *P* the IP address or the URL of the root peer in the WLS network as a command line argument. Peer *P* contacts another peer through either internal RPC communication or through a `POST` request on `/peers`. The payload of the request in both cases references the address of *P*. The root peer updates its list of known peers. The list not only features the address of known peers, but also their available sensors, and CPU usage if known. This list is used by the runtime whenever a topology has to be run, or when a migration by the control infrastructure is triggered.

3.7.2 Setting up a Topology with a Topology Description File

To setup a topology users have to execute a `exec` command followed by the topology description file in the CLI, or to perform a `PUT` request with a payload that contains the description of the topology which includes the links to download the scripts to run for each operator. The peer receiving the topology description fetches the scripts and deploys them on the most suited peers (more on this in Chapter 4). Peers receive the setup through either RPC or a `PUT` requests on `/topologies/example/operators/:oID` with a payload referencing the script and the connections to be performed. The level of parallelism with which the operators will be started (that is, the number of workers) is defined by the user on the topology description file.

3.7.3 Using Web Browsers to run Operators

Web browsers can be used to offload (part of) operator executions. Browsers have to connect to a URL to be part of a WLS topology, either the general URL `/remote` to be pooled as idle resources or a specific one to immediately be part of a specific operator execution. The hosting peer registers the browser as an available peer resource where computation can be offloaded, and sends the appropriate scripts to run as a reply to the request. On the one hand, if the Web

browser queried the general URL, a generic page will be shown. Whenever an operator needs more computing resource, or a new topology needs Web browser execution, the connected Web browser will receive a script and execute it automatically. On the other hand, if the Web browser connected to a specific URL, the hosting peer will immediately send the execution script of the interested operator to the browser, which can immediately start the computation. This will happen even if no more resources are needed to execute that operator, as it's the user that directly requested – through the URL – to be able to run that specific streaming operator on its machine. The connected machine will also be used to parallelise other operators, if needed.

3.7.4 Perform a new Binding

Performing a new binding at runtime can be done through the CLI executing a `bind` command specifying the operator from where the stream should go, to where it has to arrive. This can be done by the REST API as well by performing a `PATCH` request on `.../operators/:oID/bindings` to the operator with a payload specifying the new connections. The operator automatically updates its connections and redirects the traffic of its workers to the newly connected operator with the connections received through the request, thus changing the topology at runtime.

3.7.5 Load Balancing through the REST API

In general, load balancing is performed by the controller, which calculates the efficiency of the operator by accessing the state of the workers and takes decisions on the amount of workers to be added or removed from the operator. We introduce the controller in Chapter 5. Load balancing can also be done manually by the user through the CLI or through the REST API. Users may also create their own control infrastructure through the REST API.

With the received data, and by performing a `GET` request on the list of Workers `.../operators/slow_operator/workers`, a control infrastructure has the ability to compute the efficiency of the operator and detect if there are bottlenecks. If that is the case, the control infrastructure may want to parallelise more by performing a `POST` request on `.../operators/slow_operator/workers`, effectively increasing the number of running workers. If the workload decreases, the infrastructure may want to perform a `DELETE` request on the corresponding worker resources to shut them down and free some space on the peer hosting the operator execution.

If parallelisation on the peer results to be impossible (i.e., busy CPU), it is possible to exploit another available peer. This can be achieved by running a new instance of the bottleneck operator on a new peer (through `run` CLI command or `PUT` request on `/topologies/:tID/operators/:oID` and then performing the appropriate bindings), or by migrating the operator on a more powerful peer.

Chapter 4

The Web Liquid Streams Runtime

In Chapter 3 we introduced the WLS framework, showing how users can set up topologies and modify them at runtime depending on their needs. This Chapter introduces the WLS runtime infrastructure.

WLS deployment mainly targets Web servers, Web browsers, and microcontrollers and single-board PCs. To face the intrinsic differences in these environments, we developed three different implementations of the WLS runtime. A Web server implementation, a Web browser implementation, and a minified version of the Web server for some microcontrollers. Some single-board PCs and microcontrollers are able to interpret Node.js, thus we can reuse the Web server implementation. Nonetheless, some of the microcontrollers are too small to host the entire WLS framework, for that reason we created a minified version of it, which only offers a single communication channel and no control infrastructure.

Figure 4.1 overviews the WLS runtime running on the three different implementations. These WLS implementations cooperate as a single entity, abstracting the underlying hardware and offering its computational power through the WLS primitives. The only constraint posed by the runtime is to start a topology on a server deployment of WLS. In the remainder of this Chapter, whenever we refer to "server" deployment, we also include the microcontrollers and single-board PCs able to run Node.js, if not stated differently.

We start by identifying the communication layers, we then introduce the Web server and Web browser peer infrastructures and the implementation of the streaming operators. We show how topologies are implemented, and finally we introduce how we developed stateful operators.

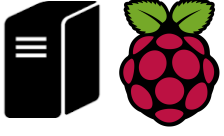


| WLS (Server) | WLS (Browser) | WLS (Minified) | |
|---|---|--|----------|
| Node.js | Web Browser | Tessel (LUA VM) | Software |
|  |  |  | Hardware |

Figure 4.1. WLS Runtime and the three different deployment implementations.

4.1 The WLS Communication Layers

There are two main communication layers in WLS. One is dedicated to the command passing and execution (command layer), the other represents the streaming channels that are created and destroyed during topology execution (stream layer).

4.1.1 Command Layer

Peers in WLS need the ability to form a network of connected, heterogeneous devices. In every WLS network of connected device, there is one WLS peer that acts as the root server (introduced in Section 3.2), and one or more client peers that connect to the root in order to offer their resources to run a streaming topology. When a root peer receives a topology as input from the user, it may need to contact known peers in order to deploy streaming operator execution on them. Similarly, when a user – or the control infrastructure – needs to modify a running topology, the root needs the ability to contact other peers in order to issue commands remotely.

We decided to implement the command layer using remote procedure call (RPC). RPC not only helps structuring a persistent network of connected peers, but it also offer bidirectional channels and detects disconnections as soon as possible. It also comes as the more natural way to implement the command layer, as we interact with remote processes as if they were running locally. We implemented RPC with the DNode¹ NPM library, which is an asynchronous RPC library that includes a Web browser distribution perfectly suited for our needs. DNode is built using WebSocket as the communication channel.

¹<https://github.com/substack/dnode>

Traditional RPC implementations have clients call procedures on the server and synchronously wait for the result of the execution. In our case we make use of reverse asynchronous RPC. The root peer executes commands (calls procedures) on the connected clients, which execute a callback when the execution is terminated. On the one hand, from an implementation point of view it comes more natural to be able to call functions on connected peers through method calls that abstract the communication layer – which is one of the purposes of RPC. On the other hand we make use of callbacks instead of waiting for synchronous results, following the JavaScript programming model.

Peer Discovery

Before setting up a distributed topology, the system should be able to discover other peers and store their addresses and general information. In our model, users can set up peers to either listen to incoming connections (root peers) or to connect to other peers (client peers).

If a root peer p is running an instance of WLS, a client peer q that wants to connect to p needs to run the WLS instance passing the address of the first peer as a command line argument. The WLS infrastructure will take the address and connect to p . Peer p will store q 's details in a list of known peers, which is inquired whenever the runtime needs to deploy a new topology or needs to migrate existing operators.

In case no address is passed when running a WLS instance, q will start a server instance and will be listening to incoming connections (becoming an RPC server instance). Web browser peers can connect by accessing the `/remote` URL on the root server running WLS to access the idle page. The idle page holds the connection with the Web server peer through WebSocket. Web browser peers that connect to the idle page effectively open a WebSocket channel (through DNode) with the server and wait for an operator to be deployed on them. Alternatively, they can directly access paths related to already running operators, starting a new instance of a given operator with the appropriate bindings automatically (see Chapter 3).

RPC Command Execution

We use the described DNode infrastructure to send commands to other peers from the root peer. They map the CLI and RESTful API high level interfaces and execute the commands that are received from either the user or the system and

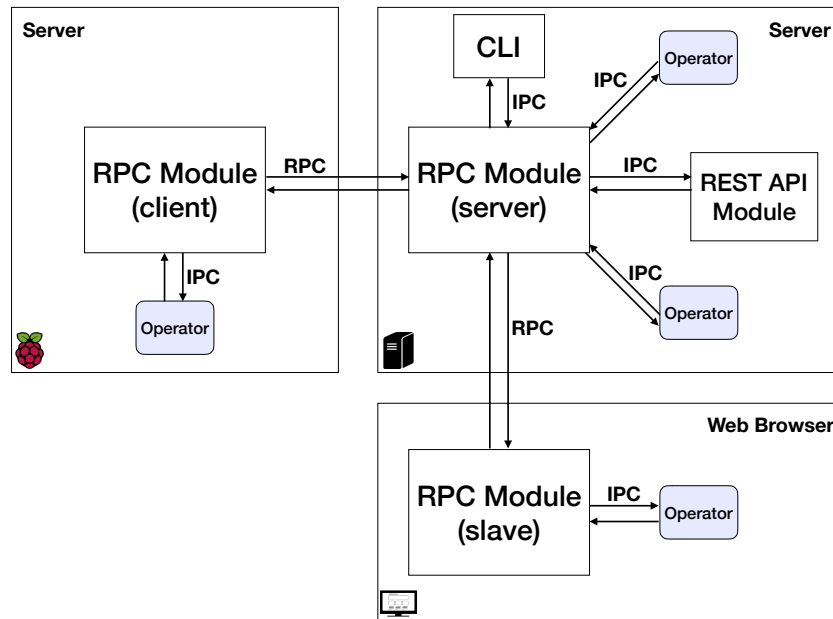


Figure 4.2. RPC and IPC Interactions among distributed and heterogeneous peers.

execute actions on a topology (e.g., run it, stop it, add a new operator/worker, remove an operator/worker, etc.).

Figure 4.2 shows RPC and inter-process communications (IPC) interactions among peers of different nature (we omitted the modules that are not related to the command execution). The peers p and q both run operators, and are connected through the RPC module. One acts as the server, while the other acts as a client. The Figure also shows a Web browser peer which is connected to the server peer and is running a single operator. Figure 4.3 shows in more details an example RPC interaction between two servers, one running the RPC module as a client, while the other running the RPC module as a server. The server RPC sends commands to the client, which performs the procedure and returns by executing a callback on the server.

We now show the low-level commands that can be called from the RPC infrastructure. We map the CLI/REST API introduced in the previous Chapter to the more low-level RPC call.

- **run_operator(String oid, String script, Int workers_number, Array wids, String automatic, String alias, Function cb)** represents the low-level implementation of the function call `run` executed in the CLI. While in the CLI we only pass as mandatory arguments the script that has to be run, in

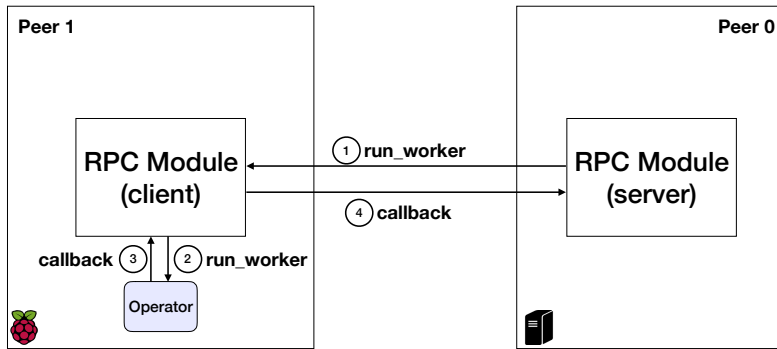


Figure 4.3. Closeup of the RPC interaction among two RPC modules, one acting as a client while the other acting as a server.

the remote procedure call we have to specify the operator id to be created (generated by the server RPC), the script to be run, the number of workers to start the operator with, their workerIDs, a string specifying if the operator has to be checked by the controller, and a callback function executed whenever the execution ends.

- **bind_operator(String from_oid, String to_oid, Array to_peers, Object aliases, Function cb)** is the low-level implementation of the `bind` CLI command. This procedure is called on the peer which hosts the sender operator (`from_oid`) It takes as input parameters the operator id from where the connection should start to where it should end, a list of peers where the `to_oid` is hosted (may be more than one in case of an operator running on multiple peers), the aliases of both the operators as an object, and a callback to be called at the end of the execution.
- **run_worker(String oid, String wid, Function cb)** is a low-level function call which runs a new worker in a given operator with a given workerID. The call is executed on (one of) the peers executing the operator with ID `oid` and starts a new worker with id `wid`. The callback `cb` is called at the end of the execution.
- **unbind_operator(String from_oid, String to_oid, Function cb)** supports the `unbind` command. Much like the high-level command, it only takes the endpoints of the binding that has to be removed. The callback function is executed when the unbind is complete.

- **kill_operator(String oid, Function cb)** is used to remove a single operator from the topology. The peer hosting it is contacted, passing the operator ID, and it will perform the unbind and disposal of the operator.
- **kill_worker(String wid, Function cb)** is used to remove a single worker. It is enough to specify the ID of the worker, when contacting the peer hosting it.

These commands represent the low-level WLS RPC interface. Other commands that can be issued through the CLI, the REST API, or the controller are a combination of such commands. For example, `migrate` include a series of remote procedure calls executed by the runtime using these functions (`run_operator`, `bind_operator`, `unbind_operator`, `kill_operator`).

4.1.2 Stream Layer

The stream communication layer is a communication channel that has to be persistent, fast, and handle a high number of messages in one direction. To handle heterogeneity in the system, we implemented different communication channels.

For server-to-server communication we aimed for a broker-less approach, looking for a lightweight low latency/high throughput message queue. We decided to opt for the NPM library ZeroMQ [Hin10] which offered such characteristics and was available on NPM and thus could be implemented in Node.js applications.

Browser-to-server and server-to-browser communication can not rely on ZeroMQ for the lack of such library on the Web browser. We took into account AJAX, which works for browser-to-server interaction but it's difficult to achieve server-to-browser interaction, besides it lacks of persistency and is generally unfeasible for a stream channel. We decided then to pick the WebSocket protocol for this type of channel. While being a primitive and requiring effort to build infrastructures, WebSocket offers flexibility and persistency (and a full-duplex channel), avoiding more complex communication infrastructures (i.e., Comet).

Finally, for the browser-to-browser communication, WebRTC DataChannels (introduced in Chapter 2) are the state-of-the-art solution for browser-to-browser streaming communication (without relying on a Web server). They are established through an ad-hoc component called signalling server (introduced later in this Chapter) which manages the WebRTC Interactive Connectivity Establishment (ICE) candidates.

For data streams running on the same machine, we rely on IPC in both cases. Table 4.1 shows the communication channels for the stream layer for each deployment configuration of the streaming operators.

The different communication infrastructure is hidden from the user, which only task is to develop operators and describe how they have to be wired to the WLS infrastructure (by either using the CLI, the REST API, or the topology description file). It is the WLS runtime task to use the appropriate channels to physically connect the operators.

4.2 Peer Infrastructure

The peer infrastructure represents the WLS runtime on the hosting machine, and is in charge of handling operators and workers running on a device. When a peer initialises, it either instantiates the RPC module as a server and waits for incoming connections, or it instantiates it as a client and connects to a root peer. Either way, it executes commands to setup, modify, or stop a streaming topology.

The number of operators that can be hosted depends on the hardware capabilities of the peer. Users may force the deployment of an operator on a given peer or let the runtime decide where to deploy the operator by relying on the control infrastructure (described in the next Chapter).

4.2.1 Web Server Peer

RPC Module

The RPC module is the most important part of the peer. It receives commands from the CLI, the REST module, and the controller to issue commands to run and modify topologies. The main methods offered by the RPC module are shown in Section 4.1.1.

| Operator 1 | Operator 2 | Data Channel |
|-------------------|------------|--------------|
| Server | Server | ZeroMQ |
| Browser | Browser | WebRTC |
| Server | Browser | WebSocket |
| Browser | Server | WebSocket |
| Browser or Server | Self | IPC |

Table 4.1. Data channels for different deployments.

The RPC module in the root is in charge of dealing with the distributed nature of the topology in WLS, and deals with the differences in terms of communication channels when setting up bindings. The global controller makes use of the RPC module through hooks when decisions about the topology are taken, and commands have to be sent to client peers.

In client peers, this module receives the commands from the root and executes them. It is also used by local controllers in order to deal with bottlenecks by using hooks to increase (or decrease) the number of workers in the hosted operators. Callbacks are also set up in the RPC module to be used by the controller when one or more operators need to be parallelised across multiple peers. By firing these callbacks, the topology controller, through the root peer, receives the commands and distributes the execution of an operator.

Signalling Server

The peer infrastructure also sets up a signalling server module, used to perform bindings with WebRTC, and when Web pages connected to the peer need to perform bindings. Signalling is not defined by the `RTCPeerConnection` API, thus the signalling server has to be implemented by developers that want to make use of WebRTC. The easiest signalling server that could be created is a passive module that implements three main functionalities:

- **Discovery:** The clients connect to the same sever, the server knows who has connected and possibly who left. Since all the clients connect to a single point, the server knows all the clients that could potentially create a new `RTCPeerConnection`.
- **Communication:** The server must establish a communication channel with each client, in fact the server must be able to send data directly to each client and initiate their peer-to-peer binding.
- **Signalling:** The signalling depends directly on the communication channel used. Once the communication channel is chosen, the server must allow clients to exchange *offers* and *answers*.

The signalling server also implements the NAT traversal functionality. The NAT traversal functionality is implemented thanks to the **STUN** (Session Traversal Utilities for NAT) or **TURN** (Traversal Using Relays around NAT) protocols. The address of a peer is usually hidden by NATs, so an additional server is needed in order to discover its own public address, and thus create the binding [HHE15].

STUN servers are cheap structures used to find Web browsers' public addresses, while TURN is an extension of STUN that implements a fallback if the peer-to-peer connection fails, named **relay server** – servers that simulate the peer-to-peer topology with a client-server-client one.

Command Line Interface

The Command Line Interface (CLI) module is a small structure that interprets the commands issued by the user and issues RPC calls through the RPC module. We decided to implement a module instead of bridging the user commands directly to the RPC module to improve flexibility and achieve loose coupling. In this way both parts can evolve independently while maintaining a fixed communication interface.

REST API Module

The Web server peer also hosts the REST API module that bridges the REST calls to the RPC module infrastructure. We decided to create an ad-hoc module instead of implementing it directly in the RPC module for a matter of separation of concerns. We introduce stateful operators and the Redis module in greater details in Section 3.5.

Redis Module

The Redis module helps the implementation of stateful operators. It bridges the communication between the operators and the underlying Redis data structure store. The Redis module offers an API which operators can use to store and read data to and from the storage system. We introduce the choice of data storage and a more in-depth implementation of the module in Section 4.5.

Controller

The controller is an infrastructure that is tightly coupled with the RPC module and periodically checks on the topology and the operators running in the peer, we discuss the controller in Chapter 5. The controller issues commands through the RPC module modifying the elasticity of the operators locally (that is, the number of workers inside an operator) or globally (by running copies of operators on more than one peer) to face bottlenecks. The controller also checks on the integrity of the topology, and issues run and bind commands if one or more operators fail (i.e., when a peer fails or disconnects abruptly).

Proxy

The last important structure in the Web server peer infrastructure is the proxy. The proxy is a module used to handle streaming channels to Web browsers. It is used by the streaming operators to avoid creating multiple WebSocket connections towards the same Web browser. Since Web workers inside Web browsers are not able to instantiate WebSocket channels, we decided to open a single WebSocket channel from operators running on server peers to operators running on Web browser peers. The WebSocket channels are opened at peer level, not at operator level. Each server peer must have at most one WebSocket channel open per Web browser peer in the topology; the proxy module helps keeping track of the open channels and routes messages accordingly.

Architecture

Figure 4.4 shows the logical view of the server peer. The view only shows the logical connections of the command layer, and not the streaming channels. The RPC module deals with incoming requests (as RPC client) or executes commands received by the REST API, the CLI, or the controller (as RPC server). It communicates directly to the operators spawned in the local peer, and with the proxy component to setup browser-to-server and server-to-browser bindings. In this case, since it interacts with the REST API module and the CLI, the portrayed peer is an instance of a root peer, and the RPC module is a server instance of the module. The signalling server is in charge of the handshake between two Web browser peers in the process of establishing a WebRTC channel, while the Redis module directly interacts with the operator(s), offering state storage.

4.2.2 Minified Web Server Peer

While most of the single-board PCs and some microcontrollers are able to accommodate WLS and its dependencies, a subset of microcontrollers is not able to run WLS or make use of external libraries needed to run it. For this reason we decided to implement a minified version of the Web server peer, in order to make it possible for smaller hardware to work within WLS. This work was part of a Master Thesis [Bla15] by Virginie Blancs, which studied the implications and implemented such infrastructure for the Tessel² microcontroller.

The minified Web server peer is divided in two main parts: the WLS runtime and the operator code. Given the capabilities of the hardware where such im-

²<https://tessel.io/>

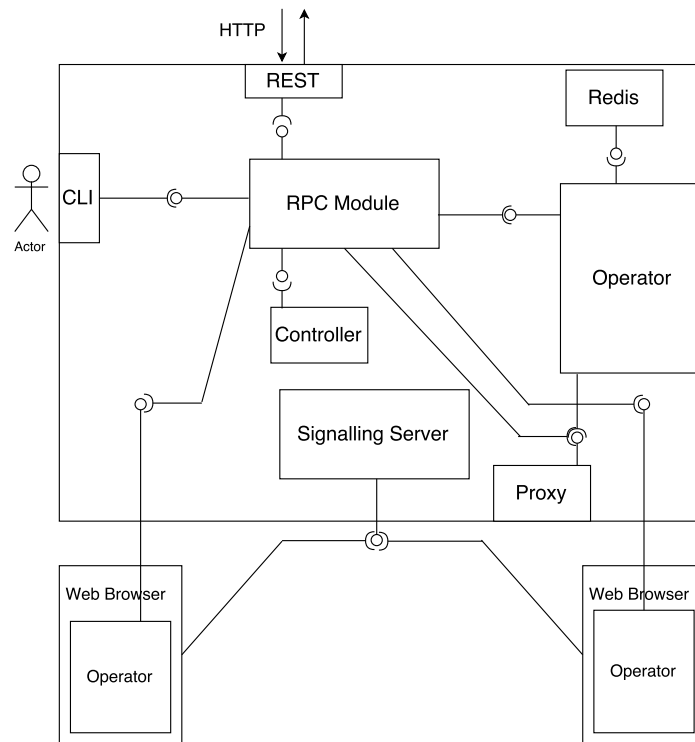


Figure 4.4. Logical view of the command layer of the Web server peer. Streaming channels are not shown.

plementation could run, we decided not to allow parallel execution of operators, and to fix the number of workers inside an operator to one. The runtime part implements WebSocket channels for both the command layer and the stream layer. At startup, the minified version connects to a WLS server instance as an idle microcontroller peer. The server peer, knowing the nature of the device, issues commands through the WebSocket channel instead of using the standard RPC infrastructure. The procedures implemented in the minified version of WLS are the most basic ones to setup and stop a streaming topology: run, bind, unbind, and stop an operator.

Figure 4.5 shows the structure of the minified Web server peer. The WLS runtime deals with the creation of the single operator and the single worker by receiving messages through WebSocket. When a streaming operator is running and a binding happened, the stream flows through a parallel WebSocket channel which connects directly to the WLS server peer proxy, which in turn routes the message to the right destination. By reusing the proxy component, we are able to hide the complexity of dealing with a different kind of hardware from the internal runtime of WLS. From the point of view of the Web server peer in fact,

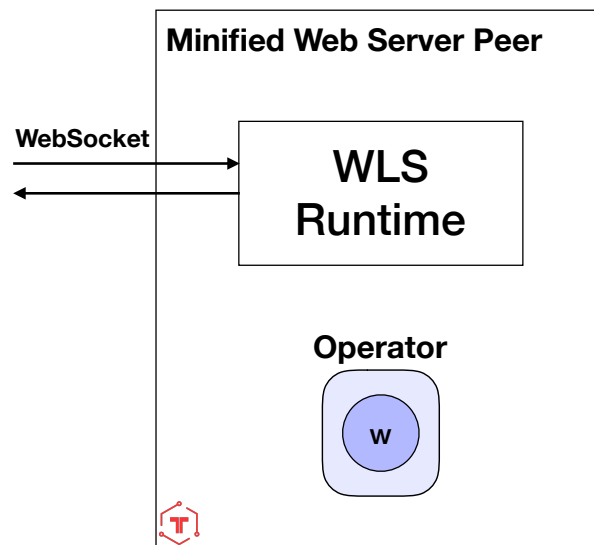


Figure 4.5. Minified Web server peer infrastructure.

the minified Web server peer is considered as a Web browser peer. The runtime just routes messages coming from ZeroMQ channels to a WebSocket channel as if it was communicating with a Web browser. From the Web browser peer, the connection to the minified version happens passing through a server and WebSocket, thus behaving like a Web server peer.

4.2.3 Web Browser Peer

RPC Module

The RPC module is used by the Web browser peer to connect to the Web server peer and receive commands from it. The commands are directly issued from the Web server peer.

HTML5 Actuator Display API

The HTML5 Actuator Display API module is in charge of dealing with the HTML and the visualisation of outputs (if any) of the operator. Through method calls such as `createHTML`, it is able to modify the output on screen and give vi-

sual feedback to the user. The module also includes the instantiation of graphic libraries.

HTML5 Sensors APIs

The HTML5 Sensors APIs are used by the operators and offer access to the underlying sensors that can be accessed from a Web browser (i.e., battery level, number of processors, etc.). The module also offers stateful operator functionalities, which call hooks in the RPC module to store data on Redis on the server peer. The module can be extended to support future utility infrastructure such as browser-related and platform-related information (available in the `navigator` Web browser object).

Controller

We implemented a control infrastructure in the Web browser peer to deal with bottlenecks and failures at browser level. While the behaviour of the controller is similar to the one running on the Web server peer, the implementation is slightly different. Chapter 5 introduces in greater details the functionalities and the implementation differences of the two controllers.

Architecture

Figure 4.6 shows the architecture of the Web browser peer. The RPC module is connected to the Web server peer serving the page (RPC channel not shown) and offers an interface both to the local controller and the HTML5 Actuator Display API. The operator is connected to the latter, and the HTML5 Sensors APIs to both use functionalities related to the Web browser and make use of the graphical user interface, if needed. It is also connected to the signalling server to establish WebRTC connections with other Web browsers.

