$_{SS}$IOTA: A SYSTEM SOFTWARE FRAMEWORK FOR THE INTERNET OF THINGS

A Thesis
Presented to
The Academic Faculty

by

David J. Lillethun

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
May 2015

ssIOTA: A SYSTEM SOFTWARE FRAMEWORK FOR THE INTERNET OF THINGS

Approved by:

Professor Umakishore Ramachandran,
Advisor
College of Computing
Georgia Institute of Technology

Professor Karsten Schwan
College of Computing
Georgia Institute of Technology

Professor Mustaque Ahamad
College of Computing
Georgia Institute of Technology

Dr. Flavio Bonomi
IoXWorks, Inc.
Georgia Institute of Technology

Professor Santosh Pande
College of Computing
Georgia Institute of Technology

Date Approved: 24 March 2015

To my mother, Marjean

who taught me the value of education

and always supported my many years of school

ACKNOWLEDGEMENTS

Completing a dissertation is a long and difficult road that I never could have navigated alone. There are too many people I must thank, but a few deserve special note.

First and foremost is my advisor, Kishore Ramachandran, without whose research advice, unwavering support, and constant encouragement I never could have done this. He always believed in me, even through the rough times.

I'd also like to thank my committee, Karsten Schwan, Mustaque Ahamad, Santosh Pande, and Flavio Bonomi whose guidance kept me on the right track and whose critical eye kept me honest about my work. I'd also like to thank Mostafa Ammar who served on my proposal committee.

I am grateful for the support of the National Science Foundation and the University of Stuttgart. Thanks also to Prof. Kurt Rothermel and his lab for great collaboration opportunities and an enjoyable and productive exchange of ideas. I am also grateful to all of the companies that have provided me with internships during my study, but most especially to Cisco, and specifically to Flavio Bonomi and Ashok Moghe, for helping me to contextualize my research and ground it in real world needs.

I'd like to thank everyone at the Center for the Enhancement of Teaching and Learning, most especially Dia Sekayi, Esther Jordan, Alexandra Coso, and Carol Subiño Sullivan, for their guidance and constant efforts to help me become a better teacher. Not only are they passionate about teaching and learning, but they sincerely care for others who share that passion.

Thanks to my frequent research collaborators, Kirak Hong and Beate Ottenwälder, with whom it has been a great pleasure to work. They are as clever as they are hard working. I'm also grateful to my seniors and classmates who have given me guidance and shared this journey with me, especially Rajnish Kumar, Hasnain Mandviwala, David Hilley, Nova Ahmed, Bikash Agarwalla, Junsuk Shin, Hyojun Kim, Lateef Yusuf, Moonkyung "MK"

Ryu, Dushmanta Mohapatra, Wonhee Cho, Yeonju Jeong, Steffen Maass, Brian Railing, Eric Hein, Bryan Payne, Rocky Dunlap, Romain Cledat, Mukil Kesavan, and everyone in the Graduate Student Council.

I sincerely appreciate my friends, both at Georgia Tech and in Atlanta at large, who have stood by me and helped me through life's