4.3 Operator Infrastructure

In the previous Section we analysed the communication infrastructure of the WLS runtime and the peer infrastructure. Now that the reader is familiar with how messages are exchanged within the runtime (be it a stream of messages or single commands to build, control, or stop a topology), we describe the operator internal infrastructure. Given the differences between the Web server environment

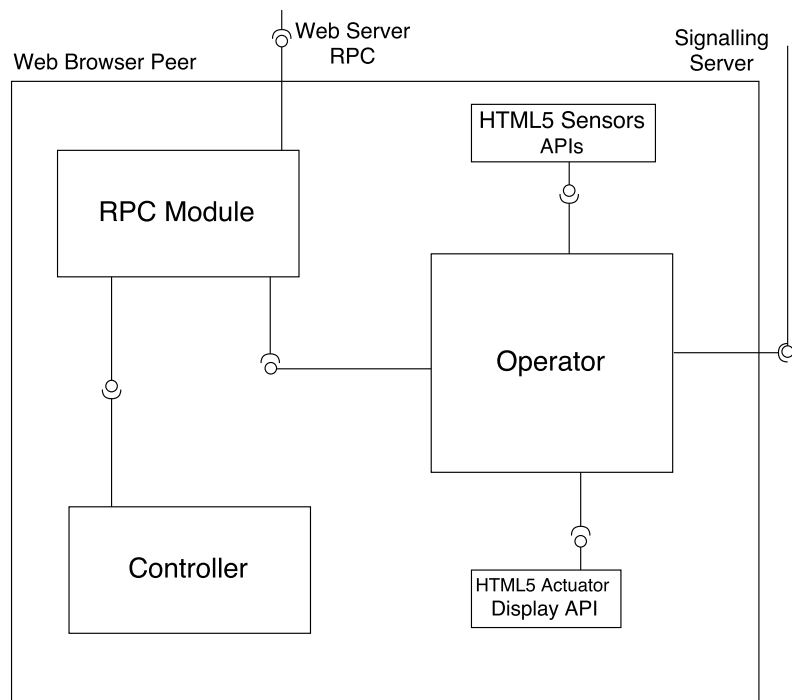


Figure 4.6. Logical view of a Web browser peer.

and the Web browser environment, we decided to build two different operator infrastructures.

4.3.1 Web Server Operator Pool

The Web server implementation of the operator pool receives commands through IPC by the RPC module and executes them on the appropriate operator. The pool contains objects representing the operators running on the peer. Each object contains all the data related to the operator, such as its operator ID (oID), the script it has to run, the list of workers currently running, its alias (human-readable unique identifier), a control flag, and a list of workers with their IDs (wIDs).

While the Web browser operator infrastructure takes care of multiplexing the messages to the underlying pool of workers (shown in Section 4.3.2), in the Web server operator the streaming channels are connected directly to the workers and not multiplexed by the operator pool. The operator pool communicates to the spawned children through `child.send(Object message)` where the message is a structured object specifying its scope (`command`) and the data needed.

```

1 {
2   "command" : {
3     "type" : "string",
4   },
5   "data" : {
6     "type" : "object",
7     "properties" : {
8       "oID" : {"type" : "string"},
9       "wID" : {"type" : "string"},
10      "target_oID" : {"type" : "string"},
11      "target_wID" : {"type" : "string"}
12    }
13  }
14 }

```

Listing 4.1. Worker message command schema.

Listing 4.1 shows the schema of the message, which is a JavaScript object wrapping the data to be sent to the worker and including the `command` field describing its scope. Not all of the fields are required, as they are strictly dependant to the type of command issued.

Table 4.2 shows the message scopes that are sent to the worker from the RPC module. The messages that can be sent to the workers cover the setup and the halting of the worker, as well as the bindings (to another Web server instance, or to a Web browser), and the data collection call from the controller.

```

1 {
2   "command" : "unbind",
3   "data" : {
4     "wID" : "from_wID",
5     "target_wID" : "to_wID",
6   }
7 }

```

Listing 4.2. Example unbind message sent by the Web server operator to a worker.

| Scope | Usage |
|--------------|--|
| setup | Sets up the worker infrastructure (i.e., wID, oID he is part of, etc.). |
| kill | Stops the worker. |
| bind | Binds the worker to another Web server worker. |
| unbind | Unbinds a worker (either from a Web server worker or from the proxy). |
| bind_remote | Binds the worker to a Web browser worker through the proxy infrastructure. |
| data_collect | Message type used by the controller to gather usage data from the worker. |

Table 4.2. Messages scopes within the RPC communication.

Listing 4.2 shows an example `unbind` command sent to a worker to remove an outgoing connection. It specifies the binding (through the workers IDs) that has to be removed.

4.3.2 Web Browser Operator Pool

Given the differences in the two deployment environments, the operator in the Web browser was implemented with a slightly different architecture. The development of such infrastructure has been the focus of Andrea Gallidabino's Master Thesis [Gal14].

The Web browser operator has the following purposes:

- Manage workers
- Receive messages from upstream
- Store messages that must be processed later (buffer)
- Send messages downstream

We created the corresponding components inside the operator to deal with those:

- Receiver
- Sender
- Message queue
- Workers handler

Workers Handler Module

The workers handler manages everything related to workers inside an operator. This logical structure contains references to all the workers running inside the operator, and offers an interface used by other components to access its functionalities.

It contains three main structures:

Workers array

The workers array is an array containing a reference to all the workers inside the operator. The workers array dynamically changes during the

lifespan of the operator and is used to pass the worker's references to outside the operator when needed. It implements the **add(Object worker)** and **delete(String wid)** functions.

Free workers stack

The free workers stack is a last-in-first-out data structure containing the reference of idle workers. Every time a worker is not initialising, or processing data (that is, it is idle), it is pushed in this structure. It implements the **push(Object worker)**, **pop()** and **delete(String wid)** functions.

Executing workers pool

The executing workers pool is an associative array containing the reference to all processing workers indexed by their own IDs. It implements the **add(String wid, Object worker)** and **delete(String wid)** functions.

These three structures are managed by a common interface called workers proxy. It also keeps track of all information related to them and of the script they are running inside the topology.

The workers proxy needs to communicate directly with the message queue, the sender, and the UI. The methods needed by the workers proxy are:

Sender

- `send(Object message)`

Message Queue

- `getMessage()`

UI

- `getDOM(String id, String command)`
- `setDOM(String id, String command, String value)`
- `addHTML(String id, String HTML)`
- `addScript(String script)`

The following list shows the methods offered by the workers proxy to the other components. These low-level methods help setting up a topology by managing the workers proxy and are used by the RPC module and the controller.

- **getScript()** Returns the path of the script associated to the operator.

- **hasFreeWorkers(Object message)** Returns `true` or `false`. If the operator has any free worker it will process the message and return `true`, otherwise it returns `false`.
- **getWorkersArray()** Returns an array of references of all the workers inside the operator.
- **getWorkersArrayLength()** Returns the number of workers inside an operator.
- **getWorkersUsage()** Returns an associative array with statistics related to the workers during a cycle: throughput, messages in, messages out number of messages executed.
- **terminateWorker(String wid)** Starts the procedure to terminate a worker.
- **newWorkers(Array wids)** Workers proxy spawns new workers given an array of worker IDs. The number of workers started corresponds to the length of the array of IDs.
- **getUpperBound()** Returns data related to the current worker number limitation of the operator.
- **producerSend(Object message)** The producer operator receives a special message from the UI, which is immediately sent to a worker.

Receiver Module

The receiver uses either PeerJS or WebSocket to establish both browser-to-browser and server-to-browser communication. It contains an array of opened incoming connections as well as statistics associated to them, such as latency, throughput, and the communication channel used.

The receiver communicates directly with the message queue and the Workers Proxy.

Message queue

- `push(Object message)`

Workers Proxy

- `hasFreeWorkers(Object message)`

Message Queue

The message queue is a *first-in-first-out* data structure which contains all the messages that can't be processed right away by the workers inside the operator. The following list shows the methods offered by the message queue to the other components.

- **push(Object message)** Inserts a *message* in the queue.
- **pop()** Returns the first message that was pushed into the queue and deletes it from the queue.
- **getQueueSize()** Returns the current size of the queue.

The message queue is a passive structure similar to a database, it just offers an interface to store and retrieve messages.

Sender

The sender module, similarly to the receiver, interacts with both the signalling server and Websocket in order to establish browser-to-browser and browser-to-server communication. It contains an array with all the opened connections and the statistics associated to them. The sender also implements the same events, but offers a slightly different interface. The following list shows the methods offered by the sender to the other components.

- **getThroughput()** Return the value of the current incoming throughput.
- **getLatencies()** Returns an array of latencies of all the established connections.
- **unbindWebRTC(String oid)** This procedure unbinds the operator from the downstream when connected through a WebRTC channel. If *ID* is null the receiver leaves the current WebRTC room, if *ID* is specified it only closes the current WebRTC connection with the corresponding operator.
- **unbindServer(String oid)** This procedure unbinds the operator from the downstream when connected through a WebSocket channel. *ID* must always be specified and corresponds to the ID of the operator that needs to unbound.
- **send(Object message)** This procedure sends a *message* downstream.

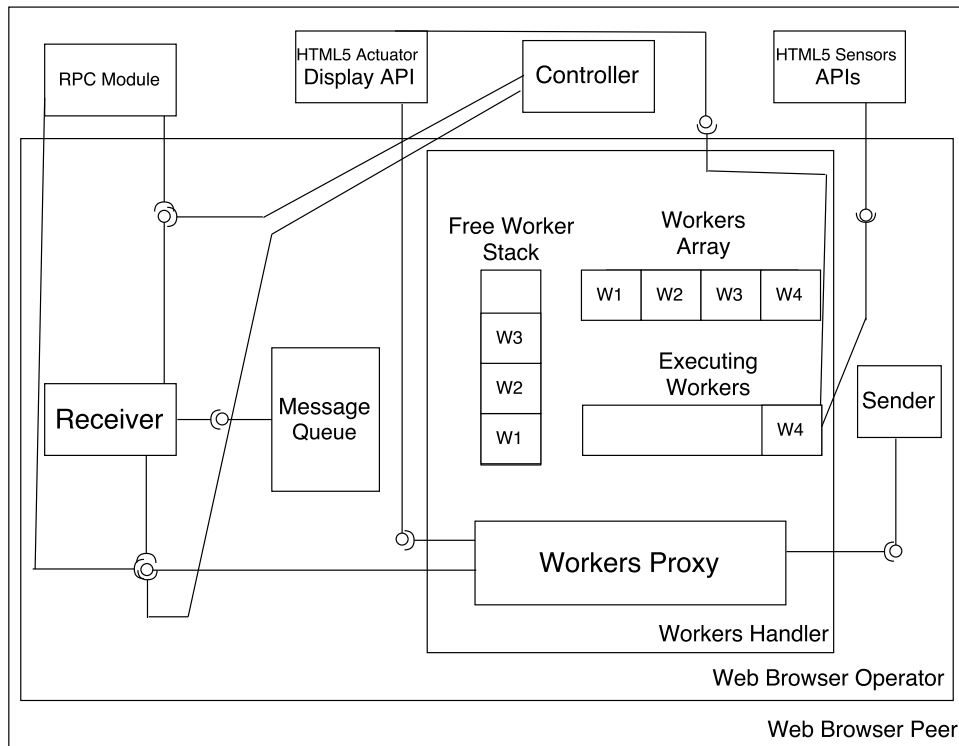


Figure 4.7. Logical view of the Web browser operator

Figure 4.7 shows the logical view of the Web browser operator. It's important to remember that a Web browser peer, much like a Web server peer, may contain any number of operators. In the pictures presented in this Section, we only show one for simplicity.

4.3.3 Web Server Worker Pool

A new worker is instantiated by spawning a new Node.js `child_process` executing the script submitted by the user. The script imports our library, which immediately instantiates the scaffolding for the communication on both the layers. Workers receive commands from the operator pool, and executes them.

The difference with respect to the Web browser workers is that everything related to the communication happens inside the worker. Web server operators do not need to deal with queues and messages dispatching. On the one hand, when communicating with another Web server operator, the Web server workers instantiate ZeroMQ channels and connect directly to the receiving workers. Queues are handled by the underlying ZeroMQ library, packets are received di-

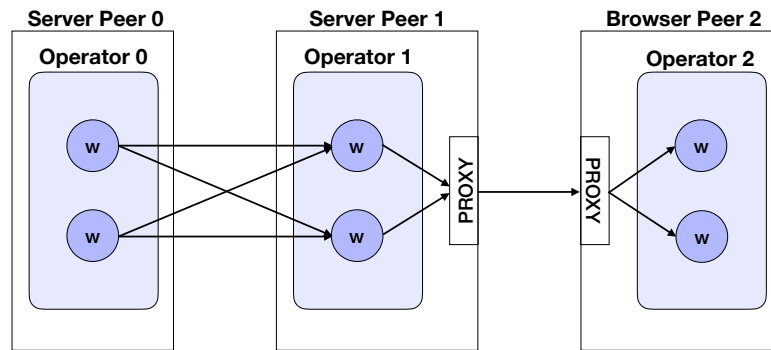


Figure 4.8. The worker communication infrastructure for server-to-server and server-to-browser communication.

rectly by the correct worker. On the other hand, if the connection is towards a Web browser operator, workers send messages to the proxy module, which in turn routes them on a WebSocket channel directly to the Web browser operator. The receiving operator is then in charge to dispatch it to a free worker, as shown in the previous Section. Receiving messages from a Web browser operator happens likewise: the message is received by the proxy module, which in turn routes it to the correct Web server worker. Messages are routed in a round-robin fashion to the workers in the receiving operator.

4.3.4 Web Browser Worker Pool

Workers in Web browser operators are instances of `WebWorkers`³. By definition, `WebWorkers` can't instantiate communication channels (either `WebRTC` or `WebSocket`), thus their only purpose is to receive packets from the message queue, execute the script they implement, and pass the result to the sender which forwards the result downstream.

Figure 4.8 shows the two different approaches for the server-to-server connection and the server-to-browser connection, which holds for the browser-to-server connection as well. The workers inside server peer 0 are directly connected to the workers running in server peer 1 through `ZeroMQ` channels. To send data downstream towards Web browser workers, the workers in server peer 1 make use of the proxy component which gathers the messages for the Web

³<https://www.w3.org/TR/workers/>

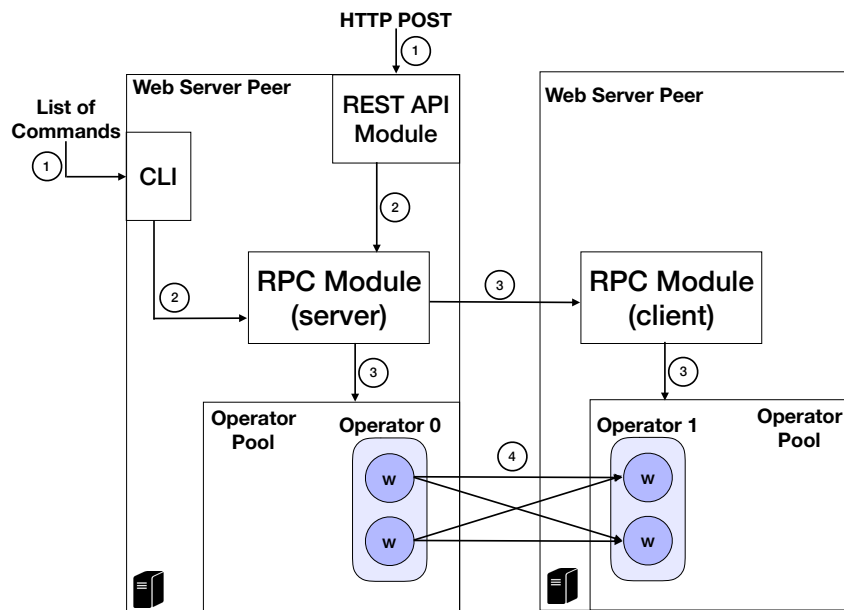


Figure 4.9. Step-by-step setup of a topology.

browsers and uses a single channel to send them. Whenever their are received downstream by the receiver, they are dispatched to WebWorkers to be executed.

4.4 Topology Creation and Dynamic Evolution

In the previous Sections we introduced the communication infrastructure as well as the basic building blocks to set up a topology. We now show how topologies are created by illustrating the steps taken by the CLI through the `exec` command, or from a `POST/PUT` request from the REST API. We then show how we perform changes in the topology at runtime (new bindings, migrations, etc.).

We mentioned that the `exec` command takes as input a file containing a topology. The file can be a list of commands to be executed (`.k` file) or a topology in a JSON format (`.js` file). In the first case, the runtime parses the file, splits it, and executes all the commands serially as some commands cannot be executed in parallel (i.e., binding two operators while they are being instantiated). In the second case, the JSON file is parsed, operators are taken from the list of operators and run with the given properties in an asynchronous way. Once the operator execution is done, the bindings are applied, again asynchronously, and the topology is started.

Following the same principle, the REST API upon receiving a `PUT` or a `POST` request on `/topology/:tid` and `/topologies` respectively, takes the payload of the request – which is a JSON representing the topology – and performs the appropriate calls to the RPC module to set the topology up. The order of the requests is the same as the `exec` execution, where first operators are run in an asynchronous way, and then the bindings are performed asynchronously as well. Figure 4.9 shows the step-by-step procedure in setting up a topology. After a topology has been submitted in the two previously described ways (1), either the REST or the CLI modules parse the topology and send commands to the RPC module (2), which in turn executes RPC calls and instantiates the operators and the bindings (3). Operators and workers are instantiated, and the bindings are performed (4).

Topologies can be retrieved through a combination of `ls` commands (described in Chapter 3) which can show a list of operators and a list of bindings. Alternatively, by performing a `GET` request on `/topologies/:tid` specifying the id of the topology, users are able to get the representation of the topology with ID `tid`. This can be useful when the user has built – by hand – a topology and wants to get its JSON representation, to save it in a file for further reuse.

Modifying a topology can be done at runtime by using commands such as `run`, `bind`, `unbind`, `kill` to respectively run, bind, unbind, and stop operators. This may impact the end result of the topology and change its semantics while the stream is running. If a binding is removed from a running operator and no more outgoing channels are available, the elements sent by the operator are not stored by the runtime and lost, thus the user should be careful and first bind new outgoing channels before removing old ones.

Operators may be patched through the `update_script` command or a `PATCH` request on the `/topologies/:tid/operators/:oid/script` to start a different script (or an updated version of it). Likewise the previous operations, this can be executed at runtime. The runtime stops the outgoing and incoming channels, restarts the operator with a different script by re-instantiating all the workers and creating new incoming and outgoing channels.

4.5 Stateful Operators

4.5.1 Overview

Stateful operators are streaming operators that need to maintain state throughout the execution of the topology. For example, they can be used to store home

environmental data that can be subsequently retrieved to highlight trends on energy consumption. The following list illustrates the features we took into account in the decision making process to pick a suitable data storage system.

- **Replication and Persistence** Operators can be distributed across several peers, and they may communicate with the root peer in order to perform reads and writes on the storage system. Distributing and replicating the storage system not only improves the performances by offering local reads, but also replicates the state (and makes it persistent) across a distributed set of connected peers that can be used on failover.
- **Availability** The storage system should be able to deal with internal faults without human intervention. In this way the stateful operators could be kept up and running even when a fault happens at database level.
- **Flexible Data Structure** Given the flexibility offered by WLS at topology and operator level (the possibility to rewire, add, remove, or update operators) a flexible NoSQL database could be the best solution. The horizontal scaling of NoSQL DBs also fits the distributed nature of WLS.

4.5.2 Redis

Among the available databases that offered the mentioned features, we decided to pick and implement Redis ⁴ in WLS. The decision making process is illustrated in Davide Nava's Master Thesis [Nav15].

Redis is a key-value [BCE⁺12] cache and storage system. It features a data structure that lets the users save strings, hashes, lists, sets, sorted sets, bitmaps, and offers the flexibility to store structured data while keeping the application as transparent as possible.

The features offered by Redis satisfy the requirements we defined for the WLS implementation.

- **Replication:** Redis implements a master/slave replication infrastructure. There must be at least one master instance in the system, which receives write commands, and there can be an arbitrary number of slaves connected, on which data is eventually replicated. This helps keeping data consistent since there is only one possible place where data is written, which then is eventually propagated to all slave instances. In WLS the

⁴<http://redis.io/>

master can be stored in the root peer, while the slaves run in the client peers connecting to the system.

- **Persistence:** when used as a cache, Redis operates in-memory and keeps all data in RAM, which is still the most expensive part of a server, so it is limited in quantity. Redis deals with the lack of RAM with an eviction policy that is able to free space for newer data. Data can also be dumped to disk in two ways: dumping database to disk with a background operation, or by writing on a log file all the operations performed. Failures can always be recovered by reading back the copy stored on disk, or by executing all the logged operations.
- **Flexible Data Structures:** Redis offers support for different data types. In a typical key-value system a key is associated with a simple string. The support of strings as data structure comes in handy when stringifying complex JavaScript objects. In Redis it is also possible to link a key to data structures like lists, sets, sorted sets and hashes. These data structures allow the system to shape data to fit software requirements.
- **Availability:** Redis supports replication: all client writes to a master instance are replicated on slaves, staying up to date. Replication improves availability by performing automatic failovers of masters to slaves replica through Redis Sentinel. Its work, paired with our controller, makes topologies more resistant to software faults. Whenever a master results not to be responsive anymore, a slave is elected as new master and it takes over receiving requests from clients.

The main features of Redis Sentinel are:

Monitoring: checks for availability on master and slave instances.

Notification: notifies via API an administrator (or another computer program) that something is not working on a particular instance.

Automatic Failover: promotes a slave instance to master whenever the latter is not responding or faulty.

Configuration provider: clients connect to a Sentinel to ask where a master instance is located. Sentinels are particularly useful in case of failover when other instances need to be notified about a new master.

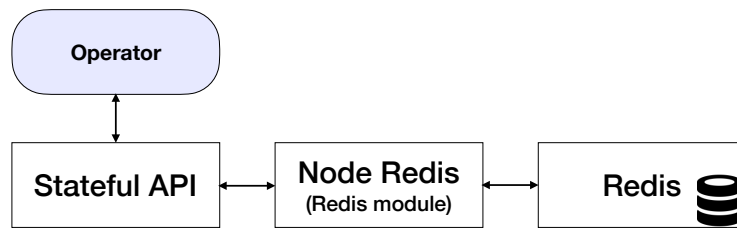


Figure 4.10. Redis communication abstraction with WLS

4.5.3 Implementation

We implemented Redis clusters in such a way that each Web server peer running WLS could have its own Redis instance running. This results in having a single master Redis instance (on the root Web server peer) while the slaves would be running on the client Web server peers.

Stateful operators are able to store and retrieve data on the Redis instance, while other Web server peers connected offer their Redis distribution as slaves for replication. The replication is done under the hood by Redis which is shipped in the WLS distribution, and is already configured with Sentinel to deal with faults. In this way operator migration is not affected by Redis: the operator is migrated and replicated data will be accessible on the new peer.

To make operators and workers able to interoperate with Redis, we designed a wrapper for the `redis` Node.js NPM library⁵. Calls to the `stateful API` described in Section 3.3 pass through the wrapper and are executed against the Redis instance, as shown in Figure 4.10.

The wrapper abstraction to communicate with Redis takes into consideration modularity, flexibility, and loose coupling of modules. Figure 4.11 shows the interactions of the WLS infrastructure running on a single Web server peer with a connection to a Web browser peer. Each operator makes use of the exposed API to perform state operations. Encapsulation makes the module transparent to WLS which only needs to acknowledge the exposed methods and does not need to be aware of changes in the module.

Stateful Web browser operations, and stateful operations coming from the minified WLS, are supported through the proxy module, which receives read/write calls and forward them to the local Redis instance.

⁵https://github.com/NodeRedis/node_redis

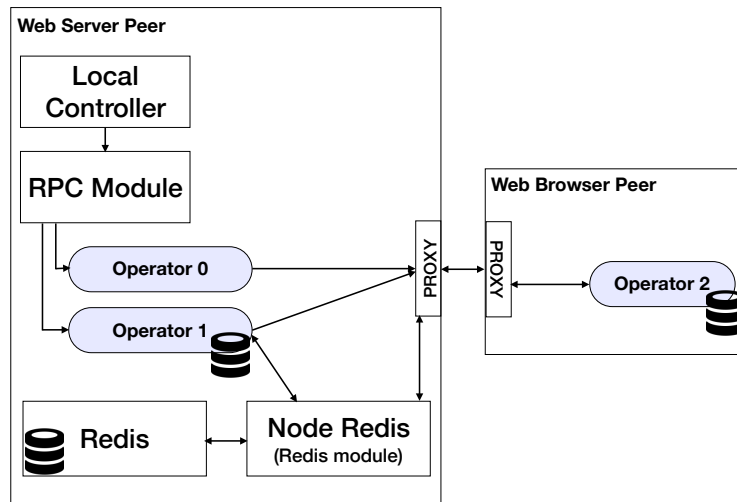


Figure 4.11. Interaction with the Redis module.

4.6 Summary

Web Liquid Streams has two main implementations to deal with the differences between Web browsers (sandboxed JavaScript event loop) and Web servers (Node.js). The main difference in the two implementation is the way JavaScript is interpreted in the Web browser. The limitations around the Web Worker concept – which make Web applications safe – had to be worked around in WLS. This impacted the implementation of the runtime as well as the controller. Another difference between the two implementation lies in the message handling: we do not deal with queues server-side, while on the Web browser we implemented a queue infrastructure.

A third implementation of WLS is a minified version for smaller microcontrollers (i.e., Tessel) which are able to interpret JavaScript but have less resources than a single-board PC. The implementation removes most of the components inside the WLS runtime and only keeps the most important ones to deal with the operator execution.

The next Chapter introduces the controller we implemented for WLS.

Chapter 5

The Control Infrastructure

The control infrastructure is part of the WLS runtime and is distributed across the connected peers. Its purpose is to check the peers connected for disconnections, overloads, and faults. It is composed by two main instances: a global controller which is in charge of checking the running topology and take actions upon peer crashes and disconnections, and a local controller which is deployed on the peers involved in the streaming topology, and is in charge of checking on locally deployed operators to balance the load and avoid bottlenecks. Local controllers are directly connected to the global controller and query it for operator migration and operator cloning when needed.

In this Chapter we introduce the controller use cases, tasks, and Web server and Web browser implementations.

5.1 Controller Use Cases

We first show WLS use cases scenarios that define some of the controller's tasks within the WLS runtime. We show how they are carried out from the user, the controller, and the REST API perspective.

5.1.1 Operator Migration

If a peer has to disconnect, a gentle shutdown message is sent to the root peer where the topology started. The root peer, with the help of the controller, takes care of migrating the execution of the leaving peer to another available peer with roughly the same (or more) computing power. The runtime first creates a new operator on the receiving machine, it binds it, and then proceeds to unbind the leaving operator, and finally stop it. The migration process can also be triggered

manually by the user through the `migrate` CLI command, or by the REST API by creating copies of the operators and binding them (`PUT` requests on the receiving peer), and remove the operators from the leaving peer (`DELETE` request on the leaving peer).

5.1.2 Peer Failure

When a gevent shutdown triggers, the runtime is able to deal with it by migrating the operators from the leaving peer. This is not the case when a peer abruptly disconnects (i.e., because of a failure). In this case, the controller is able to restore part of the lost computation on another available peer by using the last seen configuration of the failed peer. If no controller is running, it is the user's task to restore the lost operators and bindings using the aforementioned CLI commands or REST method calls.

5.1.3 Root Peer Failure

The root peer holds all the information regarding the topology started by it. If it fails, part of the control infrastructure as well as the command line interface become unavailable. The stream continues to run as long as no operators were deployed on the starting peer, but failures are not treated by the controller anymore, while bottlenecks are still dealt with on a peer level, but no distribution can be done anymore if a peer lacks resources. The current implementation of WLS does not implement recovery in such scenarios.

5.1.4 Lack of Resources for Parallelisation

A running topology may end up requiring more resources than the currently available ones. This issue impacts on the latency of the packets passing through the topology, which in the long run starts dropping packets, and eventually may crash some of the available resources. Users monitoring the topology can add resources at runtime to deal with the workload. As new resources are added, the runtime liquidly spreads the stream on the newly added machines, trying to deal with the bottlenecks.

5.1.5 Lack of Peers for Deployment

When a topology starts, either the user or the runtime have to deploy the streaming operators on the available resources. If one or more operators need specific

criteria to be deployed (i.e., Web browsers, sensors, etc.), and those criteria are not met, the topology is unable to start, throwing an error when running the commands. The user should connect the appropriate resources to WLS before starting the topology again.

The same issue may rise when operators only running on specific devices (i.e., Web browsers) are left without devices where they can be run when the topology is already up and running. This disrupts (parts of) the topology.

5.2 The Controller Tasks and Constraints

The control infrastructure deals with operators deployment, disconnections of peers, load fluctuations, and operator migrations. In this Chapter we target the joint work of both the controllers to deploy, migrate, and clone operators seamlessly across peers in order to face disconnections, but also to improve the overall performance of the topology in terms of latency and resource consumption by parallelising the execution when possible.

To do so, the controllers take into account a list of constraints, presented here in descending order of importance.

- **Hardware Dependencies**, the availability on the peer device of specific hardware sensors or actuators must be taken into consideration by the global controller as first-priority deployment constraint. An operator that makes use of a gyroscope sensor cannot be migrated from a smartphone built with such sensor to a Web server.

- **Battery**, whenever a peer has battery shortage, the controllers should be able to migrate the operator(s) running on such peer in order not to completely drain out the battery. At the same time, a migration operation should not be performed targeting a peer with a low battery level.

- **CPU**. The current CPU utilisation of the peer must leave room to deploy another operator. Since JavaScript is a single-threaded language, we use the number of CPU cores as an upper bound on the level of parallelism that a peer can deliver.

These constraints are used to select candidate peers (for either the deployment of a topology, or for a clone operation), which are then ranked according to additional criteria (introduced in 5.3.4), whose purpose is to ensure that the end-to-end latency of the stream is reduced, while minimising the overall resource utilisation.

All the constraints, with the only exception of the dependencies, can be relaxed. In fact, the global controller does not allow operators with a deployment

constraint to be migrated on peers that do not satisfy the constraint, but it accepts deploying on peers with, for example, low CPU availability as long as it is able to run the operator. It can also happen that the number of peers is lower than the number of operators. In that case one or more peers will host more than one operator each.

5.2.1 Automatic Deployment

By feeding a topology to the runtime, users of the system can deploy and execute their topologies on (a subset of) the available peers, which will be autonomously and transparently managed by WLS. Upon receiving a topology description or manual configuration commands, the system checks if the request can be satisfied with the currently available resources (i.e., sensor-requiring operators, Web-based operators). First of all, it will try to run the operators that have a fixed host destination specified by the user. Then, the runtime proceeds to query the global controller for available peers to run the remainder of the operators. For each operator, the global controller will check the constraints and assign a peer to host it, following the above mentioned constraints. Once all the operators are running, the RPC module takes care of the bindings, as illustrated in Chapter 4.

5.2.2 Load Balancing

While a Topology is running, operators exploit the parallelism of the underlying host processors in order to achieve better performance and solve bottlenecks. This is done by the local controller forking more processes and parallelising the execution of the operator. Each process receives part of the streamed data and executes the operator function. As illustrated in Section 4.3, on Web servers forked processes are Node.js processes which implement the operator script, while on Web browsers we make use of HTML5 Web Workers.

If the rate of the incoming data stream decreases, the operator automatically decreases the number of forked processes by detecting their idleness. This is implemented through an auto-adjusting timeout inside each worker. The timeout adjusts itself based on the number of messages received per second, and is fired whenever the worker has not received any message after a given amount of time. The implementation of such mechanism is illustrated in 5.3.

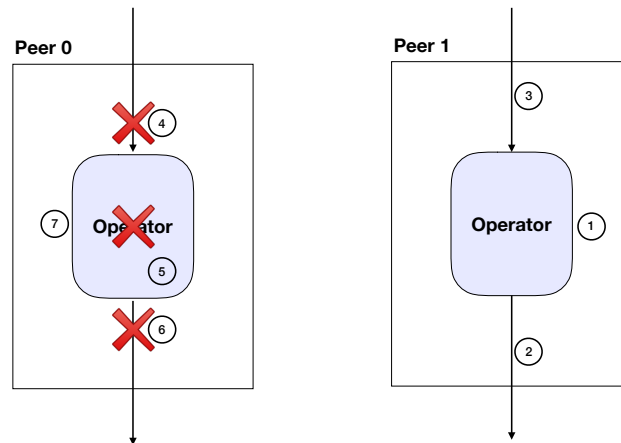


Figure 5.1. Step-by-step migration of an operator.

5.2.3 Operator Migration

The operator migration is an autonomous functionality that lets operators move freely from peer to peer in a transparent way without disrupting the data flow of the topology. There are two main reasons we implemented operator migration:

- **Gentle Peer Disconnection** Whenever a peer needs to disconnect, it starts a gentle disconnection procedure which informs the global controller. The global controller is then in charge of issuing a migration of the operators from the leaving peer to other available peers, respecting the constraints of the operators to be migrated.

- **Battery Saving** In case of operators running on portable or mobile devices, a migration command may be issued when the battery level drops under a certain threshold. In order to avoid an abrupt disconnection caused by a battery shortage, the local controller, which monitors the battery levels, contacts the global controller to start a migration procedure.

Operator migration can be manually triggered as well; users may be willing to move operators around by themselves (testing operators on different peers, moving the GUI from one host to another, etc.), they can issue a migration procedure directly from the CLI or the RESTful API specifying the operator to be migrated and the destination peer. If the constraints allow it, the operator is migrated by the runtime while the topology is running.

Figure 5.1 shows the step-by-step procedure that the runtime takes to migrate an operator. To execute a migration the runtime first creates a copy of the operator to be migrated on the target peer (1) and binds the associated input

and output channels ② ③. Then it unbinds the first incoming channel of the operator to be migrated ④, it waits until every worker finished executing ⑤, and then it unbinds the outgoing channel ⑥. In this way the runtime first stops the incoming flow, and then waits until all the messages are sent downstream. Finally, the runtime removes the operator ⑦. If the operator was stateful, the state is shared by Redis replication, thus already available on the target peer.

Operator Cloning

The process forking described in section 5.2.2 is executed up to the point in which the operator is not a bottleneck anymore (i.e., the incoming data stream does not accumulate in the incoming queue), or the peer hosting the operator has no more CPU available to host more forked processes. In that case, the local controller informs the global controller which in turn looks for a suited peer to host part of the operator's computation (based on the constraints of the operator and the ranking algorithm 5.3). Once found, the copy of the operator is started on the found peer, and connected accordingly to the topology.

The process works in the following way. Once a suitable peer is found, a copy of the operator is run, then first the outgoing bindings (if any) are performed, and then the incoming bindings are performed. This way, when the data starts flowing, we have the outgoing bindings ready to forward data downstream.

5.2.4 Disconnection Handling

During the execution of a topology, a peer may abruptly disconnect from the WLS network. This could happen as a result of a network error, or as a peer crash. Application errors resulting from wrong operators implementation are not dealt by the runtime and should be resolved by the users of the system. Whenever a peer abruptly disconnects (i.e., a Web browser crashes), the controller notices the channel closing through error handling and starts executing a recovery procedure in order to restore the topology by restarting the lost operators on available peers. The recovery is similar to the migration decision algorithm: the controller restarts the lost operators on peers satisfying the deployment constraints and with the highest ranking. It may be possible that the channel closing was caused by temporary network failures. In this case, if the peer assumed lost comes back up, the connections will be restored by the communication channel primitive (ZMQ and WebRTC automatically handle reconnections, while for WebSockets we implemented a reconnection timeout on the peer).

5.3 Implementation

In this Section we describe the implementation of the global controller and the local controller. The local controller features two different implementations, one for Web servers and one for Web browsers.

5.3.1 Global Controller Implementation

The main concerns of the global controller are to monitor connected peers and to take decision on where to run operators. It is global to the topology and thus, to all the WLS peers. The global controller is composed by routines that analyse the current state of the peers connected and, upon request, decide where to run a given operator. This turns out to be useful when submitting a topology to WLS, but also when local controllers ask for help to the global controller (operator cloning), in order to distribute the work on available peers. The metrics used to take the deployment decision are introduced in Section 5.3.4.

From the point of view of the global controller, the operator cloning procedure works exactly like running a new operator on a given peer. The global controller finds an appropriate peer candidate, and through the RPC command layer it starts a copy of the overloaded operator, binding it accordingly.

The monitoring of the peers is done through heartbeats sent by the local controller of each peer to the global controller. In this way, the global controller has a snapshot of the peers (with all the operators and workers running on them) that can be used in case of faults. At the same time, the global controller can keep into account the latest configuration of the peers in order to decide where to parallelise the execution of operators or run another topology.

5.3.2 Web Server Local Controller Implementation

The role of the local controller is to check on the peer and the different operators running on it, and to help parallelising the work on the peer if needed. The parallelisation is based on the metrics collected on the workers running in the peer, as well as the data of its CPU, accessible thanks to the `os` module available on Node.js, and the `overcpu` module¹. The worker metrics, specifically the request rate and the response rate, help the controller decide if the topology is experiencing a bottleneck on that operator.

¹Developed by Achille Peternier at USI

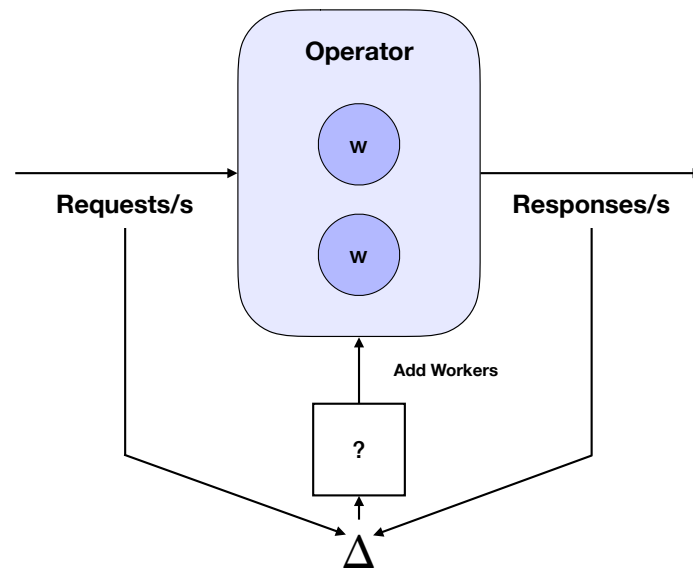


Figure 5.2. Visual representation of the Web server local controller cycle.

Figure 5.2 shows a visual representation of the Web server local controller algorithm over a streaming operator. Request and response rates are used to determine if the controller needs to add more workers to the operator. Algorithm 1 shows the pseudocode implementation of the controller.

Data: Operator o

Result: Workers to be added

```

requestRateMean  $\leftarrow$  computeRequestRateMean(Operator  $o$ ) ;
responseRateMean  $\leftarrow$  computeResponseRateMean(Operator  $o$ ) ;
delta  $\leftarrow$  requestRateMean - responseRateMean ;
if delta > 0 then
  | requiredWorkers  $\leftarrow$  delta / o.numberOfWorkers ;
else
  | requiredWorkers = 0;
end
return requiredWorkers

```

ALGORITHM 1: Server-side local controller algorithm to add new workers to an operator.

The algorithm takes the metrics of each worker in a given operator. It first computes the request rate mean for the whole operator, and then the response rate mean. Then it computes a delta by subtracting the response rate to the re-

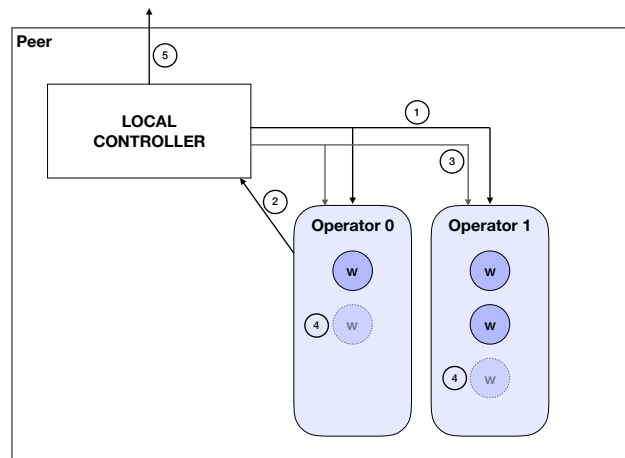


Figure 5.3. Local Web server controller behaviour.

quest rate. This tells the controller how the operator is performing. If the request rate is higher than the response rate ($\Delta > 0$), then the operator is experiencing a bottleneck: more stream elements are incoming than the ones that are outgoing, additional workers may be required to solve the issue. To decide how many workers have to be added, the algorithm divides the delta by the number of workers currently running in the operator. The higher the number of currently deployed workers, the smaller the impact of delta over the resulting workers to be added. If delta is big and there are very few workers running in the operator, a higher number of workers will be added. If the two rates are equal ($\Delta = 0$), the operator doesn't need additional workers. In some cases the response rate may end up being higher than the request rate ($\Delta < 0$). This could be a temporary result due to the addition of workers to solve a bottleneck. In this case the controller treats the result as $\Delta = 0$ and does not interact with the operator.

The local controller checks on the running operators through polling: the controller polls the operators every 500 milliseconds for metrics related to their execution, if an operator takes too long to reply a controller query message, the local controller assumes it is too busy dealing with incoming messages. As a result, the local controller tries to add more workers, one per unresponsive cycle. If the CPU of the peer indicates that no more workers are able to run on the machine, the local controller will contact the global controller using the same RPC channel used to send messages from peer to peer, asking for a peer where to parallelise the execution (operator cloning). Otherwise, one worker is added to the operator.

Figure 5.3 shows the cycle of the Web server local controller. The local controller asks for local values to the operators through polling (1). If the reply by the operator shows a bottleneck situation (2), or if no reply is received, the local controller issues a command to add a single worker on the operators (3) which in turn add the worker (4). The polling cycle continues during the whole lifespan of the peer, as long as there are operators running on it. If the CPU usage is capped, the local controller contacts the global controller to clone the bottleneck operator(s) (5).

Workers Self Shutdown

Workers implement a timeout (from now on, the idle timeout) based on the average interval between messages received, and a counter. Each time a worker receives a message, its counter resets to zero. When the idle timeout fires, the counter is increased, the average interval between messages is re-computed, and the timeout is set again. If the counter reaches a given threshold and the idle timeout fires, the worker initiates a process to self shutdown. For the experiments we present in Chapter 7, we set a timeout counter threshold of five. The value of the threshold can be modified in the code to suit different families of applications.

5.3.3 Web Browser Local Controller Implementation

The Web browser local controller executes the work of the Web server local controller with all the limitations given by the Web browser environment. Web browser peers are fundamentally different from Web servers peers and need a different approach to deal with load balancing and fault tolerance for the following reasons.

- The Web browser hides most of the hardware information from the JavaScript virtual machine. The OS cannot be inquired directly, nor use the Node.js modules we used server-side to inspect the machine's CPU.
- The Web browser is just an application running on a fast mutating environment, where the user opens and closes applications, or leaves them idle indefinitely. Thus, it is difficult to predict the resource availability of the device. Besides, the sandboxed processes of JavaScript have no different priority than other running applications.

- By their nature, Web browsers live shorter in comparison with Web servers. Ideally, the controller should be able to adapt the streaming computation as fast as possible to exploit the machine before it leaves.

To cope with the differences and the lack of more in-depth information regarding the hosting Web browser, we decided to implement a flow control mechanism that we called slow mode. The slow mode triggers when the queue of an operator becomes too long, and tries to avoid overfilling queues of overloaded operator. It is triggered by a two-threshold rule:

$$\begin{aligned} Q(t) > T_{qh} &\rightarrow \text{SlowModeON} \\ < T_{ql} &\rightarrow \text{SlowModeOFF} \end{aligned}$$

The idea behind the slow mode is to slow down the input rate of a given (overloaded) operator to help it dispatch the messages in its queue Q , while increasing the input rate on other instances of said operator. Once the queue is consumed below a given threshold T_{ql} , the controller removes the slow mode, re-enabling the normal stream flow. In [BGP15b] we tuned many aspects of the controller, including the slow mode, for three different families of experiments. Results suggested that $T_{qh} = 20$ messages in the queue were enough to trigger the slow mode, which was released the moment the queue reached $T_{ql} = 10$ or less elements. We show said results and more in-depth evaluation in Chapter 7.

To compute the CPU usage we rely on the `navigator.hardwareConcurrency` API, which is not as precise as inquiring the underlying machine.

$$P(t) > T_{CPU} * \text{hardwareConcurrency}$$

When the number of WebWorker threads P on the machine reaches the amount of concurrent CPUs T_{CPU} available on the machine (100% CPU capacity), the operator cloning procedure is started.

Like the Web server local controller, the Web browser local controller cycles through the operators hosted on the device and collects data through polling, every 500 milliseconds. The data is gathered by the operators by querying their workers, and is passed to the controller upon being queried. The execution data contains throughput information, input queue sizes, and how many messages were executed in that cycle. This helps the controller determining if any of the operators running on the Web browser is a bottleneck in the same way as the Web server local controller, but keeping into account the queue size as well.

The local controller checks the number of times each worker has been called and remove the worker that worked less in that cycle. A more in-depth description and analysis of the Web browser controller can be found in Andrea Gallidabino's Master Thesis [Gal14]. We also studied the tuning of the controller

cycle, the slow mode threshold and the CPU threshold for the Web browser local controller. We presented the work in [Bab17] and show the results in Chapter 7.

5.3.4 Ranking Function

The ranking function is used by the global controller to take decision on where to run operators. The following metrics are taken into account by the controller for selecting the most suitable peer for each operator:

- **Energy consumption** for example when the computation of an operator is too taxing on a battery-dependent peer. In this case a migration may occur, moving the heavy computation from a mobile device to a desktop or Web server machine. Thus, the priority is given to fully charged mobile peers or to hard wired peers without battery dependency.

- **Parallelism** can be achieved by cloning a bottleneck operator. For example, a migration or a cloning operation may be expected if the computation is very CPU-intensive and the peer hosting the operator not only has its CPU full but it is also the topology bottleneck. Thus, higher priority is given to peers with larger CPUs and higher CPU availability.

- **Deployment Cost Minimisation** by prioritising Web browsers instead of making use of Web servers, WLS tries to avoid incurring in additional variable costs given by the utilisation of pay-per-use Cloud resources.

The presented metrics can be accessed on a Web server through the Node.js APIs, while on the Web browser we can again use the HTML5 APIs to gather the battery levels besides a rough estimate of the maximum available processing power.

The ranking function is evaluated by the controller at topology initialization to find the best deployment configuration of the operators, and while the topology is running to improve its performance and deal with variations in the set of known/available peers. For each peer, the function uses the previously described constraints and metrics to compute a value that describes how much a peer is suited to host an operator. The function is defined as follows:

$$r(p) = \sum_{i=1}^n \alpha_i M_i(p) \quad (5.1)$$

Where n is the number of metrics, α_i is the weight representing the importance of the i -th metric. We set these values to 1, but can be modified based on user requirements. M_i is the function that returns the current value of the i -th metric in p . It linearly maps the score of the peer in terms of CPU availability and

remaining battery levels from -100 to 100. Devices plugged to a power source (no battery consumption) have a fixed score of 100. The result gives an estimate of the utility of the peer p to run one or more Operators on it. In case the utility turns out to be negative, the semantics implies that the peer p should no longer be used to run any operator.

Whenever a Topology has to start, the controller polls the known Peers and executes the procedure shown in Algorithm 2.

Data: Known Peers, Topology

Result: Peers Ranking

$P \leftarrow$ Known Peers ;

foreach Operator o in Topology **do**

foreach Peer p in P **do**

if $\text{!compatible}(o, p)$ **then**

$P \leftarrow P \setminus p$;

end

end

end

if $P = \emptyset$ **then**

return cannot deploy Topology

else

foreach Peer p in P **do**

 Poll p for its current metrics $M_i(p)$;

 Compute $r(p)$ according to Eq. (5.1);

end

return Sorted list of available Peers P according to their assigned metrics $r(p)$

end

ALGORITHM 2: Peer Ranking for Topology Initialization

The controller first determines which of the known peers are compatible with the operators of the topology. Then it polls each peer for its metrics. Once received, the ranking is computed and stored for further use. The peers are then ordered from the most suited to the least suited, then for each operator to be deployed the controller iterates the list top to bottom deciding which peer will host it. This deploys the whole topology and starts the data stream. As the topology is up and running, the ranking is performed periodically. This is used by the controller to check the status of the execution and adapt the topology to a) changes in workload (reflected by changes in the CPU utilisation of the peers), b) changes in the available peers (and thus deal with disconnections), as well as c) changes

to the topology structure itself (e.g., when new operators are added to it). The status check is shown in Algorithm 3.

The controller checks each peer hosting at least one operator in the topology. If the value of the ranking is negative, the controller will try to find a better peer to host its operator(s). First it will filter out the peers based on the constraints of the operators, then based on the outcome of the ranking function it will run the operator on other better ranked peers. If no peer is found that is suited for the migration, the migration does not take place. The controller will migrate the operator as soon as a peer with the needed constraints connects to the WLS network.

Data: Known Peers

Result: Migrated Operators

foreach *Known Peer p hosting at least one Operator* **do**

```

    | if  $r(p) < 0$  then
    | | Get Operators running on p foreach Operator o running on p do
    | | |  $P \leftarrow$  Find Peers satisfying constraints of Operatori to host o;
    | | |  $p' \leftarrow$  Best Peer in P to run Operator o based on the ranking  $r(P)$ ;
    | | | if ( $p'$ ) then
    | | | | Migrate Operator o to Peer  $p'$ ;
    | | | else
    | | | | return;
    | | | end
    | | end
    | end
end

```

ALGORITHM 3: Migration Decision

Part III
Evaluation

Chapter 6

Application Case Studies

The design and improvements of WLS have been user and developer-driven. This Chapter introduces some of the applications that have been developed using the WLS framework, showing how the proposed features in the framework have been useful for the developers. The application use cases and the developer's comments and feedback helped us improving the overall interaction between the programmers and the framework.

6.1 Study Week in Informatics

In collaboration with Schweizer Jugend Forsch, every year the University of Lugano organises a study week in informatics for middle and high-school students from all over Switzerland. In 2014 we proposed one project related to Web Liquid Streams for such event. During the first day, the students would learn the basics of programming with JavaScript. For the remainder of the week they would code a simple WLS application that monitors a small plant. It was the first time we let non-developers use WLS.

Figure 6.1 shows the topology implemented by the students. The monitoring was done through a webcam (on a Web browser running the `webcamProd.js` operator) and a THP (temperature, humidity, and pressure) sensor wired to a Raspberry Pi running the `data_producer.js` operator. The data is forwarded and joined in `filter.js` which is placed on a Web server to format the data and store it in a database, and then forwarded downstream to a Web browser that showed the data about the plant's environment as well as the webcam feed.

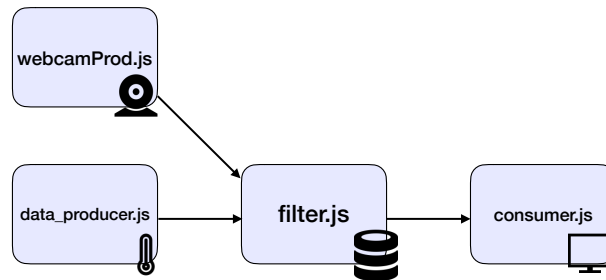


Figure 6.1. Topology implemented in the study week in informatics.

The code related to the Web browser plotting of the data and the webcam feed retrieval was given as part of the skeleton to the students. The students coded by themselves all the operators and the topology.

6.1.1 Lessons learned

The students were very proactive and tried to understand bugs by themselves. Some of the bugs were difficult to spot because the WLS runtime didn't give proper info about what was happening. People with a background in IT or some experience with JavaScript could easily spot the issues by reading the error messages, but for middle and high-school students with no experience in programming it was a difficult task. After this experience, we decided to improve the error logs in order to help developers understand by themselves some of the most common issues regarding operator development.

6.2 Inforte Seminar on Software Technologies and Development for Multi-Device Environments

In Summer 2015 we held a workshop on WLS as part of the Inforte seminar on software technologies and development for multi-device environments summer school¹ in Tampere, Finland. The workshop had an 8-tasks assignment in which participants would get started with WLS and its functionalities (how to set up a topology, how to run it, how to run it in a Web browser, how to migrate operators,

¹http://inforte.jyu.fi/events/multi-device_environments

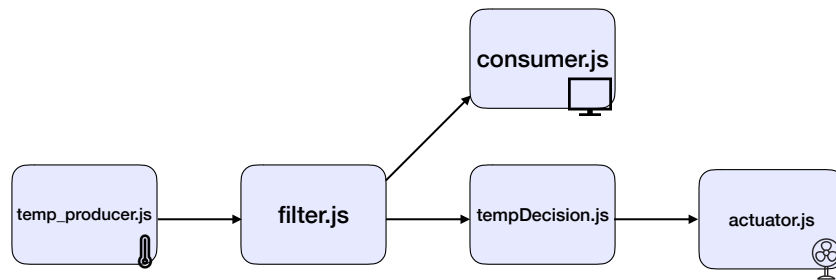


Figure 6.2. Topology implemented in the workshop.

how to gather data from Raspberry Pi sensors), and by the end of the workshop they would produce a fully functioning data stream topology.

Figure 6.2 shows the final topology implemented during the workshop. The producer `temp_producer.js` runs on a Raspberry Pi with a DS18X20 temperature sensor wired to it. The raw data is forwarded to a filter which polishes it and sends it further downstream to the `tempDecision.js` and the `consumer.js` operators. The `consumer.js` Web browser operator shows the temperature on screen by the means of the D3 plot tool to which the polished data is fed. The `tempDecision.js` runs on a server and decides if an actuator has to be started. If the temperature gets above thirty degrees, the `tempDecision.js` operator forwards a message to the `actuator.js` operator which starts a fan. If the temperature is below thirty degrees, the fan is turned off. During the summer school we did not have access to many servo motors to simulate a fan, so the `actuator.js` operator was implemented on a Web browser operator which showed a fan animated gif if the fan had to start running.

We had around fifteen participants, some of which worked in pairs. The total number of Raspberry Pis available was ten, each group could work with one. Participants had access to a single server on which they started the WLS framework to run the server side of their streaming topologies. Web browser operators were run on their own laptops.

6.2.1 Lessons learned

We started by giving a small introduction on the WLS framework, gave access to one of our servers to the participants, and explained how to start WLS. The list of commands displayed with `help` gave them an idea on how to run WLS from the command line interface. Since we did not have much time, WLS was already installed on our server. About half of the participants didn't know how to program in JavaScript, so it took some time for some of them to complete simple programming tasks. Some of the participants had hard time understanding the commands in the command line interface offered by WLS. In fact, we used nontrivial keywords which were not user-friendly for people using WLS for the first time.

After the workshop we decided to improve the command line interface by implementing more self-explaining commands in the command line interface. We also took the chance to improve the commands descriptions when typing the `help` command.

6.3 WLS as a Mashup Tool

A mashup is a Web application that makes use of data, presentation, or functionality from two or more Web sites to create a new service. The concept of mashup and the subsequent interest in the mashup tools started to appear as more and more Web services and Web Data sources were released [ZRN08]. While mashups can be built using traditional Web development tools, languages, and frameworks, specialised mashup composition tools have appeared focusing on raising the level of abstraction and thus enabling non-programmers to compose mashups [LLX⁺11]. In this Section we show an example of how streaming APIs can be integrated using our stream processing framework. We do so by showing our submission [GBP16] to the Rapid Mashup Challenge 2015 [DP16] held at ICWE 2015. Our work ranked third (out of eight) in the competition.

Within Web Liquid Streams, mashup components can be seen as stream operators, while the mashup can be defined by building a streaming topology. The following list shows the features that WLS offers to mashup developers.

Reusable mashup components Mashup components written as JavaScript operators can be reused in more than one topology. Component development is completely open for developers that can reuse JavaScript libraries and remotely access any Web service API.

Live mashup development Thanks to the flexibility of the topology at runtime, mashup developers can change their mashup while it runs. Developers can *run*, *stop*, or *bind* mashup components, furthermore they can decide to *migrate* them on any peer connected to the application.

JSON mashup definition language Mashups can be defined by using our internal DSL based on the JSON syntax. Once the mashup is launched, the structure of mashups created can be edited and deployed to reconstruct a different mashup.

Web of Things mashups WLS enables integration with smart devices and sensors which can create streams of data. WLS can run mashup components directly on those devices so that they can directly access hardware sensors and actuators.

Distributed user interface mashups Multiple operators to visualise the data stream can be instantiated and deployed on different client devices so that the same mashup results can be shared among multiple users.

In the challenge demo we presented a topology deployed both on Web server and Web browsers. We used three different APIs: Google Maps², GeoNames³, and the Twitter REST⁴ and streaming⁵ APIs. During the presentation we showed how operators and bindings behave in a topology and how the latter can change dynamically at runtime by adding, removing, or migrating operators. Figure 6.3 shows the topology that has been implemented during the mashup challenge. At the end of the demo we also involved the audience by deploying the mashup UI components on their Web browsers to see for themselves the results of the stream processing.

The topology has three producers: `clickMarker.js`, `hoverMarker.js`, and `tweetRetriever.js`. The first two are placed on a Google Map Web page and react to user-produced events: `hoverMarker.js` reacts to the user hovering the mouse on top of a marker on the map and draws the retrieved Tweet on top of the marker. `clickMarker.js` receives the click event on top of a marker, and forwards the event to the `retweetGatherer.js` operator which, using the Twitter APIs, gathers the retweets of the clicked tweet on the map. The tweets are passed to the `tweetGeolocator.js` operator

²<https://developers.google.com/maps/>

³<http://www.geonames.org/>

⁴<https://dev.twitter.com/rest/public>

⁵<https://dev.twitter.com/streaming/overview>

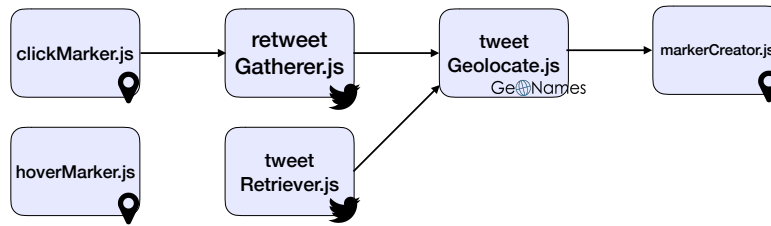


Figure 6.3. The mashup topology.

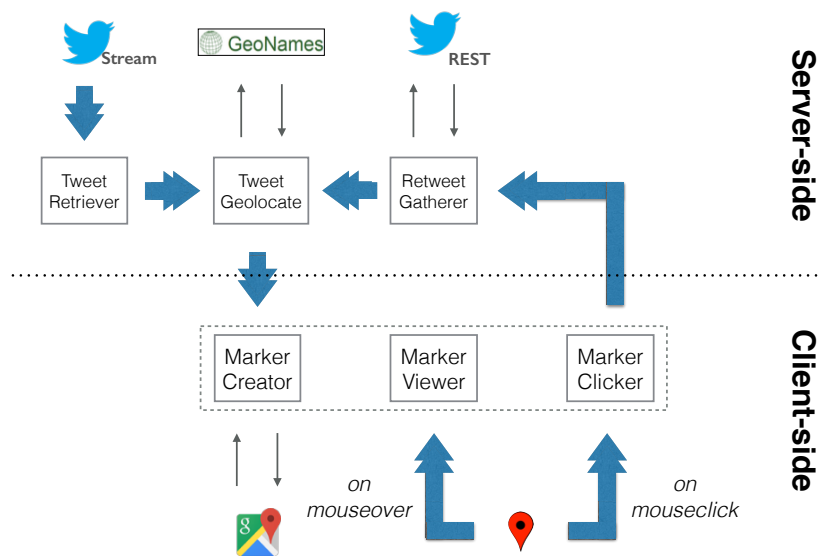


Figure 6.4. Physical deployment of the operators and data flow in the mashup topology [GBP16].

which, through the GeoNames API, gets the location of the retweets, and forwards the result to the consumer. `markerCreator` receives the retweets and places them on the map. The third producer is `tweetRetriever.js` which is constantly connected to the Twitter API and forwards the received tweets to the `tweetGeolocate.js` operator to eventually display new tweets on the map.

Figure 6.4 shows the physical deployment of the operators as well as the data flow of the topology in our mashup example.

Figure 6.5 shows a screenshot of the mashup running on a Web browser. By clicking on a tweet (big marker), the retweets (small markers) are placed on the interactive map.

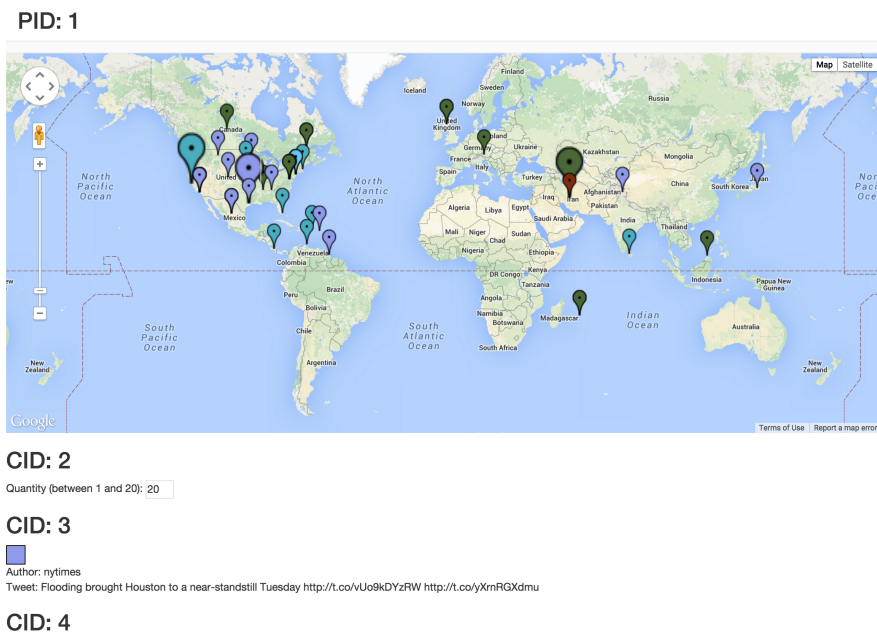


Figure 6.5. Screenshot of the mashup running on a Web browser [GBP16].

6.3.1 Lessons learned

During the execution of the demo, Andrea Gallidabino migrated one of the Web browser operators from his machine to another (random) machine by using the numerical ID assigned by WLS. After the migration we had to ask to the participants who received the operator, and sometimes it took a while for somebody to raise their hand, wasting demo time. By using a logical naming scheme – for example, the name of the machines – instead of physical addressing, we could migrate more easily operators from one machine to another being sure of the migration destination.

6.4 Software Atelier 3: The Web – Home Automation System Project

During the Web Atelier course for second year students in the Fall semester 2015 students learn client/server programming, emerging Web technologies, and Web design. Web Atelier covers Web technologies such as REST, HTTP, CSS3, HTML5, and Web Components, and teaches how to program in JavaScript on the client side and on the server side with the Node.js framework. During the second part of the semester, students focus on a project that can be proposed by themselves,



Figure 6.6. Topology implemented during the Software Atelier 3 project

or by the teaching assistants. Given the skills learned during the first part of the semester, during the Fall semester 2015 we proposed a small home automation system project built with WLS. We provided all the help and support related to the implementation of sensors, while the students had to work on their own on the topology and the operators implementation.

The students that picked our project decided to build an open space live monitor. The open space in the Faculty of Informatics of the University of Lugano is a space where students can work together and cooperate for projects and study for exams. The open space worked very well during the first years of the faculty, but quickly became overcrowded as classes got bigger and students from other faculties decided to use it to work on assignments. Nowadays the open space is a crowded and noisy place where it has become difficult to concentrate. Students use it to work on assignments and projects, but it's not as silent as it used to be. To show the degradation of the open space, the WLS project team decided to monitor the noise levels of the open space through this live monitor. They also decided to keep track of the light level, in order to see how frequently the open space was used during the night.

Figure 6.6 shows the topology implemented by the students. The noise and light levels are gathered by sensors wired to three different Tessels and a Raspberry Pi deployed across the open space. The data is sent to a server where it is stored and polished, and then forwarded to a Web browser operator which shows the noise and light levels.

While the topology and the operators could be built by coding, students had to build by hand a small module with a light sensor, a microphone, and a resistor. Figure 6.7 shows the module they have built. The module was then wired to the Tessel and deployed across the open space.

Figure 6.8 shows a screenshot of the running application. The four graphs show the four different deployment of the Tessels and Raspberry Pi across the open space. The students decided to normalise the values of light and noise in

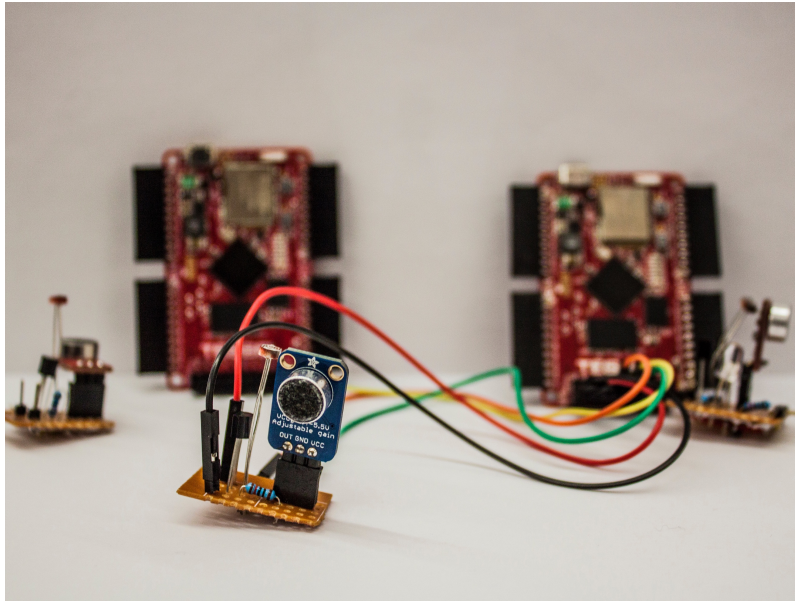


Figure 6.7. Tessels and microphone modules built by the students.

a range between 0 and 100. They also decided to test two different visualisations of the data: one with smooth transitions and one with a bar chart.

6.4.1 Lessons learned

We worked closely with the students, helping them whenever they had issues. The main problems the students had were the following.

- Dealing with dependencies

Installing the framework, the dependencies, and the OverCPU module with `node-gyp` can be an issue. The students did not have a lot of experience with the terminal and installing tools from command line, so it took them some time to get the whole system up and running. Two members of the team also installed the wrong ZeroMQ distribution, causing bugs during the execution of the framework. Once again, by having an installer that takes care of installing the right dependencies for the underlying OS could solve this issue and save time.

- Developing operators
- Understanding stream processing applications



Figure 6.8. Screenshot of the application running on a Web browser.

The development of operators was difficult for some members of the team because they did not understand the model in which data was sent and received by operators. This was also related with the difficulty in understanding how stream processing applications work. In this case code examples were very useful, as they could build a topology using example code and see how it worked. For this reason, we decided to keep some code examples in the WLS distribution available on Git that can be used by developers to learn how WLS works. Students were also having difficulties with understanding how some method worked, so we prepared a well-written documentation of most of the method they needed to use, which is now available on the Git page as well.

- Dealing with the controller

While the controller helped the students during their deployments, at the very beginning it was difficult for them to understand bugs. Bugs in the operator implementation could cause operators to be overloaded, and having a controller automatically trying to deal with the issue wasn't helpful as it wasn't clear for the students what was going on at first, and subsequently if the streaming operator they implemented was working properly or not. For this reason we decided to remove the automatic initialisation of the controller for their test phase. They would then turn it on manually when the deployment started.

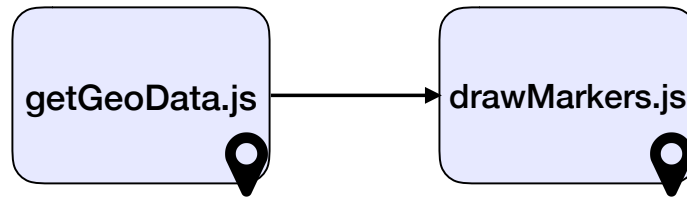


Figure 6.9. Topology for the Karlsruhe example .

6.5 Experimentelle Evaluation des Web Liquid Streams-Framework

During the Spring semester 2016 Web Liquid Streams has been the target of a short-term (6ETCS) software development project. The project focused on concrete collaborative browser-to-browser Web applications and was organised by Professor Christian Zirpins at the University of Applied Sciences in Karlsruhe, in Germany⁶. The project was developed by Tobias Fuss, a Master student, and resulted in a 40-pages report [Fus16] (in German).

In the report, the author describes the interest in exploring novel Web technologies to directly connect Web browsers without passing through a Web server, and the development of such applications. The interest in WLS stems from the application of novel Web technologies that let developers connect in a peer-to-peer fashion Web browsers through a data stream. The report describes the technologies used under the hood in WLS, and the problems encountered by the developer during the installation and development phase. Then, a use case scenario making use of geo-localisation and Google Maps is presented.

The presented application takes the geolocation of mobile Web browsers connected to a topology and, through WLS, displays them on a map. The author explains that the idea of such application originated from the poor road situation in the Karlsruhe area, where many road construction sites exist and are constantly changing. By having an overview of the current road situation in real time, drivers may adjust their own route leading to potentially large time savings.

The topology is a very simple linear two-stage pipeline shown in Figure 6.9. The producer sends the data gathered from by the `navigator.geolocation` Web browser API, and the consumer displays it on the map. Depending on the speed of the device, the Google Maps marker is given a different colour.

```
1 var k = require("../k_globals/koala.js");
```

⁶<https://www.hs-karlsruhe.de/en/>

```

2
3 // Creates the script to access the geolocation
4 k.createScript("locationScript", "js/location-script.js");
5
6 var latitude , longitude , accuracy , speed;
7
8 setInterval(function() {
9   k.callFunction("getLocation", [], function(position) {
10    latitude = position.latitude;
11    longitude = position.longitude;
12    accuracy = position.accuracy;
13    speed = position.speed;
14
15    k.send({
16      "latitude": "longitude": "accuracy":
17      latitude , longitude ,
18      accuracy ,
19      "speed": speed
20    });
21  }, true);
22 }, 5000);
23 console.log("Producer started , sending locations!");

```

Listing 6.1. Real time road analysis producer.

Listing 6.1 shows the `getGeoData.js` producer code implemented to gather data from connected mobile Web browsers. The operator first adds the `location-script.js` on the Web page, then every 5 seconds sends the geolocation data associated with the peer: latitude, longitude, accuracy, and speed. The geolocation data is gathered directly from the Web page through the `callFunction` call, which calls the `geoLocation` function on the Web page.

```

1 /**
2  * Calls the HTML5-API method "getCurrentPosition" to access
3  * the users location.
4  * @param cb: Callback function
5  */
6 var getLocation = function(cb) {
7   navigator.geolocation.getCurrentPosition(function(position)
8   {
9     cb ({ "latitude": "longitude": "accuracy":
10     position.coords.latitude , position.coords.longitude ,
11     position.coords.accuracy ,
12     "speed": position.coords.speed });
13   });

```

12 };

Listing 6.2. The geoLocation function implemented on the Web browser in the location-script.js file..

Listing 6.2 shows the function getLocation implemented on the Web page and called through callFunction.

The procedure calls the navigator.geolocation.getCurrentPosition function which returns an object describing the position of the device.

```

1 // Creates the Google Maps object and the script to work with
  it
2 k.createHTML("map", "<div id='map-canvas' width='500px'
  height='500px' style='height:500px'></div>");
3 k.createScript("mapScript", "js/map-script.js");
4
5 var latitude , longitude , accuracy , speed;
6
7 k.createNode(function(data) {
8   latitude = data.latitude;
9   longitude = data.longitude;
10  accuracy = data.accuracy;
11  speed = data.speed;
12  k.callFunction("addMarker", ["Test", latitude , longitude ,
  speed ], undefined , false);
13 });
14
15 console.log("Consumer started , showing locations!");

```

Listing 6.3. Real time road analysis consumer.

Listing 6.3 shows the drawMarkers.js consumer code. After adding the map-script.js JavaScript file to the Web page, the operator calls the addMarker function on the Web page forwarding the latitude, longitude, and speed data.

```

1 var mapOptions = {
2   center: {lat: 49.0, lng: 8.4}, // Karlsruhe
3   zoom: 13
4 };
5
6 var map = new google.maps.Map(document.getElementById("map-
  canvas "), mapOptions);
7
8 /**
9  * Adds a marker at the given position with optional title to
  the
10 map.

```

```

11 * @param title: Title of the marker
12 * @param latitude: Latitude for the marker
13 * @param longitude: Longitude for the marker * @param speed:
    Current speed (optional)
14 */
15 var addMarker = function(title, latitude, longitude, speed) {
16     // Color mapping according to speed
17     var color = "D"; // Blue (default)
18     if (speed != undefined) {
19         if (speed == 0) {
20             color = '0'; // Red
21         } else if (speed > 0 && speed <= 10) {
22             color = '4'; // Orange
23         } else if (speed > 10 && speed <= 30) {
24             color = '8'; // Yellow
25         } else if (speed > 30 && speed <= 50) {
26             color = 'I'; // Blue (dark)
27         } else if (speed > 50) {
28             color = 'J'; // Green (light)
29         }
30     }
31
32     var pin = new google.maps.MarkerImage("<Image-URL>" + color
33         ,
34         new google.maps.Size(21, 34),
35         new google.maps.Point(0, 0),
36         new google.maps.Point(10, 34));
37     var marker = new google.maps.Marker({
38         position: new google.maps.LatLng(latitude , longitude),
39         map: map,
40         title: title,
41         animation: google.maps.Animation.DROP,
42         icon: pin
43     });

```

Listing 6.4. The addMarker function implemented on the Web browser in the map-script.js file.

Listing 6.4 shows the code implemented in the map-script.js file. The script first sets up the Google Map widget, then upon receiving a function call from the consumer, it draws a marker in the appropriate position with the assigned color scheme representing the speed of the vehicle.

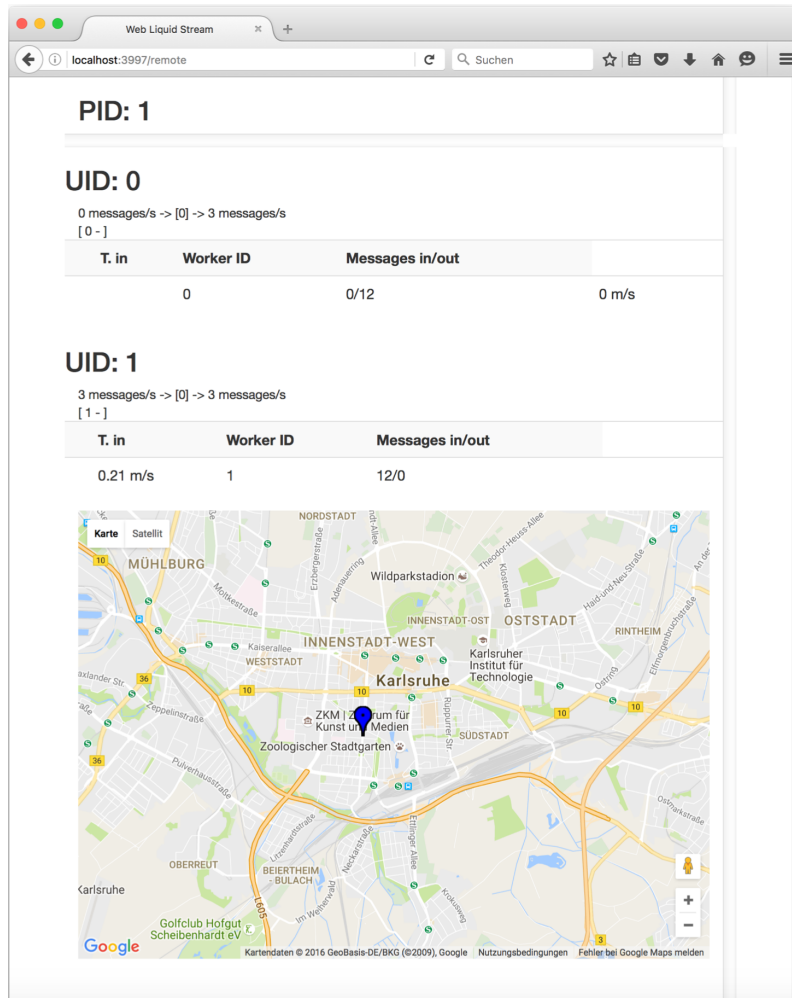


Figure 6.10. Screenshot of the consumer Web page on the running application [Fus16].

Figure 6.10 shows a screenshot of the running application on the consumer's page. A single producer is bound to the consumer, thus the map only shows a single blue marker.

In the conclusion of the report, the author describes his overall satisfaction with the tool, and appreciates how simple and versatile WLS is (besides the initial installation struggle). The author mentions his inexperience with JavaScript and how simple was implementing and connecting operators for WLS.

6.5.1 Lessons Learned

During the development of the project, Tobias asked many questions through e-mail and sought help when stuck in the implementation of his topology. The major problems Tobias has faced happened while carrying the following tasks out:

- Dealing with dependencies

The issues regarding the installation of the framework were caused by the installation of the OverCPU module. Being it a module never released on NPM⁷ (Node Package Manager), it has to be manually installed. To install OverCPU developers are required to install `node-gyp` and use it to compile the module. Depending on the platform the developer is working on, compilation flags in C++ (located in the `bindings.gyp` file) have to be modified in order for OverCPU to be compiled properly. The first struggle for Tobias was discovering how to compile such module by modifying the flags accordingly for his OS (MacOSX).

- Interaction between the Web browser operator and external libraries

The second issue he had was implementing Web browser operators. Tobias knew very little JavaScript before starting the project, but quickly developed a small toy topology to play with WLS before starting with his project. The problems he had while implementing Web browser operators were related to the interaction between external libraries and helper files, and the Web browser operator. This is done in WLS through the `callFunction` method call (described in Section 3.3), which Tobias had a hard time understanding. Through e-mail help he managed to understand how the method works, and managed to include it in his code to show the map.

We took the chance to perform a formative evaluation of WLS by taking advantage of the e-mail exchange we had with Tobias and annotate every time he had issues with the system. To solve the installation issues, we should build an installer which takes care of the difference in OS and performs all the needed dependencies installation, removing this burden from the developers. The issues regarding the understanding of the framework API could be solved by preparing a more detailed documentation of each method available. This has been done in this Dissertation and on the WLS GitHub page⁸ dedicated to the project after Tobias had such issues.

⁷<https://www.npmjs.com>

⁸<https://github.com/masiarb/Web-Liquid-Streams>

Chapter 7

Performance Evaluation

7.1 Overview

In the previous Chapters have shown how Web Liquid Streams is able to deal with faults autonomically, as well as parallelize the execution of operators on a single peer by increasing the number of workers, and on multiple heterogeneous peers. Our aim is to let developers only deal with the implementation of the code and the topology, and let the framework deal with nonpermanent disconnections, battery shortages (given that many Web-enabled devices are battery powered), and load fluctuations autonomically. In this Chapter we evaluate how well Web Liquid Streams deals with such issues in a fully autonomic way.

Table 7.1 introduces the machines we used to perform the evaluation in the various Sections of this Chapter. All the Web server resources run Node.js version 0.10.15, while all the Web browser resources run Google Chrome Version 40. Web servers and laptops are wired through ethernet with 100Gbit/s of maximum bandwidth, while smartphones and iPad are either connected through 4G or WiFi (802.11n, 300Mbit/s), unless otherwise stated.

The Chapter is organised as follows. In Section 7.3 we show how WLS is able to migrate operators (introduced in Section 4.4) and recover disconnections at runtime (presented in Section 4.1.1). In Section 7.4 we show how well the local controller (introduced in Section 5.3.2) adapts to changes in the workload at runtime, increasing and decreasing resource usage (workers) to adapt to the throughput. Section 7.5 shows the global controller (shown in Section 5.3.1) evaluation when picking target peers for deployment with respect to a random decision and an inverse ranking function. In Section 7.6 we evaluate the local controller in the Web browser (presented in Section 5.3.3) and perform some fine tuning for a specific class of experiments. We show how we are able to keep

| Machine | Experiment |
|--|--|
| (3x) MacBook Pro quadcore i7 (2012) 2.3GHz, 16GB RAM, OSX 10.12 | Section 7.3 (3x), Section 7.4 (3x), Section 7.5 (1x), Section 7.6 (1x), Section 7.8 (1x) |
| MacBook Pro quadcore i7 (2011) 2GHz, 4GB RAM, OSX 10.12 | Section 7.6, Section 7.8 |
| MacBook Pro quadcore i5 (2011) 2.53GHz, 4GB RAM, OSX 10.12 | Section 7.6, Section 7.8 |
| Windows 7 quadcore 2.9GHz, 8GB RAM | Section 7.3, Section 7.4, Section 7.5 |
| iPhone 5S Dual-core 1.3 GHz, 1 GB RAM, iOS 8 | Section 7.3, Section 7.4, Section 7.5 |
| Samsung Galaxy S4 Octa-core (4x1.6 GHz Cortex-A15, 4x1.2 GHz Cortex- A7), 2 GB RAM, Android Lollipop 5.0.1 | Section 7.3, Section 7.4 |
| (3x) iPad 3 WiFi Apple A5X, 1 GB RAM, iOS 8 | Section 7.3 (3x), Section 7.4 (3x), Section 7.5 (1x) |
| DELL Server with twenty-four Intel Xeon 2GHz cores, 128 GB RAM, Ubuntu 12.04 | Section 7.3, Section 7.4 |
| DELL Server with four Intel Core 2 Quad 3GHz cores and 8GB RAM, Ubuntu 12.04 | Section 7.3, Section 7.4 |
| Samsung Galaxy Tab A, Octa- core1.6GHz, 2GB RAM, Android Marshmallow 6.0.1 | Section 7.8 |

Table 7.1. Machines used during the WLS evaluation.

throughput and queue sizes low by adjusting the local controller through four different configurations. Finally, in Section 7.8 we show how the system behaves in case of failures, and how the performance of the topology degrades as less machines are available.

7.2 Metrics

In this Chapter we use a set of different metrics to evaluate our system.

The throughput of a topology is an important metric that defines how many messages pass through the topology each second. The higher the throughput, the higher the effort on the streaming operators that have to deal with a higher number of messages. We use the throughput to define the effort on the topology. We measure the throughput at every controller cycle, which corresponds to 1 second for both the server and the Web browser controller. The Web browser controller cycle has been studied and modified in Section 7.6, thus the computation of the throughput is affected as well.

The latency of a message is the time taken by a message to traverse the whole topology. As the stress in the topology grows (i.e., by increasing the throughput, or by sending messages that have long processing times), the topology may experience bottlenecks that affect the latency of the messages. It is measured at the consumer after the consumer is done processing the single message.

The queue size gives an idea of how much a Web browser topology is stressed, and is a measure strictly related to the Web browser part of WLS. We explain the differences in the implementation of the Web server and Web browser controllers in Chapter 5. The size of the queue is affected by the throughput and suggests bottlenecks in a Web browser streaming operator, and affects the latency of a message. We measure the queue size in CPU-intensive operators through the messages passing in the operator. Each message will have the current queue size attached to it before leaving the operator.

The worker number is a measure related to how many workers are spawned to face high loads in a streaming operator. As the load increases, the controller increases the number of workers in an operator to face bottlenecks, keep the input throughput, and lower latencies and queue sizes. This number oscillates throughout the experiment depending on how the load changes.

The peer number measures the number of peers available to deploy streaming operators. It affects how the topology is able to deal with bottlenecks (by parallelising the execution on more than one peer) and failures (by restoring the computation on another peer). Both the worker number and the peer number



Figure 7.1. DES Encryption topology.

are measured when the message leaves the CPU-intensive operator (filter). This gives us an idea of the state of the topology at the end of the execution of each message after the CPU-intensive operation.

In some plots we use the message count as the x-axis. We order the messages starting from zero in their arrival order and plot the measures attached to them. This shows how the topology behaves through time as messages arrive at the consumer. We store the payloads attached to the messages at their arrival at the consumer and plot them using the index of the message as the x-axis.

7.3 Operator Migration and Disconnection Recovery

In this Section we aim to demonstrate how the framework, through the coordinated work of the controllers, is able to autonomously migrate streaming operators, and to reconstruct the topology when a peer disconnects (and the operators running on it are lost).

The topology we use to evaluate the ranking controller is shown in Figure 7.1. The topology takes as input a stream of tweets, encrypts them using triple DES and stores the encrypted result on a server. The topology thus consists of a pipeline of three operators: the constraints of the first and last operators are to run on a server, in order to respectively read the Twitter stream and store the encrypted result.

The CPU-intensive operator average execution times significantly change depending on the peer computational power: from 2.003ms on the MacBook (WLS server), 2.103ms on the MacBook (WLS browser), 2.115ms on the Windows machine (WLS server), 12.508ms on the iPhone and 13.005ms on the iPad. Since this operator can be deployed on all of these devices, as we are going to see, the

controller decisions on the liquid deployment have an impact on the topology end-to-end latency. Given the unpredictable dynamics of the Twitter Firehose API we decided to sample 100000 tweets and use them as benchmark workload for all experiments. The size of the messages exchanged along the stream is thus less than 1Kb.

We compare three scenarios: the first one is a stable scenario where the controller deploys the operators in the best possible way and no failure, or battery shortage happens. The second scenario involves deployment on devices that can handle the computation but are short on battery. To make the experiment reproducible, a battery shortage is triggered every 5000 messages, which in turn triggers a migration on another device. The final scenario instead triggers a disconnection every 5000 messages. Disconnected peers are substituted by the controller with idle ones.

We used the twenty-four cores server as the data producer, and the four-cores server as the consumer. We left the other peers idle for the deployment of the filter, which can be deployed on both Web server and Web browsers. We always start the experiment on the Web browser peer running on the MacBook Pro. We also manually blacklisted the twenty-four cores and the four-cores servers for the deployment of the filter operator. This was done as in the battery shortage scenario we would have to trigger a battery shortage event on machines that are plugged in, while in the failure scenario we wanted to avoid triggering a failure in a peer that was hosting the producer operator, or the consumer operator. For these reasons we decided to only use the above mentioned pool of devices to run the filter operator.

Figure 7.2 shows the throughput measured at the consumer during the experiment runs for the three different scenarios. The static and the battery scenarios share similar throughput, while the failure scenario presents downward spikes caused by the sudden disconnection of a peer hosting the computation, thus breaking the data stream. The vertical dashed lines represent the "battery shortage" and the "crash" events every 5000 messages. The Y axis represents the throughput measured as messages per second, while the X axis represents the total number of messages received by the consumer.

On the one hand, the battery scenario gives a good indication of how well the controller performs in gracefully migrating the operators from the low battery peer to another one while the topology is running, without having significant impact on the throughput. On the other hand, the failure scenario shows that the controller is able to quickly recover the disconnected operators redeploying them on another suitable peer so that the stream throughput returns to the nominal level after the failure occurs.

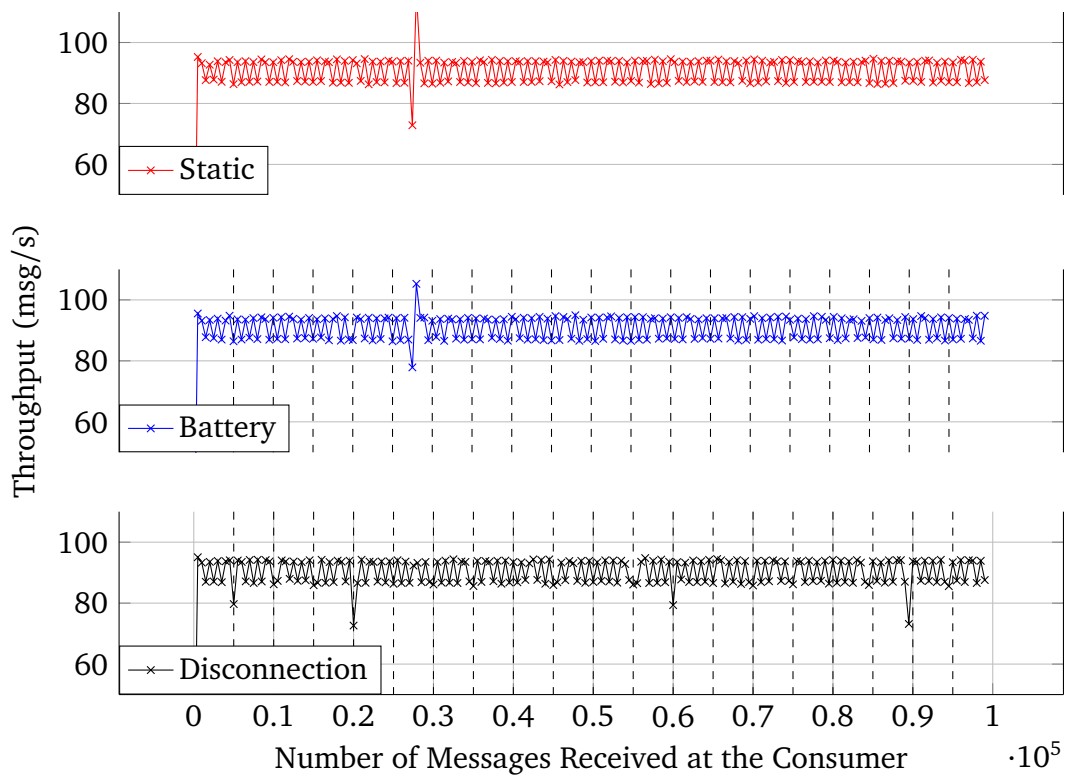


Figure 7.2. Operator Migration and Recovery Impact on Throughput. Vertical lines indicate Low Battery and Disconnection events.

Figure 7.3 statistically summarises the throughput over the entire runs. The battery and stable scenario show once again a very similar throughput, confirming our claims on the adaptability of the controller. The failure scenario shares the same throughput in most cases, but it presents a lower median throughput and lower whisker, caused by the sudden disconnections that momentarily stop the data stream.

Figure 7.4 shows the number of workers in the filter operator during the whole experiment. The Y axis represents the number of workers, while the X axis is once again the number of messages received by the consumer.

We can see how in the static case, the controller parallelises on a single machine, adding up to four workers in the MacBook Pro's Web browser, and then lowering them to three towards the end of the experiment. The battery shortage scenario shows how the shortage happens on one peer and the migration immediately triggered moves the operator with the same number of workers on another available peer. In this case the peer chosen is the MacBook Pro with the

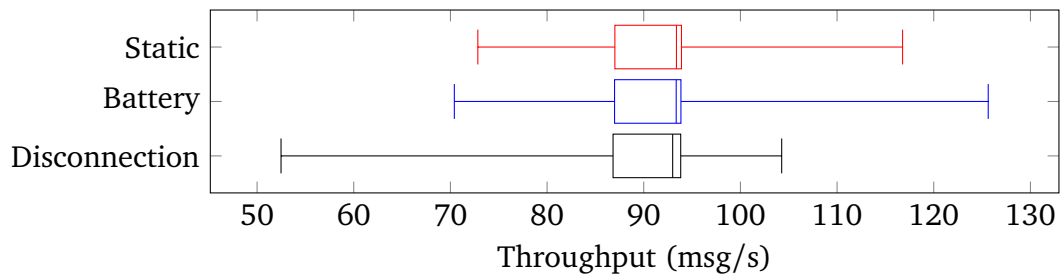


Figure 7.3. Throughput distribution during the operator migration and recovery experiments with low battery and disconnection.

Web server deployment of WLS. When the battery event is triggered on it, the controller chooses again the MacBook Pro running the Web browser peer (observed by the controller as the most powerful machine available). This operator migration between the two happens every 5000 messages as the event triggers throughout the experiment.

The same can be observed in the failure scenario experiment, where the two peers keep exchanging the operator. Once we crashed one machine, we made it immediately available afterwards, that is how the controller always picked one of the two MacBook Pros. The choice of the global controller among the whole pool of machines available always falls on the MacBook Pro available, as it is the most powerful machine in the pool. Also in this case we notice message loss as the topology runs and experiences the scheduled failures.

7.4 Elastic Parallelisation Experiment

In this Section we aim to show how well the local controller in the Web server peers is able to parallelise the execution of an operator. We decided to run an experiment where a CPU-intensive operator is running on a very powerful machine. In this way the controller can parallelise the execution of the operator without offloading, and we can measure the impact of a mutating workload on the topology in terms of number of workers (that is, resource consumption), latency of the messages, and the overall throughput. This Section offers an extended result of the experiments depicted in [BGP15a].

To do so, we reused the topology shown in Figure 7.1 and the dataset we employed in Section 7.3 but changed the throughput rate. Instead of reusing the throughput measured with the Twitter Firehose and replicate it, we decided to send messages with two fixed throughputs alternating over time, and mea-

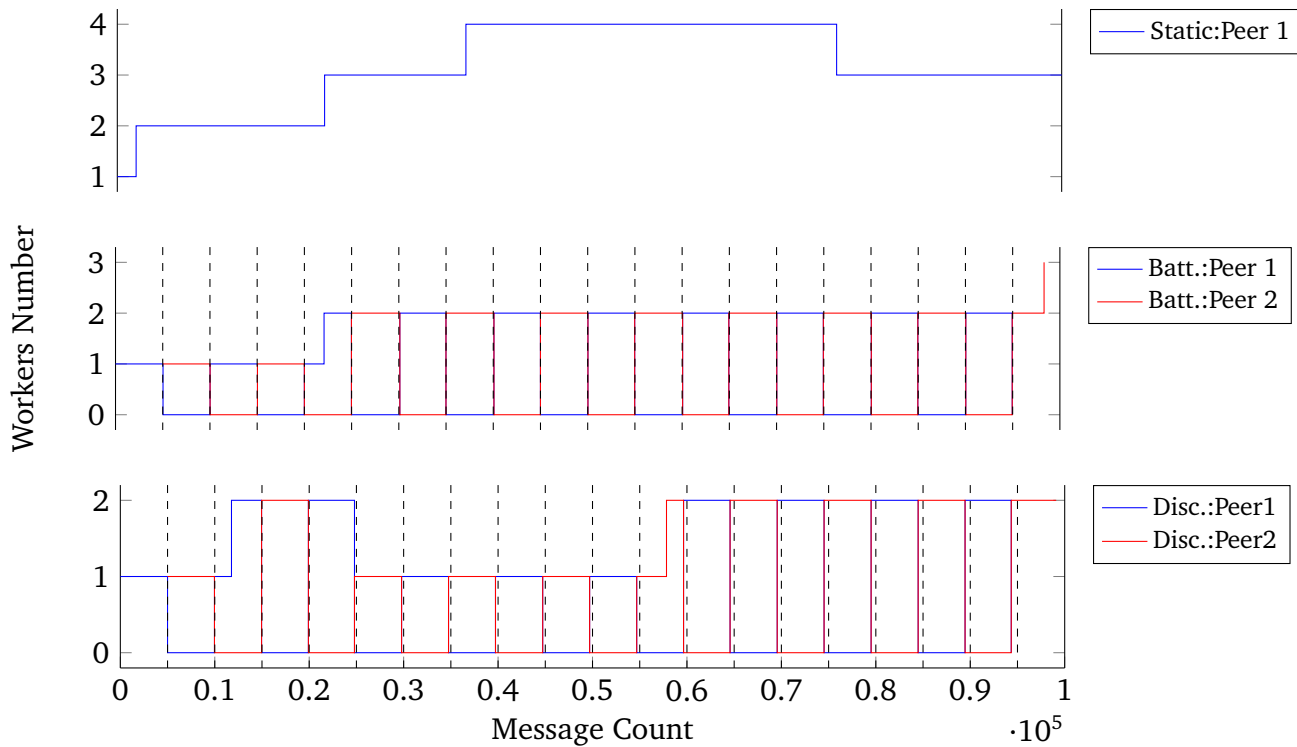


Figure 7.4. Number of workers throughout the three scenarios.

sure how the controller adapts the resource usage to the change of workload. We devised two different workloads: a light workload where the producer sends 40 messages per second, and a heavy workload, where the producer sends 500 messages per second. These two workloads are alternated in two different experiments. The first experiment alternates the two workloads every 5000 messages (slow workload transition), while the second one alternates the two workloads every 500 messages (fast workload transition).

To show the ability of the controller to dynamically parallelise the execution of an operator we opted for the following deployment. The producer runs on the MacBook Pro, the filter operator runs on the twenty-four cores machine, while the consumer runs on the four-cores server. The aim is to show how the controller is able to parallelise the execution, thus we picked the most powerful machine we had and forced the execution of the filter there.

Figure 7.5 shows the number of workers, the delay, and the throughput during the execution of the slow workload transition. The dashed vertical lines indicates the workload change. As the workload transitions from 40 messages

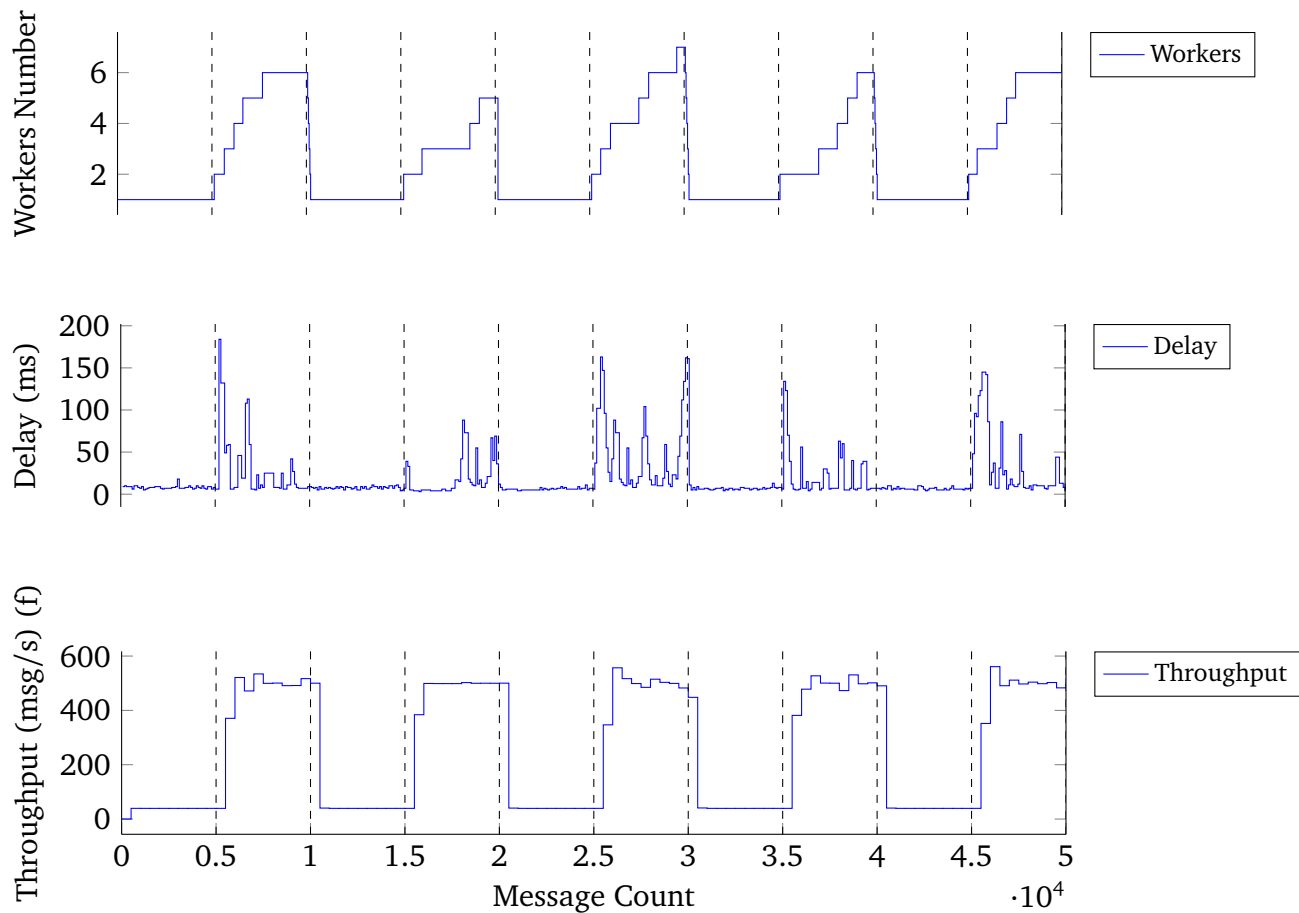


Figure 7.5. Parallelisation of the execution as the workload mutates every 5000 messages (slow).

per second to 500 messages per second we can see how the number of workers increases as the demand for resources increases. This is triggered by the local Web server controller which, by computing the request and response rate of the operator, adds workers. We can see how the delay spikes when the workload changes from slow to fast, caused by the lack of workers. As the controller adds workers to solve the bottleneck, the delay of the messages decreases and the throughput grows, reaching almost 500 messages per second, showing that the controller is reacting to the change in workload as expected.

Figure 7.6 shows the number of workers, the delay, and the throughput during the execution of the fast workload transition. From the plots we notice how the controller only manages to add a second worker to deal with the bottleneck

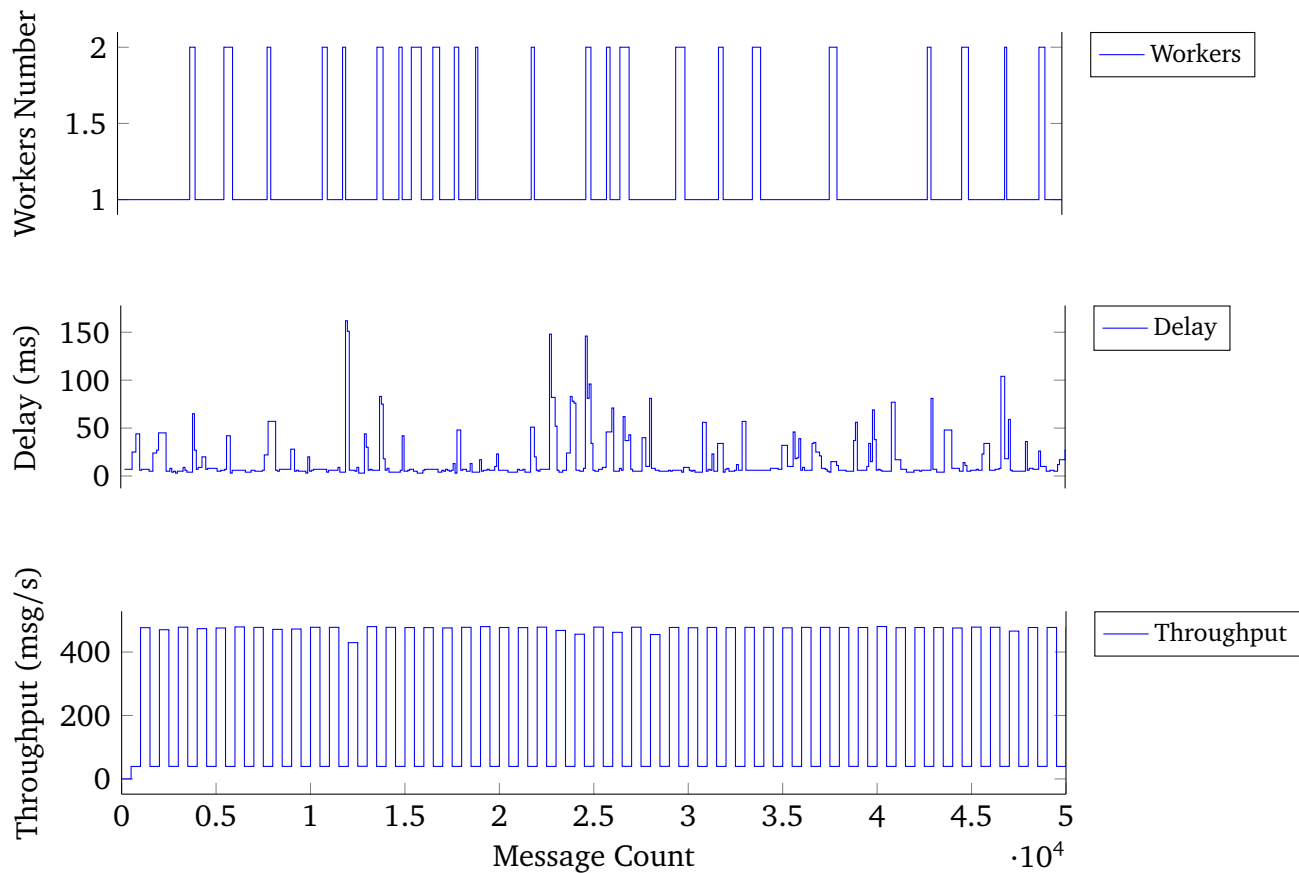


Figure 7.6. Parallelisation of the execution as the workload mutates every 500 messages (fast).

before the throughput goes back to 40 messages per second. The reaction is not fast enough to add the same amount of workers added in the previous experiment. Despite this the topology manages to maintain a similar delay and throughput.

Figure 7.7 shows two sets of box plots picturing the throughput in the two different experiments, both in the slow workload and the fast workload. The slow transition experiment shows to be able to maintain the throughput of the producer (40 messages per second in slow workload, 500 messages per second in fast workload), by achieving 39.4 messages per seconds as a median during the slow throughput and 499 messages per second as a median for the fast throughput. The fast transition experiment shows a good 39.5 messages per second as a median in the slow throughput, but it is not able to go past 477 messages per

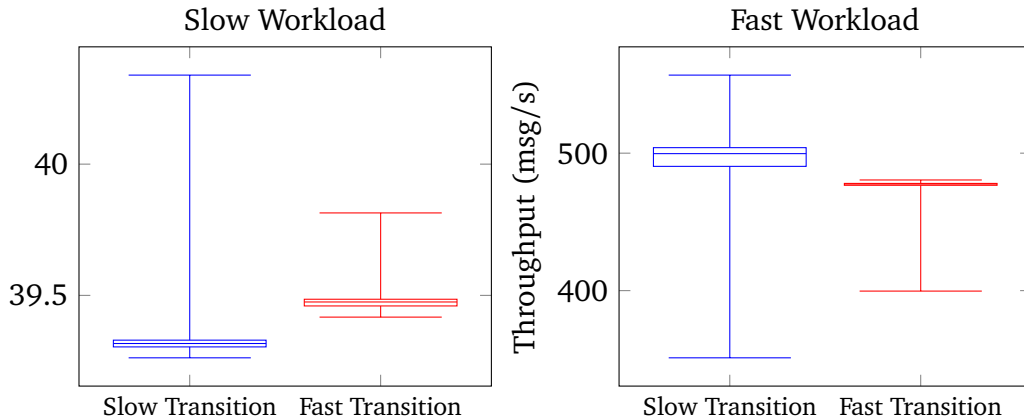


Figure 7.7. Throughputs for the two different workloads in the two experiments.

second as a median value in the fast transition. The controller is not able to add enough resources to face the fast transition, only adding one worker most of the times as shown in Figure 7.6, thus reducing the filter throughput in the fast workload.

7.5 Global Controller Algorithm

This experiment extends the work shown in [BGP15b], where we analyse how the decisions of the global controller regarding the liquid deployment of operators affect the performance of a topology in terms of latency of the messages, and how this impacts the resource consumption on the available resources.

We once again reused the topology we employed in Section 7.3, but doubled the throughput of the topology (from around 80 messages per second to around 160 messages per second) in order to stress the filter and see how the initial deployment of the streaming operators impacts on the resource usage. A good deployment should make use of as few resources as possible while keeping the throughput as high as possible. A random deployment, on the other hand, could make use of more resources by initially deploying on any available resource, and possibly ending up parallelising on other devices. We compare our ranking algorithm (described in Section 5.3) with four random deployments and an inverse ranking function, always taking the worst available peer.

Figure 7.8 shows a stacked bar chart illustrating the resource usage per peer, per number of workers, and the end-to-end median latency for each run of the experiment for six different configurations. We can see how the ranking approach

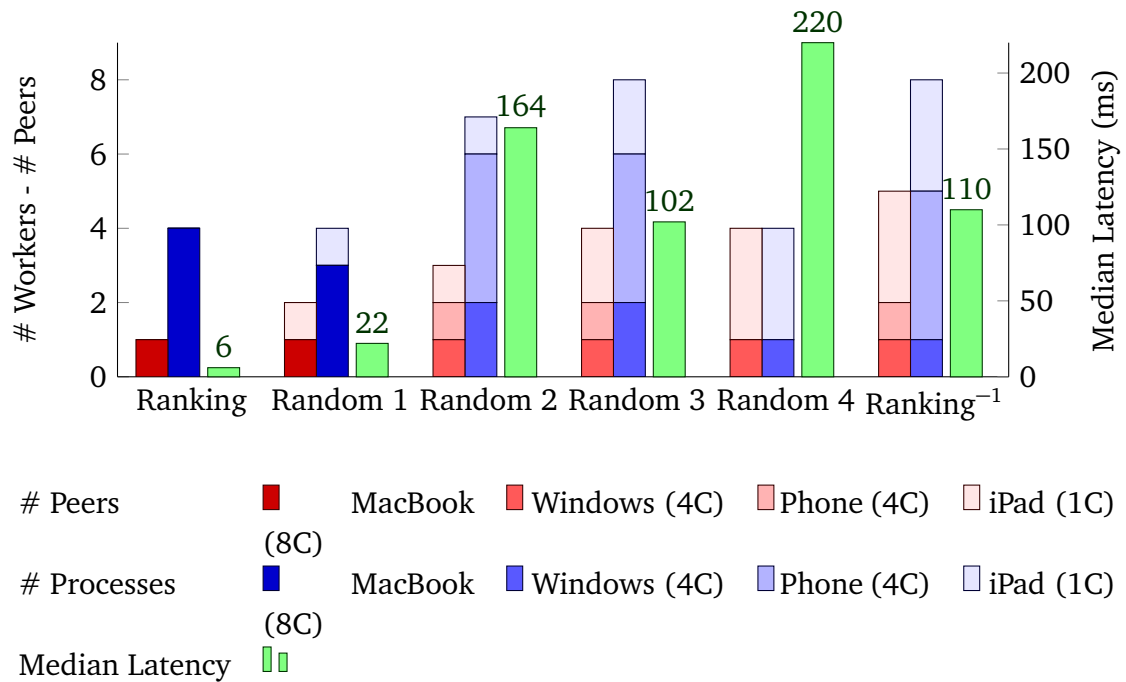


Figure 7.8. Liquid deployment: comparison of random vs. ranked resource allocations and their median end-to-end latency.

in deploying the filter operator takes into consideration the availability cores availability and deploys on the strongest machine, being the MacBook Pro featuring 8 threads (4 cores virtualised). The MacBook pro gets up to four workers to deal with the incoming load and maintains a low median latency. By randomly deploying the filter operator, we can see how receiving peers become overloaded and ask for help. Random 1 is deployed first on the iPad, and then the controller randomly picks the MacBook Pro when the iPad is overloaded. The Mac spawns three workers and manages to keep the latency low. This is not the case for the rest of the random controller experiments, in which the controller does not pick the MacBook Pro at all, and forces the computation to spread on more than devices. These small devices cannot keep up with the workload, causing the topology to experience higher end-to-end latencies.

The results of this experiment suggest that even when dealing with peers of different nature, in most cases using the number of CPUs as a rough estimate of the computing power of a device helps the controller ranking the peers.

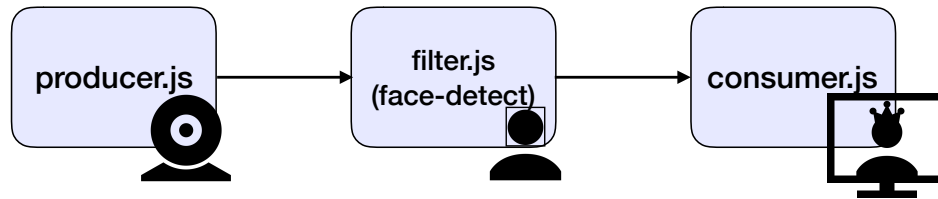


Figure 7.9. Face detection and decoration topology employed for the experiment.

7.6 Web Browser Local Controller and Fine-tuning

In Chapter 5 we introduced the controller and illustrated the differences between the local controller in the Web browser and the one running in the Web server operators. The differences mostly rely in the infrastructure offered by client-side JavaScript that forced the development of ad-hoc infrastructures such as the message queue. In Section 5.3.3 we mentioned three main variables inside the Web browser local controller: the controller cycle (which is available in the Web server controller too), the CPU usage threshold, and the slow mode thresholds (one to trigger it, one to release it).

In this Section we introduce how we fine-tuned the Web browser local controller in order to deal with a certain class of streaming applications [Bab17].

The experiment features a three-staged linear topology fully deployed on Web browsers. The topology is shown in Figure 7.9. The producer acquires the webcam feed taken from the user's webcam and forwards it to a face-detecting filter which, after detecting faces in the received snapshot, applies a decoration on the faces found. The resulting image is forwarded to the consumer which displays it. The idea of the experiment is to stress the filter and trigger parallelisation while modifying parameters in the controller configuration. Through empirical results we then study the impact of those changes on throughput and delay.

Table 7.2 shows the configuration of variables we tuned for this experiment. Configuration 1 is the one we used throughout all the experiment presented in this Chapter, unless explicitly stated. Configuration 2 reduces the controller cycle timeout, making it more reactive to changes as it checks on the operators more often. Configuration 3 keeps the change of cycle introduced in configuration 2 and halves the threshold to offload and parallelise operators execution.

While on the one hand this does not use the machine at its fullest, it leaves some free resources that could be used to further parallelise the execution, while already moving part of the computation elsewhere – if possible. Configuration 4 keeps the changes introduced in configuration 2 and 3 and halves the slow mode thresholds, becoming more greedier on the resources used. Instead of waiting for the queue to reach 20 messages, it already triggers the slow mode at 10 messages in the queue, halving the time taken for the operator to execute messages in the queue and remove the slow mode. We expect to see that for use cases in which the throughput is high and the computation takes long, the controller is able to parallelise faster and tries to solve queues faster by redirecting the traffic momentarily from one peer to another.

We decided to deploy the producer and the consumer on a single machine, peer 3, while we used peer 1 (P1) and peer 2 (P2) as free machines where the WLS runtime can run and eventually offload the computation of the filter. In every run of the experiment we let WLS pick the peer where to deploy the filter operator, which always ended up being peer 2, the most powerful available peer. This nonetheless lead to parallelisation on the weaker machine during the experiment runs.

We decided to send a total of 6000 messages at three different rates: 6 messages per second, 10 messages per second, and 13 messages per second. The size of the webcam image is fixed to 400x300 pixels, which are converted into messages to be sent, for a total weight of about 800Kb per message. We used the WiFi (802.11n, 300Mbit/s) to simulate a normal use case scenario environment.

Figure 7.10 shows the impact on end-to-end latency and queue size of the four controller configurations in the 6 messages per second scenario. The graph shows the percentage of messages executed on the y-axis and the delay of the messages on the x-axis. We can see how configuration 1 (C1) shows some delay with respect to the other configurations, this can be seen reflected on the queue size plot, where C1 shows a longer queue in P1, which brought the need to

Table 7.2. Controller configuration parameters

| Tuned Parameter | Config 1 | Config 2 | Config 3 | Config 4 |
|------------------|----------|----------|----------|----------|
| Controller Cycle | 500ms | 300ms | 300ms | 300ms |
| T_{CPU} | 100% | 100% | 50% | 50% |
| T_{qh} | 20 | 20 | 20 | 10 |
| T_{ql} | 10 | 10 | 10 | 5 |

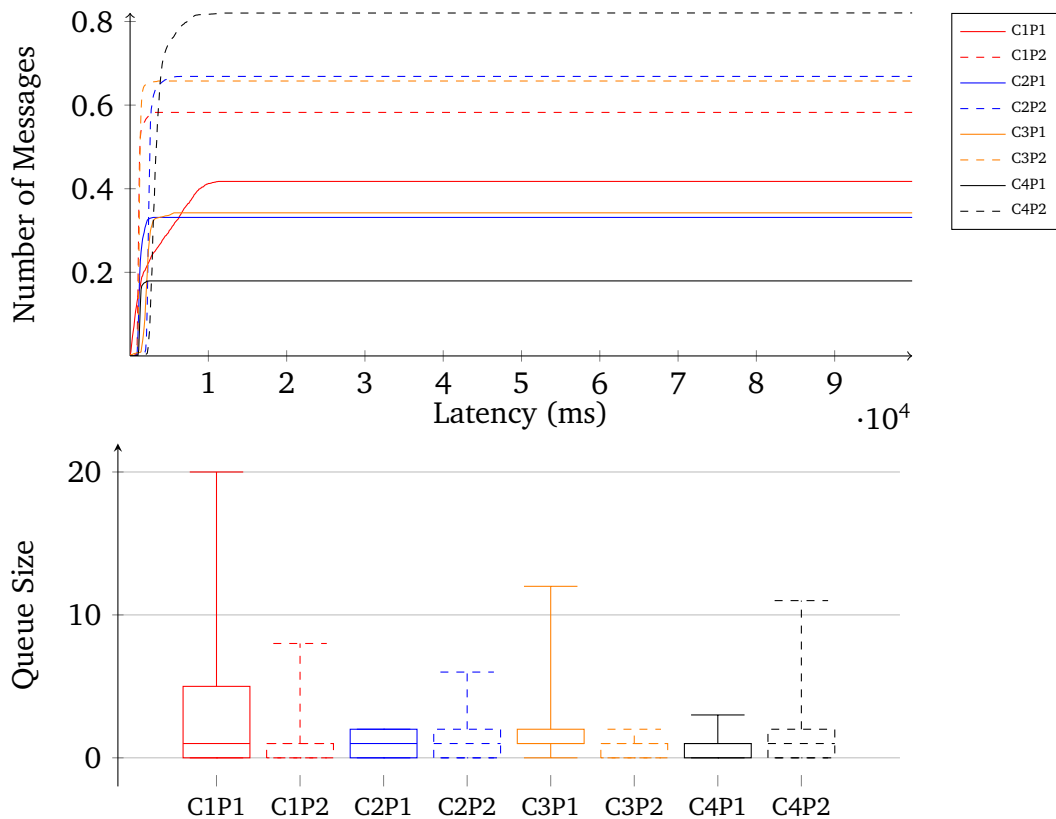


Figure 7.10. Message latency and queue size distributions per peer running on different controller configuration with 6 messages per second.

trigger the slow mode. This helped solve the long queue, which also caused the messages to experience latency.

In Figure 7.11 we can see the result of the experiment sending 10 messages per second. We can see how the performance in terms of delay degrades for C1 and C3, while C2 and C4 seem to be able to deal with the increased throughput, in fact both P1 and P2 in both cases maintain a lower delay for most of the messages received than the same peers in C1 and C3. If we look at the queue size we can see the delay reflected on the number of messages in the queues: C1 and C3 show longer queues, with respect to C2 and C4 that manage to keep the queue much smaller. This is because while on one side C1 notices late the increase of resource demand by the filter operator, on the other side the controller triggers a parallelisation soon enough, but doesn't get to the threshold to trigger the slow mode, thus P1 keeps receiving and queueing work, increasing the delay of its messages executed. C2 manages to deal with the workload by using 100% of the CPU before triggering a CPU overload, which results in a better

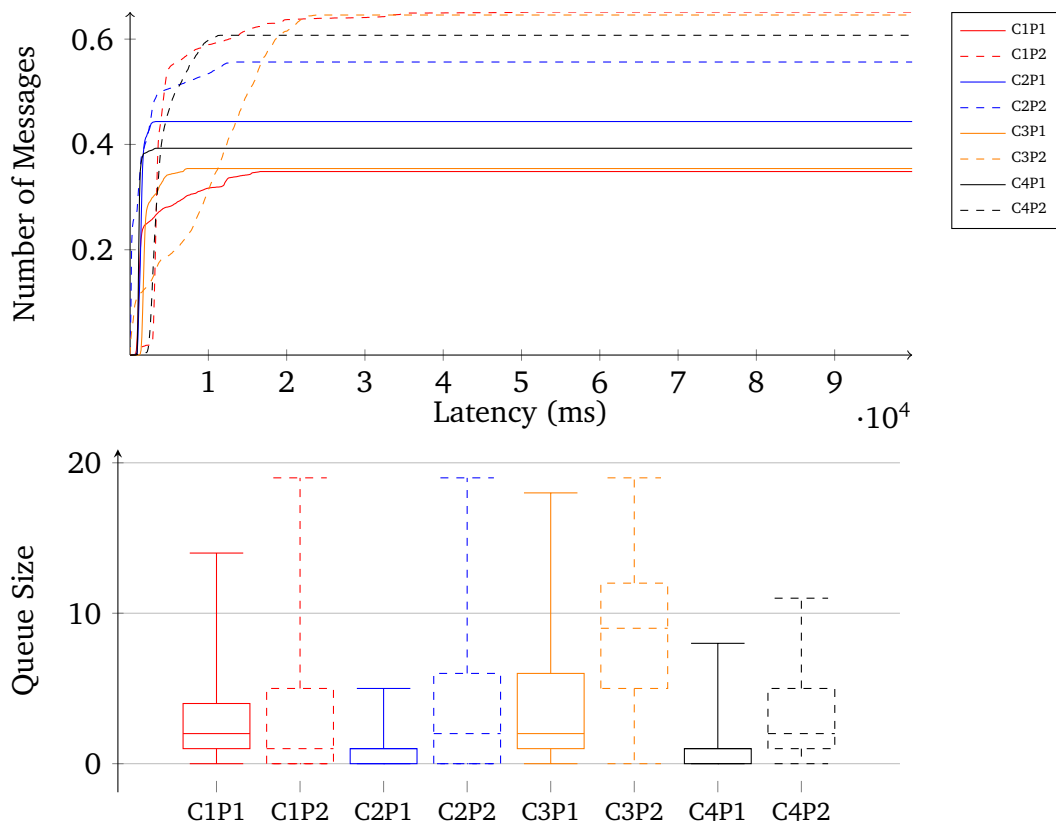


Figure 7.11. Message latency and queue size distributions per peer running on different controller configuration with 10 messages per second.

use of the resources, while C4 triggers the CPU overload request but unlike C3 it triggers the slow mode earlier (shown in the queue sizes plot), being able to redistribute the work and deal with the workload without impacting as much as C3 on the delay.

Figure 7.12 shows the results of the experiment with a producer throughput of 13 messages per second. The delay shown in the four configurations suggests that C4 can handle a higher throughput better than the other configurations, maintaining less latency overall. Both curves in fact show a better performance than the others, in which either one peer or the other exhibit latencies in message execution. C1 does not react soon enough, parallelising late and resulting in high latencies, and less messages executed by P1 (the target of parallelisation). C2 has a faster controller cycle but waits until 100% of the CPU is used. This results in longer queues, and thus longer latencies. C3 also reacts fast and is able to parallelise sooner, keeping queues and latencies lower. The same applies to C4 which maintains even lower latencies by triggering and releasing the

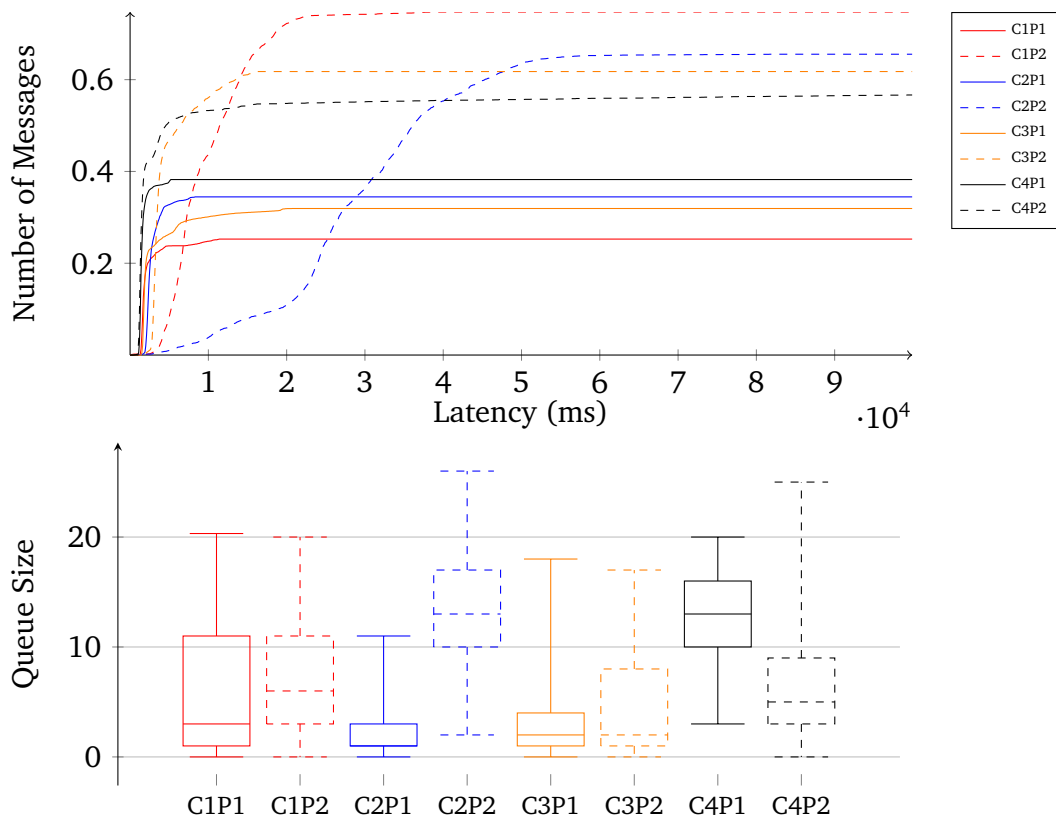


Figure 7.12. Message latency and queue size distributions per peer running on different controller configuration with 13 messages per second.

slow mode sooner. This results in slightly longer queues as the slow mode is triggered sooner, but also in an improved latency and overall parallelisation. In fact, by triggering the slow mode more often, we can see that the difference in the number of messages executed by the two peers is narrower.

Figure 7.13 shows the throughput differences for the four configurations in the three analysed scenarios. As we increase the throughput of the topology, measured in kilobytes per second in the plot, we notice how the performances of C1 and C2 drift, while C3 and C4 maintain a slightly higher throughput. The trend in general drifts from the optimal throughput, indicating that we are reaching the machines maximum capacity in terms of CPU power.

The three runs with the four different controller configurations show that a more reactive and more resource-aware controller is able to deal sooner with CPU-intensive operators running in topologies with high throughput. While maintaining a good throughput, more reactive controllers are able to save in latency and keep relatively short queues through a more sensible flow control policy.

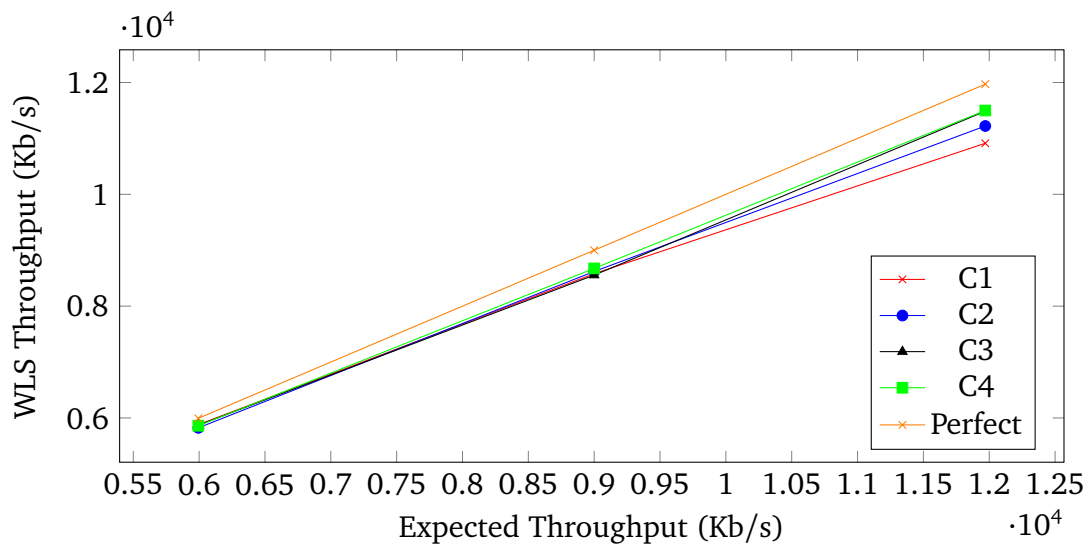


Figure 7.13. Throughput of the topologies in the three experiments with the four controller configurations.

This approach requires more effort from the WLS-runtime to constantly adjust the data flow when in slow mode, and by keeping a low threshold on the CPU usage, some use cases may spread the computation unnecessarily.

7.7 1 Failure

In Section 7.3 we analysed how the local and global controller cooperate to migrate operators and recover computation from disconnections. In this Section and the next one we aim to induce crashes in one or more peers at runtime while not bringing them online afterwards, showing how the system gracefully degrades while analysing the impact of the crashed machines on the overall throughput, end-to-end latency, and queue sizes.

We reused the topology presented in Section 7.6, but we decided to lower the throughput and send a message every 250ms (4 messages per second), and only use the four tablets as platform for the runtime to deploy the execution of the filter. This time we run the experiment for 4400 messages.

Figure 7.14 shows the oscillation in the number of workers throughout the experiment. We can see how the workload is immediately spread across all the available devices, and how the controllers try to keep the total amount of workers throughout the topology the same after peer 1 crashes. This is the result of the

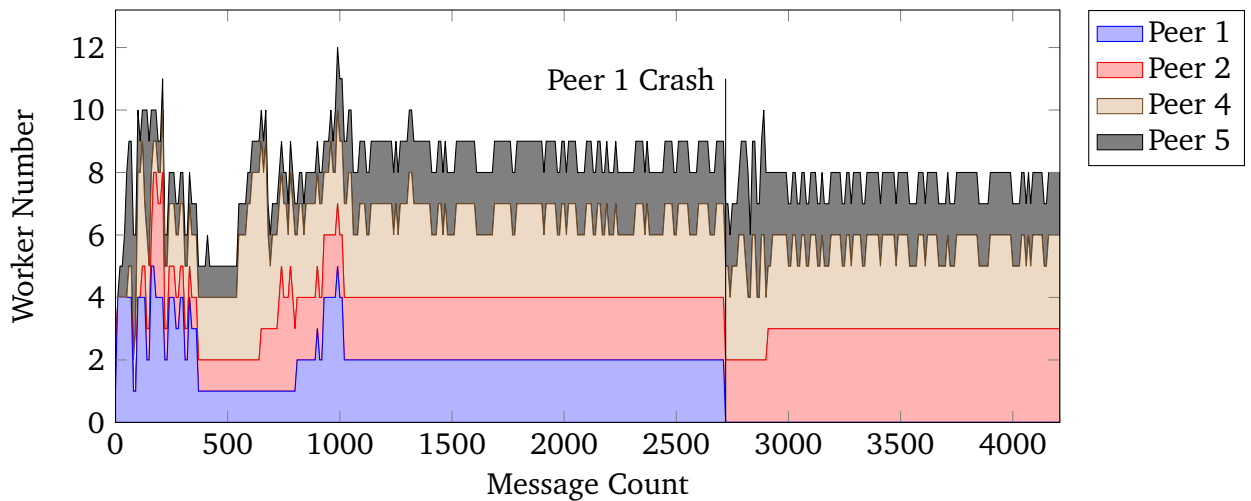


Figure 7.14. Oscillation in the number of workers throughout the experiment with only four tablets as filter deployment.

increased effort required by the topology as the workload is rebalanced on the remaining peers.

Figure 7.15 shows the end-to-end latency of the messages passing through the four tablets. We can see the effect of the crash of the first peer on the latencies experienced by the messages passing through the available peers, especially on peer 4 which has a small spike given by the increased workload received. Overall the topology appears to be able to deal with the increased workload, as the latencies just slightly increase.

Figure 7.16 shows the queue sizes throughout the experiment. Once again we can see how the crash happening in peer 1 effects the queues of the remaining peers, especially peer 4 and peer 5, without being too taxing – unlike the previous experiment. The reduced workload, more tailored with the given resources, does not impact so heavily on the remaining peers after a crash.

7.8 N-Failures

In this Section we induce more than one failure at runtime and study the behaviour of WLS in these circumstances. Again, we reused the topology presented in Section 7.6 (13 messages per second, but 7500 messages in total), this way we could use the Galaxy Tabs Web browsers and stress them with the face detection and image editing. Once again we used a single machine (peer 3) to act as producer and consumer, and deployed the filter on the available machines. We

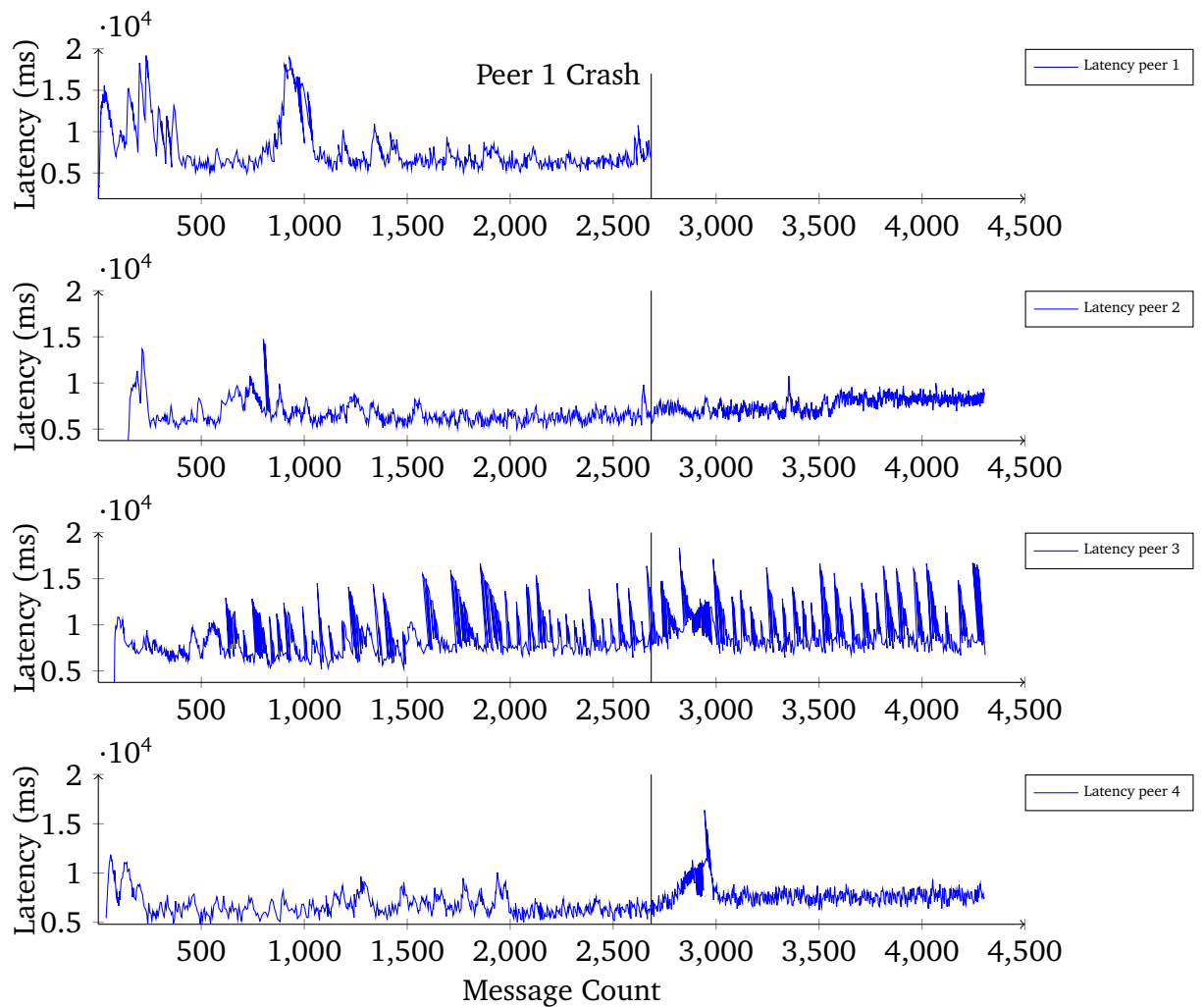


Figure 7.15. End-to-end latency for the four tablets.

let the WLS runtime pick the best machine where to deploy, always resulting in peer 1. We then observe how peer 1 and the WLS runtime liquidly spread the computation and face crashes happening on some of the Galaxy Tabs due to the excessive stress. We run the experiment for 7500 messages.

Figure 7.17 shows the resources allocated by the WLS runtime per device, and the crashes happened during the execution. The plot shows the number of workers in each peer at a granularity of 10 messages per measurement. We can see how after an initial deployment on peer 1 and peer 2, the runtime starts exploiting the tablets. At around 1800 messages through the topology we can see peer 4 crashing, then peer 2 at 4000 messages, peer 5 at 4800, and finally peer 7 at around 7400 messages.

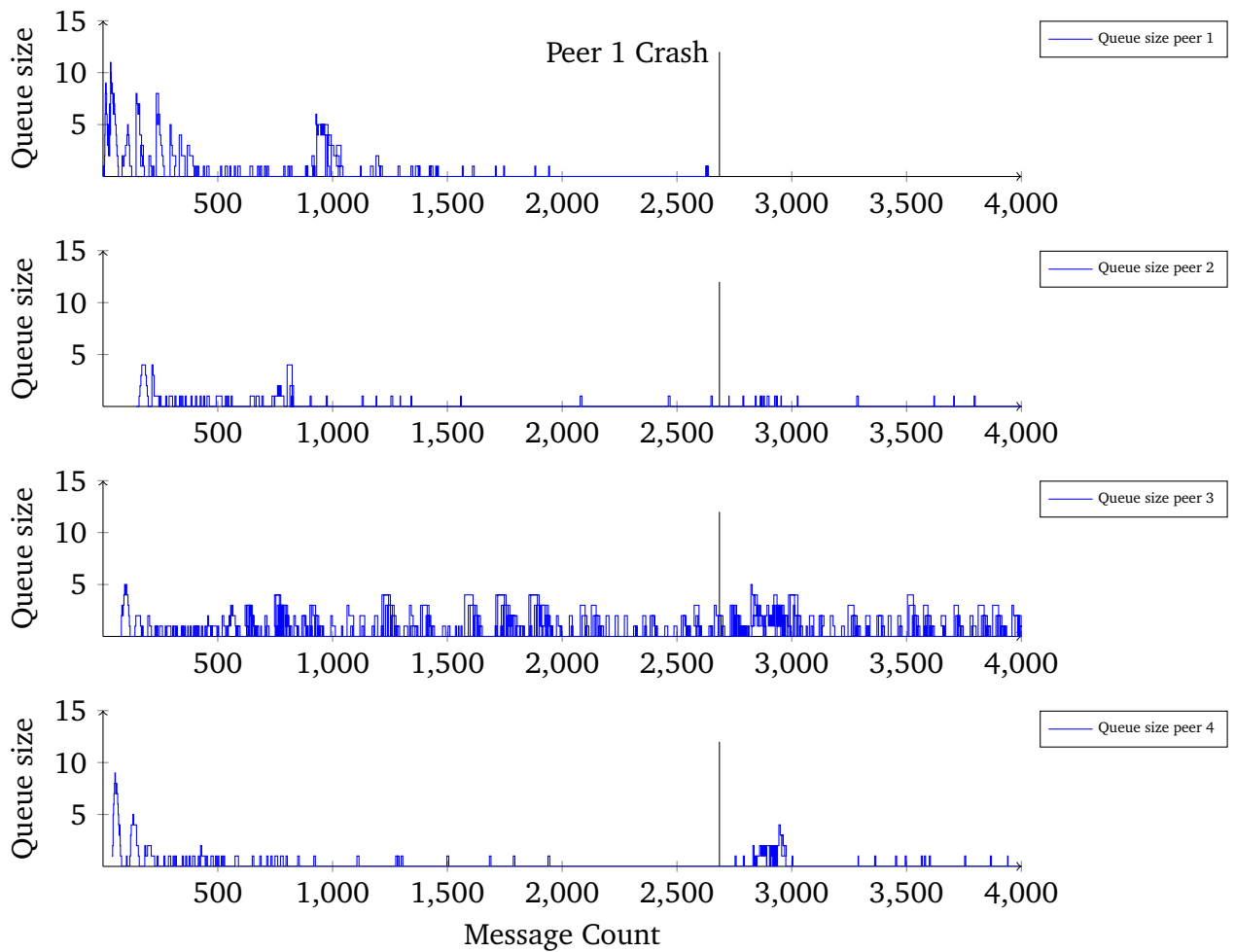


Figure 7.16. Queue size shown per peer.

We can see the effect of these crashes on the end-to-end latency in Figure 7.18. The Figure shows how crashes (peer 2, 4, 5, 7 – second plot) increase the latency of the messages processed by the peers which are still up and running (peer 1, 6 – first and third plot). Peer 1 deals with the crashes in a better way as it has more resources to cope with the increased workload, while peer 6 start slowly to struggle as more and more devices crash, ending up with accumulating long delays because of the increased workload.

Increased latencies can be associated with long queues. Figure 7.19 shows how the queue sizes change as the topology experiences crashes. As the various peers crash, we can see how the queue sizes increase on the remaining peers, growing more and more as crashes happen, resulting in queues with more than

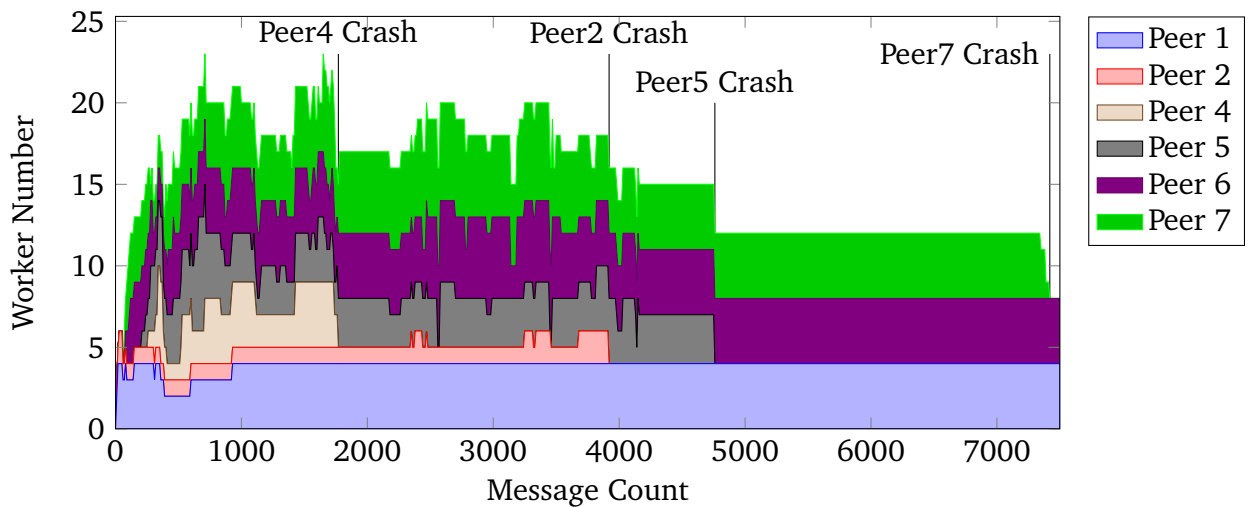


Figure 7.17. Oscillation in the number of workers throughout the experiment.

100 elements for the remaining peers. Peer 1 is able to deal with messages faster, keeping the queue at around 100 messages towards the end of the experiment, while peer 6 struggles more as shown in the curve on the plot, which goes beyond 100 messages and keeps growing linearly.

The experiment shows that the WLS runtime with the help of the controller is able to use the available resources when a (large portion of) the system crashes, keeping the data stream running through the topology. The performance degrades drastically in terms of end-to-end latency, but the topology is still able to process messages relying on the available peers.

7.9 Summary

In this Chapter we have shown the evaluation we performed on Web Liquid Streams. The system we built is able to deal with a dynamic, heterogeneous, and volatile environment like the Web. We have shown how we deal with low-battery levels in battery-powered devices by migrating the computation on other devices with the same capabilities and with a higher battery availability. In this way, battery-powered devices owned by the users may be used to deploy parts of a streaming application while the runtime deals with the migration when battery levels drop under a certain threshold. Disconnection recovery is also another important factor when dealing with the Web. If a user accidentally closes the Web browser, unplugs a sensor, or loses signal on his device the topology could be dis-

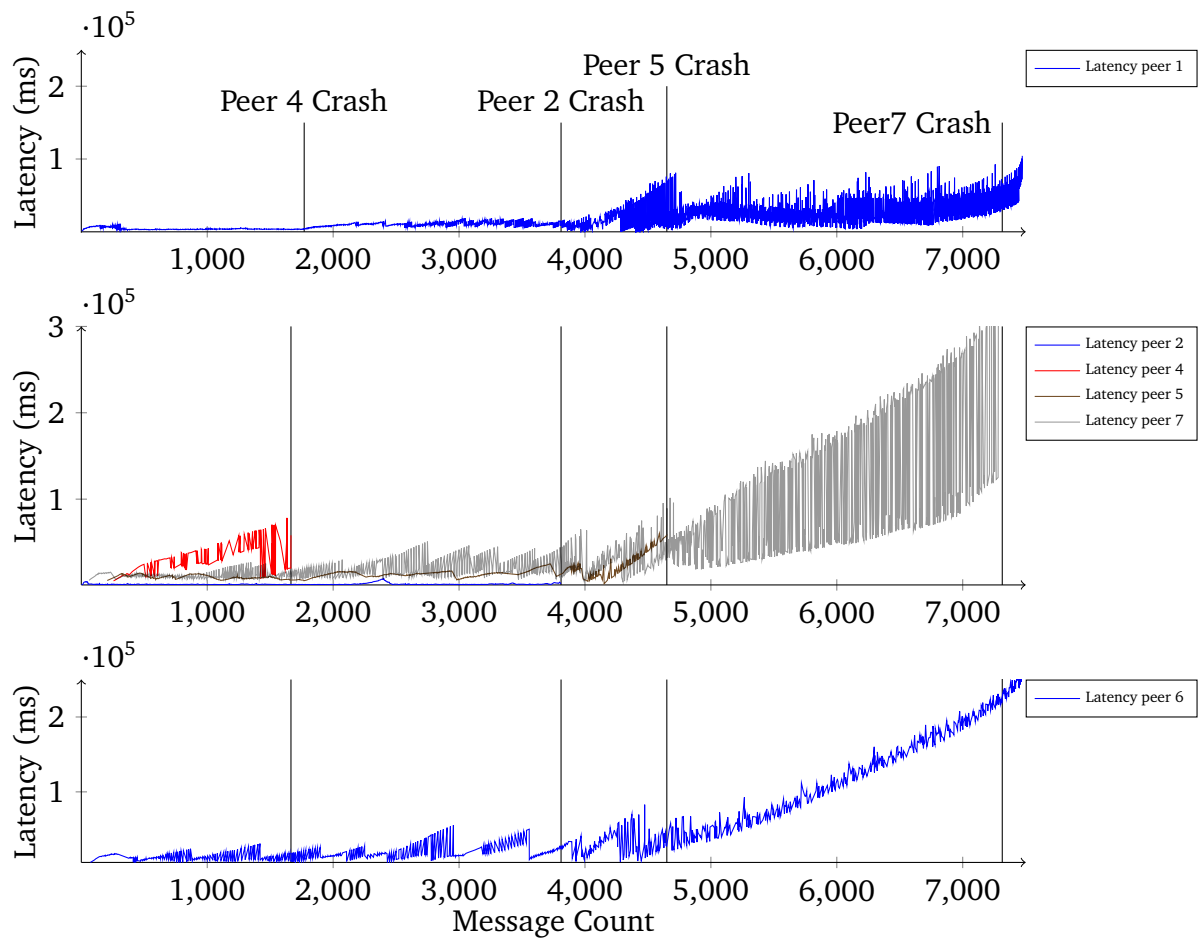


Figure 7.18. End-to-end latency shown per peer.

rupted. We have shown how our runtime is able to keep the topology running by instantiating copies of the lost operators on other available peers, and how well that happens by confronting the throughputs in a static scenario, a low-battery level scenario, and a disconnection-recovery scenario.

Users may modify the topology at runtime to solve operators bugs, or to slightly change the semantics of the application (by for example adding new sensors as producers in a topology and patching a filter to store the newly produced data). These changes at runtime may impact on the data flow, which can unpredictably oscillate and that may cause bottlenecks. We have shown how well our framework adapts to workload oscillation by elastically modifying the number of workers deployed in an operator. The number of workers increases and decreases following the CPU demand of the workload, in this way it allocates just the right amount of workers needed for the computation, and removes

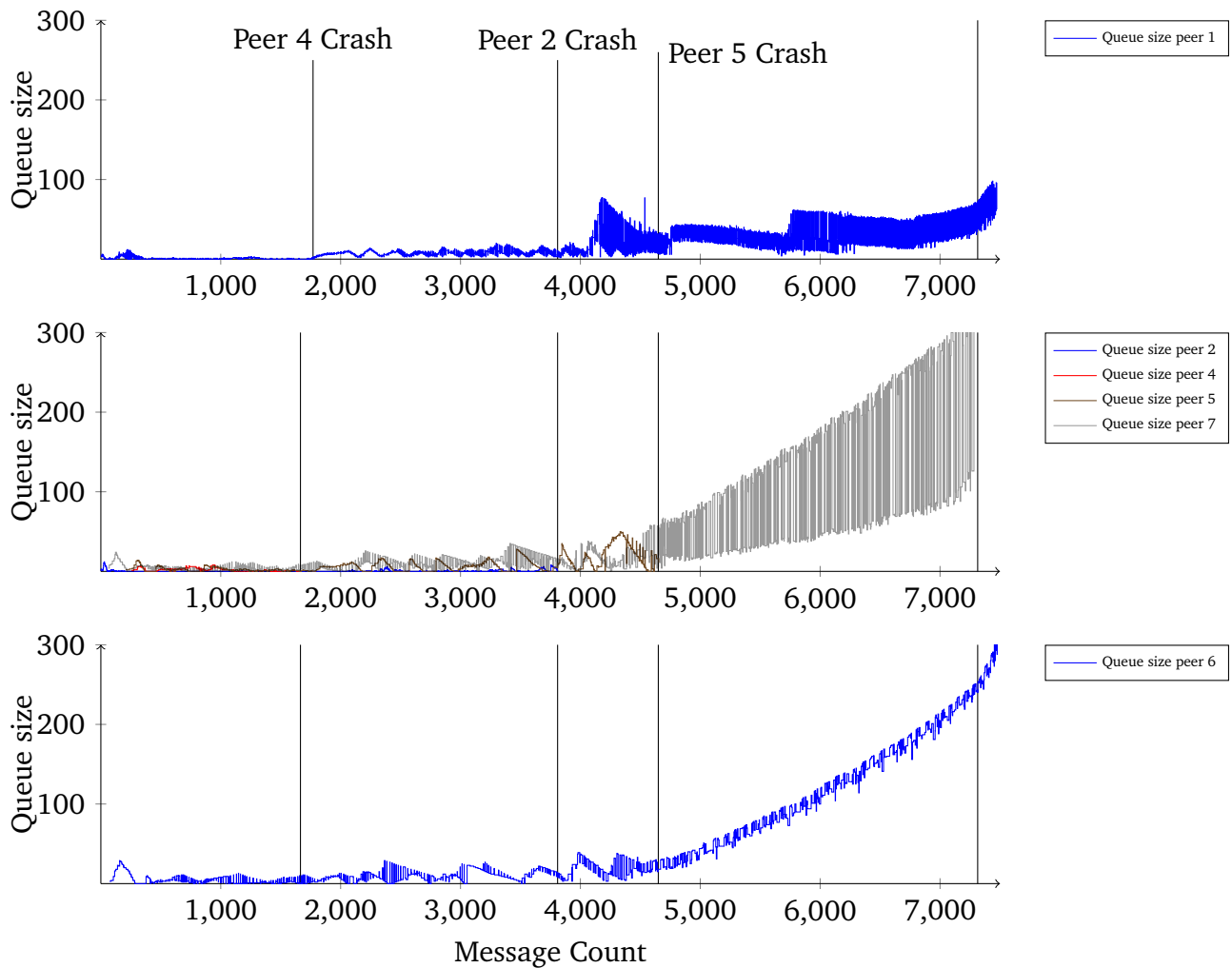


Figure 7.19. Queue size shown per peer.

unnecessary workers when the resource demand decreases. This avoids resource consumption that may impact on the battery of the devices.

Web Liquid Streams abstracts the complexity of dealing with the deployment of streaming operators and communication channels from the users. Our runtime is able to deploy streaming topologies on the available devices by the means of a ranking function which ranks the available devices. By using the number of CPUs as a rough estimate of the computing power of a device, the runtime gives the best possible deployment keeping into account the operators deployment needs (i.e., sensor-based operators).

We have studied how to improve the Web browser controller, making it more resource-aware and more reactive. The studied configuration fits streaming applications that have CPU-intensive operators running in topologies with high throughput. We have shown that a more sensible control flow policy is able to save in latency and keep queues relatively short, despite making use of more resources and possibly spreading the computation on other available devices when not strictly needed.

Finally, we have shown how WLS is able to make use of the remaining devices when a failure happens. When a single failure happens, the runtime is able to use the connected devices to run the lost operators, even if no free device is available. This gives time to the user to notice the failure and connect more resources to the runtime while WLS guarantees that the stream continues to flow. The same holds when more than one device fails, and the computation is forcefully spread on busy devices. The runtime tries to keep the topology alive, even though this drastically impacts on the end-to-end latency.

Part IV

Epilogue

Chapter 8

Conclusion

Recently proposed Web technologies such as WebSockets and WebRTC made the Web a mature platform to run distributed applications, connecting and orchestrating Web-enabled devices of different nature in a single cloud of computing devices. The hosting platform of such applications could be user-owned desktops and laptops, big Web servers, but also microcontrollers and single-board PCs. These devices are growing in number and shrinking in price, becoming an affordable playground for programmers and Makers – people that do not have a background in computer science, but enjoy experimenting with technology and like to apply the Do It Yourself approach.

While the development of small applications on microcontrollers and single-board PCs may be simple even for a Maker, more complex applications involving more than one device with different nature may become a tougher task. For example, integrating a Web server in a distributed application to host a Web monitor for a home automation system, or storing and processing streamed environmental data on a privately owned laptop may be a complicated task for an unskilled developer which should deal with different environments, different programming languages, and different communication channels. A distributed application could also suffer of bottlenecks in the execution, and failures that should be dealt with to keep it up and running unattended over long periods of time.

In this dissertation we presented the Web Liquid Streams framework. WLS offers developers a JavaScript programming environment for streaming Web applications. Developers do not have to worry about running applications on devices of different nature: with WLS streaming operators can be developed in JavaScript, while the runtime takes care of dealing with hardware differences thanks to Node.js and the Web browser. The WLS runtime takes care of the data

channels as well, hiding from the developers the complexity of dealing with hardware and network heterogeneity. By following the principles of the liquid software architectural style, WLS offers full code mobility, letting developers move the application code transparently from device to device. Dealing with bottlenecks and failures is done by the controller, a runtime component which purpose is to keep the topology up and running and solve bottlenecks despite device failures during the execution.

We surveyed the state of the art in stream processing frameworks, languages, and tools. The results suggested a growing interest in sensor data and microcontrollers, being the target of some of the surveyed systems. Last but not least, given the introduction of WebRTC to stream data between browsers without passing through a server, and the growing number of Web-enabled devices, we expect to see more pure browser-to-browser streaming applications.

Next, we presented the Web Liquid Streams framework from the developer's perspective. We described the terminology we use in describing streaming topologies, and the liquid abstraction we use to describe streaming applications developed with WLS. The proposed API lets developers build streaming topologies by implementing operators that are called whenever a stream message arrives. Developers can also modify the structure of the topology within the operator code, as well as use a database through stateful operators. Browser-related methods only work on Web browsers and extend the functionality of an operator by adding GUI methods. Developers can include HTML and CSS inside the operator code to build a GUI to, for example, monitor and visualise their streams. Through the command line interface of WLS, developers are able to create and change streaming topologies on the fly; by creating a JSON topology file instead, developers can statically describe a topology by specifying operators and bindings. Such file can then be fed to the runtime which takes care of deploying the operators on the available devices and perform the bindings. By the means of a RESTful API external applications can interact with the WLS runtime as if developers were interacting with the command line interface. We have shown a monitor for WLS that uses the RESTful API to show a graphical representation of the topology, the operators, the bindings, and the workers.

We introduced the WLS runtime describing its two communication layers. The first communication layer, the command layer, is used by WLS to send and receive command messages (i.e., run an operator, bind an operator, add a worker, etc.) to and from peers, or the root. We have shown the RPC method calls that are used by the root to call procedures on the connected peers. We have then introduced the stream layer by describing the three different channels that WLS uses to stream data across streaming operators. For server-to-server communi-

cation we make use of ZeroMQ, for server-to-browser and browser-to-server we use WebSockets, while for browser-to-browser communication we use WebRTC. We introduced the minified Web server peer implemented to run on smaller and weaker devices, such as Tessel. The Web browser peer implementation includes modules to deal with the display APIs and with the HTML5 sensors APIs. While in the Web server peer, the operator pool is an array of objects, in the Web browser peer it is a more complicated infrastructure that has to deal with the deficiencies of the browser's sandboxed environment. Inside both operator infrastructures, a worker pool stores all the workers running to parallelise the execution. The connection with Redis, the database of choice for WLS, happens server-side through an ad hoc module built to support database transactions.

To support less skilled developers to deal with bottlenecks and different kinds of failures in the execution of the topology, we implemented a control infrastructure. The control is composed by two instances: a single global controller running on the root which checks the running topology and takes actions upon peer crashes and disconnections, and local controllers running on the peers involved in the topology that check on locally deployed operators to balance the load and avoid bottlenecks. The local controllers are connected with the global controller and query it for operator migrations and cloning when needed – for example, when the CPU usage is too high the operator is cloned on another machine, while if the battery level of a device drops under a certain threshold, the controller starts a migrating procedure to avoid losing the running operator. The global controller is also in charge of checking available peers for the automatic deployment of the topology upon receiving command line interface commands or a JSON topology file. By making use of the presented algorithms, the local controllers are able to deal with bottlenecks by autonomically increasing and decreasing the amount of workers deployed.

Web Liquid Streams has been used by different developers and with different purposes during the past few years. We used it as a mashup tool for the first Rapid Mashup Challenge 2015, scoring a third place and showing how WLS can be expressive and flexible enough to be used as a mashup tool. WLS has been the target of an experimental evaluation performed in the University of Applied Sciences in Karlsruhe, Germany, and also presented during the Inforte seminar on SW technologies and development for multi-device environments summer school in Tampere, Finland. During the summer school we let the participants get their hands dirty with the code and build their first topology integrating Raspberry Pi sensors, Web servers and Web browsers. We let middle- and high-school students play with WLS during the USI Study Week, where participants with no background in computer science tried coding for the first time. With a little help

with the programming language at first, the students managed to implement a simple topology to stream data from a Web cam and a sensor to a Web server, storing the data, and showing the video feed with the sensor data printed on screen on a Web browser. Through a project proposed in Software Atelier 3, second year bachelor students implemented a monitor for the USI open space, monitoring light and noise levels throughout the project weeks. The feedback we obtained at each deployment was very useful to improve the framework design and implementation.

Besides a formative evaluation, we also did a performance evaluation on WLS and its controller. The purpose of the evaluation was to test how well the controller works under certain conditions and further improve it. We studied the throughput, the latency, the queue sizes, but also the number of available peers and the number of workers spawned to deal with a bottleneck in different scenarios. We have shown how the controller deals with operator migration and disconnection recovery by illustrating three different cases (static, battery shortage, and seldom disconnections) and illustrating the throughput differences. The results showed only little differences in the three cases. We explored the parallelisation of the execution on a single operator, by adding and removing workers, showing how the number of workers adapts to the needs of the topology and does not consume more resources than needed during the execution. Finally, we evaluated the Web browser local controller by fine-tuning its variables and showing improvements of the controller decisions under certain conditions.

To conclude, in this dissertation we have shown how, thanks to emerging Web technologies and the liquid software metaphor, it is possible to abstract the complexity of the underlying hardware of Web-enabled devices and offer a robust JavaScript data stream framework with autonomic failure recovery and distributed execution on heterogeneous hardware. Through our implementation of the Web Liquid Streams framework, Makers and less experienced programmers are able to use their own machines to build and run complex streaming applications.

8.1 Future Work

The security of a topology implemented in Web Liquid Streams is an important topic which was left as a future work. Given the premises, the devices in a topology are all owned by a single user. The moment more than one user want to use a single streaming application, and URLs of the operators become public, we must find a way to hide sensitive data (home temperature, home energy con-

sumption, etc.) that may be streamed through a WLS topology. A solution for such issue may be adding a password verification when accessing an operator on a Web page. If the attacker doesn't know the password, he/she will never be able to access the operator execution – thus the sensitive data.

The WLS experience by Tobias Fuss at the University of Applied Sciences in Karlsruhe, with almost no JavaScript expertise gave us feedback on the difficulties while installing and using the system. While the programming API was not difficult to understand, in his experience the installation of the framework and the dependencies was difficult. As the framework is targeted for Makers that may not have experience in dealing with command-line installations, the easiest solution would be a single bundle that installs the framework and dependencies automatically with a single click.

Thanks to the controller, the WLS runtime is able to deal with failures by restarting lost operators on other available devices. If the failure happens on the root peer, where the global controller runs, it is impossible for the runtime to restart it. To solve this issue a leader election algorithm could be implemented to elect a new root peer which would then start another instance of the global controller. If the previous root comes back up online, the new root will inform it of its new role and downgrade it to normal peer.

In the very last experiment shown in Section 7.8 we noticed how the degradation of the topology (in terms of queue sizes and latencies) increases as the number of peers with permanent failure increases. To solve this issue we thought about implementing a load shedding protocol [GWYL05], which would help the topology reduce the load if there are no more peers where the stream can be parallelised across.

Bibliography

- [AAB⁺05] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryzkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005*, pages 277–289, 2005.
- [AAB⁺06] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. Spc: A distributed, scalable platform for data mining. In *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms, DMSSP '06*, pages 27–37. ACM, 2006.
- [ABB⁺03] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: The stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD '03*, pages 665–665. ACM, 2003.
- [ABB⁺13] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *The Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.
- [ABQ13] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems, DEBS '13*, pages 207–218. ACM, 2013.

- [ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.
- [ACGS11] Mohamed Ali, Badrish Chandramouli, Jonathan Goldstein, and Roman Schindlauer. The extensibility framework in microsoft streaminsight. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering, ICDE 2011*, pages 1242–1253, 2011.
- [Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
- [AGR⁺09] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Ying Li, V. Di Nicola, X. Wang, David Maier, S. Grell, O. Nano, and I. Santos. Microsoft cep server and online behavioral targeting. *The proceedings of the VLDB Endowment*, 2(2):1558–1561, 2009.
- [AGT14] Henrique C. M. Andrade, Buğra Gedik, and Deepak S. Turaga. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, 1st edition, 2014.
- [AHS06] Karl Aberer, Manfred Hauswirth, and Ali Salehi. A middleware for fast and flexible sensor network deployment. In *Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB '06*, pages 1199–1202. ACM, 2006.
- [And12] Chris Anderson. *Makers: the new industrial revolution*. Random House Business Books, 2012.
- [APC15] João Azevedo, Ricardo Lopes Pereira, and Paulo Chainho. An api proposal for integrating sensor data into web apps and webrtc. In *Proceedings of the 1st Workshop on All-Web Real-Time Systems, AWeS '15*, pages 8:1–8:5. ACM, 2015.
- [AT99] Enrique Alba and José M. Troya. A survey of parallel distributed genetic algorithms. *Complex.*, 4(4):31–52, March 1999.

- [Bab17] Masiar Babazadeh. Tuning browser-to-browser offloading for heterogeneous stream processing web applications. In *Proceedings of the 9th ZEUS Workshop, Lugano, Switzerland, February 13-14, 2017.*, ZEUS '17, 2017.
- [BAD14] Andrea Bellucci, Ignacio Aedo, and Paloma Diaz. Ecce toolkit: Prototyping ubicomp device ecologies. In *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces, AVI '14*, pages 339–340. ACM, 2014.
- [BBJN12] Adam Bergkvist, Daniel C Burnett, Cullen Jennings, and Anant Narayanan. Webrtc 1.0: Real-time communication between browsers. *Working draft, W3C*, 2012.
- [BCE⁺12] Cristina Băescu, Christian Cachin, Ittay Eyal, Robert Haas, Alessandro Sorniotti, Marko Vukolic, and Ido Zachevsky. Robust data sharing with key-value stores. In *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2012*, pages 1–12, 2012.
- [BGP15a] Masiar Babazadeh, Andrea Gallidabino, and Cesare Pautasso. Decentralized stream processing over web-enabled devices. In *Proceedings of the 4th European Conference on Service-Oriented and Cloud Computing*, volume 9306, pages 3–18. Springer, 2015.
- [BGP15b] Masiar Babazadeh, Andrea Gallidabino, and Cesare Pautasso. Liquid stream processing across web browsers and web servers. In *Proceedings of the 15th International Conference on Web Engineering, ICWE 2015*. Springer, 2015.
- [BHK⁺02] Andrew P. Black, Jie Huang, Rainer Koster, Jonathan Walpole, and Calton Pu. Infopipes: An abstraction for multimedia streaming. *Multimedia Syst.*, 8(5):406–419, 2002.
- [Bla15] Virginie Blancs. Introducing Tessel.io microcontroller into the Web Liquid Streams framework. Master's thesis, University of Lugano, Switzerland, 2015.
- [BMT12] Ozalp Babaoglu, Moreno Marzolla, and Michele Tamburini. Design and implementation of a p2p cloud system. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12*, pages 412–417, New York, NY, USA, 2012. ACM.

- [BP11] Daniele Bonetta and Cesare Pautasso. An architectural style for liquid web services. In *9th Working IEEE/IFIP Conference on Software Architecture*, WICSA 2011, pages 232–241, 2011.
- [BSM⁺06] Sara Bly, Bill Schilit, David W. McDonald, Barbara Rosario, and Ylian Saint-Hilaire. Broken expectations in the digital home. In *Extended Abstracts on Human Factors in Computing Systems*, CHI EA 2006, pages 568–573. ACM, 2006.
- [Can14] Mattia Candeloro. A RESTful API for controlling dynamic streaming topologies. Master’s thesis, University of Lugano, Switzerland, 2014.
- [CB74] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET ’74, pages 249–264. ACM, 1974.
- [CBB⁺03] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research*, CIDR 2003. ACM, 2003.
- [CBC⁺10] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys 2010, pages 49–62. ACM, 2010.
- [CcC⁺02] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: A new class of data management applications. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB ’02, pages 215–226. VLDB Endowment, 2002.
- [CFS⁺14] Ching Yu Chen, Jui Hsi Fu, Today Sung, Ping-Feng Wang, Emery Jou, and Ming-Whei Feng. Complex event processing for the internet of things and its applications. In *Proceedings of the 2014 IEEE International Conference on Automation Science and Engineering*, CASE, pages 1144–1149, 2014.

- [CGM14] Sven Casteleyn, Irene Garrigós, and Jose-Norberto Mazón. Ten years of Rich Internet Applications: A systematic mapping study, and beyond. *ACM Trans. Web*, 8(3):18:1–18:46, 2014.
- [CIM⁺11] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314. ACM, 2011.
- [CM08] Dave Crane and Phil McCarthy. *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*. Apress, 2008.
- [CM12] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, 2012.
- [CVHDVF09] Stefano Ceri, Frank Van Harmelen, Emanuele Della Valle, and Dieter Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24:83–89, 2009.
- [DA11] Michael Duller and Gustavo Alonso. A lightweight and extensible platform for processing personal information at global scale. *Journal of Internet Services and Applications*, 1(3):165–181, 2011.
- [DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
- [DGHN16] Linda Di Geronimo, Maria Husmann, and Moira C. Norrie. Surveying personal device ecosystems with cross-device applications in mind. In *Proceedings of the 5th ACM International Symposium on Pervasive Displays*, PerDis 2016, pages 220–227. ACM, 2016.
- [DJL15] Caroline Desprat, Jean-Pierre Jessel, and Hervé Luga. A 3d collaborative editor using webgl and web rtc. In *Proceedings of the 20th International Conference on 3D Web Technology*, Web3D '15, pages 157–158. ACM, 2015.
- [DLLW08] Robert F. Dickerson, Jiakang Lu, Jian Lu, and Kamin Whitehouse. Stream feeds - an abstraction for the world wide sensor web. In Christian Floerkemeier, Marc Langheinrich, Elgar Fleisch, Friedemann Mattern, and Sanjay E. Sarma, editors, *IOT*, volume 4952 of *Lecture Notes in Computer Science*, pages 360–375. Springer, 2008.

- [DP16] Florian Daniel and Cesare Pautasso. *Rapid Mashup Development Tools: First International Rapid Mashup Challenge, RMC 2015, Rotterdam, The Netherlands, June 23, 2015, Revised Selected Papers*. Springer, 2016.
- [DR07] Krista M. Dombroviak and Rajiv Ramnath. A taxonomy of mobile and pervasive applications. In *Proceedings of the 2007 ACM Symposium on Applied Computing, SAC '07*, pages 1609–1615. ACM, 2007.
- [DRAT11] Michael Duller, Jan S. Rellermeyer, Gustavo Alonso, and Nesime Tatbul. Virtualizing stream processing. In *Proceedings of the 12th International Middleware Conference, Middleware 2011*, pages 260–279. IFIP, 2011.
- [DW16] Audrey Desjardins and Ron Wakkary. Living in a prototype: A re-configured space. In *Proceedings of the 2016 Conference on Human Factors in Computing Systems, CHI '16*, pages 5274–5285. ACM, 2016.
- [FM11] I. Fette and A. Melnikov. The websocket protocol (proposed standard), 2011.
- [FMG⁺16] Ioannis Flouris, Vasiliki Manikaki, Nikos Giatrakos, Antonios Deligiannakis, Minos Garofalakis, Michael Mock, Sebastian Bothe, Inna Skarbovsky, Fabiana Fournier, Marko Stajcer, et al. Ferari: A prototype for complex event processing over streaming multi-cloud platforms. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD/PODS 2016*, pages 2093–2096. ACM, 2016.
- [FTE⁺17] Roy T. Fielding, Richard N. Taylor, Justin Erenkrantz, Michael M. Gorlick, E. James Whitehead, Rohit Khare, and Peyman Oreizy, editors. *Reflections on the REST Architectural Style and "Principled Design of the Modern Web Architecture"*, 2017.
- [Fus16] Tobias Fuss. Experimentelle evaluation des web liquid streams-framework. Short-term software development project on collaborative peer-to-peer Web browser communication, 2016.
- [Gal14] Andrea Gallidabino. Browser-to-browser Pipelines. Master's thesis, University of Lugano, Switzerland, 2014.

- [GBMP13] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Gener. Comput. Syst.*, 29(7):1645–1660, 2013.
- [GBP16] Andrea Gallidabino, Masiar Babazadeh, and Cesare Pautasso. Mashup development with web liquid streams. In *Proceedings of the 1st International Rapid Mashup Challenge*, RMC 2015, pages 98–117. Springer, 2016.
- [GHOS96] Jim Gray, Pat Helland, Patrick O’Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’96, pages 173–182. ACM, 1996.
- [GJM⁺12] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. Comet: Code offload by migrating execution transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI’12, pages 93–106. USENIX Association, 2012.
- [GP16a] Andrea Gallidabino and Cesare Pautasso. Deploying stateful web components on multiple devices with liquid.js for polymer. In *Proceedings of the 19th International ACM Sigsoft Symposium on Component-Based Software Engineering*, CBSE 2016, pages 85–90, 2016.
- [GP16b] Andrea Gallidabino and Cesare Pautasso. The liquid.js framework for migrating and cloning stateful web components across multiple devices. In *Proceedings of the 25th International World Wide Web conference*, pages 183–186. ACM, 2016.
- [GP17] Andrea Gallidabino and Cesare Pautasso. Maturity model for liquid web architectures. In *Proceedings of the 17th International Conference on Web Engineering*, ICWE2017, pages 206–224. Springer, 2017.
- [GPI⁺16] Andrea Gallidabino, Cesare Pautasso, Ville Ilvonen, Tommi Mikkonen, Kari Systs, Jari-Pekka Voutilainen, and Antero Taivalsaari. On the architecture of liquid software: Technology alternatives and design space. In *13th Working IEEE/IFIP Conference on Software Architecture*, WICSA 2016, 2016.

- [GPM⁺17] Andrea Gallidabino, Cesare Pautasso, Tommi Mikkonen, Kari Systs, Jari-Pekka Voutilainen, and Antero Taivalsaari. Architecting liquid software. *Journal of Web Engineering*, 16, 2017.
- [GT16] Dominique Guinard and Vlad Trifa. *Building the Web of Things: With Examples in Node.js and Raspberry Pi*. Manning Publications Co., 1st edition, 2016.
- [GTA06] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII*, pages 151–162. ACM, 2006.
- [GTMW11] Dominique Guinard, Vlad Trifa, Friedemann Mattern, and Erik Wilde. From the internet of things to the web of things: Resource-oriented architecture and best practices. In *Architecting the Internet of Things*, pages 97–129. 2011.
- [GTW10] Dominique Guinard, Vlad Trifa, and Erik Wilde. A resource oriented architecture for the web of things. In *Proceedings of the 2010 First International Conference on the Internet of Things, IoT2010*, pages 1–8, 2010.
- [Gui11] Dominique Guinard. *A Web of Things Application Architecture – Integrating the Real-World into the Web*. PhD thesis, ETH Zurich, Zurich, Switzerland, 2011.
- [GWYL05] Buğra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. Adaptive load shedding for windowed stream joins. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management, CIKM '05*, pages 171–178. ACM, 2005.
- [HAG⁺09] Martin Hirzel, Henrique Andrade, Buğra Gedik, Vibhore Kumar, Howard Nasgaard, Giuliano Losa, Mark Mendell, Robert Soulé, and Kun-Lung Wu. Spl stream processing language specification. Technical Report RC24 897, IBM Research, 2009.
- [HAG⁺13] Martin Hirzel, Henrique Andrade, Buğra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, and Kun-Lung Wu. Ibm

- streams processing language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3-4):1:7–1:7, 2013.
- [HHE15] Christer Holmberg, Stefan Hakansson, and Goran A.P. Eriksson. Web real-time communication use cases and requirements. RFC 7478, 2015.
- [Hin10] Pieter Hintjens. ZeroMQ: The Guide, 2010. <http://zguide.zeromq.org/page:all>.
- [HN15] Maria Husmann and Moira C. Norrie. XD-MVC: Support for cross-device development. In *First International Workshop on Interacting with Multi-Device Ecologies in the Wild*, Cross-Surface 2015. ETH Zürich, Switzerland, 2015.
- [HPB⁺98] John H. Hartman, Larry L. Peterson, Andy Bavier, Peter A. Bigot, Patrick Bridges, Brady Montz, Rob Piltz, Todd A Proebsting, and Oliver Spatscheck. Joust: A Platform for Liquid Software. *IEEE Computer*, 32:50–56, 1998.
- [HSS⁺14] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, 2014.
- [IBY⁺07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72. ACM, 2007.
- [IJHS14] Yuichi Igarashi, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. Vision: Towards an extensible app ecosystem for home automation through cloud-offload. In *Proceedings of the Fifth International Workshop on Mobile Cloud Computing & Services*, MCS '14, pages 35–39. ACM, 2014.
- [JAF⁺06] Shawn R. Jeffery, Gustavo Alonso, Michael J. Franklin, Wei Hong, and Jennifer Widom. A pipelined framework for online cleaning of sensor data streams. In *22nd International Conference on Data Engineering*, ICDE'06, pages 140–152, 2006.

- [JBA08] Anton Jansen, Jan Bosch, and Paris Avgeriou. Documenting after the fact: Recovering architectural design decisions. *Journal of Systems and Software*, pages 536–557, 2008.
- [KAH⁺12] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proceedings of the 2012 IEEE International Conference on Computer Communications*, INFOCOM2012, pages 945–953, 2012.
- [KAS⁺14] Vasvi Kakkad, Saeed Attar, Andrew E. Santosa, Alan Fekete, and Bernhard Scholz. Curracurrong: a stream programming environment for wireless sensor networks. *Softw., Pract. Exper.*, 44(2):175–199, 2014.
- [KCF15] Supun Kamburugamuve, Leif Christiansen, and Geoffrey C. Fox. A framework for real time processing of sensor data in the cloud. *J. Sensors*, 2015:468047:1–468047:11, 2015.
- [KCS05] Vibhore Kumar, Brian F. Cooper, and Karsten Schwan. Distributed stream management using utility-driven self-adaptive middleware. In *Second International Conference on Autonomic Computing*, ICAC’05, pages 3–14, June 2005.
- [KDFS14] Vasvi Kakkad, Akon Dey, Alan Fekete, and Bernhard Scholz. Curracurrong cloud: Stream processing in the cloud. In *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops*, ICDE 2014, pages 207–214, 2014.
- [KG10] Ramkumar Krishnan and Jonathan Goldstein. A hitchhiker’s guide to microsoft streaminsight queries. <http://go.microsoft.com/fwlink/?LinkID=196344&clcid=0x409>, June 2010.
- [KL10] Karthik Kumar. and Yung-Hsiang Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, April 2010.
- [KLvV06] Philippe Kruchten, Patricia Lago, and Hans van Vliet. Building up and reasoning about architectural knowledge. In *Proceedings of the Second international conference on Quality of Software Architectures*, pages 43–58, 2006.

- [Kov13] Matthias Kovatsch. Coap for the web of things: From tiny resource-constrained devices to the web browser. In *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*, UbiComp '13 Adjunct, pages 1495–1504. ACM, 2013.
- [KPKB12] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo: A computation offloading framework for smartphones. In *Mobile Computing, Applications, and Services*, volume 76 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 59–79. Springer Berlin Heidelberg, 2012.
- [KR70] Werner Kunz and Horst WJ Rittel. *Issues as elements of information systems*, volume 131. University of California Berkeley, California, 1970.
- [KSV13] Dharmesh Kakadia, Prasad Saripalli, and Vasudeva Varma. Mecca: Mobile, efficient cloud computing workload adoption framework using scheduler customization and workload migration decisions. In *Proceedings of the First International Workshop on Mobile Cloud Computing & Networking*, MobileCloud '13, pages 41–46. ACM, 2013.
- [Lan01] Marc Langheinrich. Privacy by design - principles of privacy-aware ubiquitous systems. In *Proceedings of the 3rd International Conference on Ubiquitous Computing*, UbiComp 2001, pages 273–291. Springer-Verlag, 2001.
- [LCBR02] Marc Langheinrich, Vlad Coroama, Jürgen Bohn, and Michael Rohs. As we may live – real-world implications of ubiquitous computing. Technical Report, 2002.
- [LLX⁺11] Yan Liu, Xin Liang, Lingzhi Xu, Mark Staples, and Liming Zhu. Composing enterprise mashup components and services using architecture integration patterns. *Journal of Systems and Software*, 84(9):1436–1446, 2011.
- [LQX⁺16] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. Streamscope: Continuous reliable distributed processing of big data streams. In *Proceedings of the 13th USENIX Sym-*

- posium on Networked Systems Design and Implementation*, NSDI 16, pages 439–453. USENIX Association, 2016.
- [MAK07] René Müller, Gustavo Alonso, and Donald Kossmann. Swissqm: Next generation data processing in sensor networks. In *Proceedings of the Third Biennial Conference on Innovative Data Systems Research*, CIDR 2007, pages 1–9, 2007.
- [McK98] George McKay. *DiY Culture: Party & Protest in Nineties Britain*. Verso, 1998.
- [Mei11] Erik Meijer. The world according to linq. *Commun. ACM*, 54(10):45–51, October 2011.
- [MRX08] Satyajayant Misra, Martin Reisslein, and Guoliang Xue. A survey of multimedia streaming in wireless sensor networks. *Communications Surveys & Tutorials, IEEE*, 10(4):18–39, 2008.
- [MSP15] Tommi Mikkonen, Kari Systa, and Cesare Pautasso. Towards liquid web applications. In *Proceedings of the 15th International Conference on Web Engineering*, ICWE 2015, pages 134–143. Springer, 2015.
- [MWA⁺03] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research*, CIDR2003, pages 245–256, 2003.
- [Nav15] Davide Nava. Stateful operators for dynamic streaming topologies. Master’s thesis, University of Lugano, Switzerland, 2015.
- [NDK⁺03] Suman Nath, Amol Deshpande, Yan Ke, Phillip B. Gibbons, Brad Karp, and Srinivasan Seshan. Irisnet: An architecture for internet-scale sensing services. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB ’03, pages 1137–1140. VLDB Endowment, 2003.
- [nod13] Node-red, 2013. <http://nodered.org/>.

- [NRNK10] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 170–177. IEEE Computer Society, 2010.
- [PdABL⁺13] Ricardo Aparecido Perez de Almeida, Michael Blackstock, Rodger Lea, Roberto Calderon, Antonio Francisco do Prado, and Helio Crestana Guardia. Thing broker: A twitter for things. In *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*, UbiComp '13 Adjunct, pages 1545–1554. ACM, 2013.
- [PDGQ05] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13(4):277–298, October 2005.
- [Pea09] Siani Pearson. Taking account of privacy when designing cloud computing services. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, pages 44–52. IEEE Computer Society, 2009.
- [QHS⁺13] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 1–14. ACM, 2013.
- [RCH⁺04] Tom Rodden, Andy Crabtree, Terry Hemmings, Boriana Koleva, Jan Humble, Karl-Petter Akesson, and Pär Hansson. Between the dazzle of a new building and its eventual corpse: Assembling the ubiquitous home. In *Proceedings of the 5th Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques*, DIS '04, pages 71–80. ACM, 2004.
- [RCTSR12] Pedro Rodríguez, Javier Cerviño, Irena Trajkovska, and Joaquin Salvachua Rodriguez. Advanced Videoconferencing Services Based on WebRTC. In *Proceedings of the 2012 International Conference on Web Based Communities*, pages 180–184, 2012.
- [sam14] Apache samza. <http://samza.incubator.apache.org>, 2014.

- [SCCH13] Matthias J. Sax, Malu Castellanos, Qiming Chen, and Meichun Hsu. Performance optimization for distributed intra-node-parallel streaming systems. In *Proceedings of the 2014 IEEE 30th International Conference on Data Engineering Workshops, ICDE2014*, pages 62–69. IEEE Computer Society, 2013.
- [SFBS12] Frederick Steinke, Tobias Fritsch, Daniel Brem, and Svenja Simonsen. Requirement of aal systems: Older persons’ trust in sensors and characteristics of aal technologies. In *Proceedings of the 5th International Conference on Pervasive Technologies Related to Assistive Environments, PETRA ’12*, pages 15:1–15:6. ACM, 2012.
- [SPGV07] Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. Streamflex: High-throughput stream programming in java. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, OOPSLA ’07*, pages 211–228. ACM, 2007.
- [sto11] Storm, distributed and fault-tolerant realtime computation, 2011. <http://storm-project.net/>.
- [Sun88] Sun Microsystems, Inc. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1057, RFC Editor, 1988.
- [TCE⁺10] Peter Tolmie, Andy Crabtree, Stefan Egglestone, Jan Humble, Chris Greenhalgh, and Tom Rodden. Digital plumbing: The mundane work of deploying ubicomp in the home. *Personal Ubiquitous Comput.*, 14(3):181–196, 2010.
- [TKA02] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction, CC ’02*, pages 179–196. Springer-Verlag, 2002.
- [TM17] Antero Taivalsaari and Tommi Mikkonen. A roadmap to the programmable world: Software challenges in the iot era. *IEEE Software*, 34(1):72–80, 2017.
- [TMS14] Antero Taivalsaari, Tommi Mikkonen, and Kari Systä. Liquid software manifesto: The era of multiple device ownership and its implications for software architecture. In *Proceedings of the 2014*

- IEEE 38th Annual Computer Software and Applications Conference, COMPSAC '14*, pages 338–343. IEEE Computer Society, 2014.
- [VJWS13] Christian Vogt, Max Jonas Werner, and Thomas C. Schmidt. Content-centric User Networks: WebRTC as a Path to Name-based Publishing. In *Proceedings of the 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–3. IEEE Press, 2013.
- [VWS13] Christian Vogt, Max Jonas Werner, and Thomas C. Schmidt. Leveraging WebRTC for P2P Content Distribution in Web Browsers. In *Proceedings of the 21st IEEE International Conference on Network Protocols, ICNP*, pages 1–2. IEEE Press, 2013.
- [YIF⁺08] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08*, pages 1–14. USENIX Association, 2008.
- [ZBW⁺12] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. Scope: parallel databases meet mapreduce. *The VLDB Journal*, 21(5):611–636, 2012.
- [ZDL⁺12] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion 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 Computing, Hot-Cloud'12*, pages 10–16. USENIX Association, 2012.
- [ZRN08] Nan Zang, Mary Beth Rosson, and Vincent Nasser. Mashups: who? what? why? In *Proceedings of the 2008 CHI Conference on Human Factors in Computing Systems, CHI2008*, pages 3171–3176. ACM, 2008.
- [ZSC⁺03] Stanley B. Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur Çetintemel, Magdalena Balazinska, and Hari Balakrishnan. The aurora and medusa projects. *IEEE Data Eng. Bull.*, 26(1):3–10, 2003.
- [ZWW13] Weiwen Zhang, Yonggang Wen, and Dapeng Oliver Wu. Energy-efficient scheduling policy for collaborative execution in mobile

cloud computing. In *Proceedings of the 2013 IEEE International Conference on Computer Communications, INFOCOM2013*, pages 190–194, 2013.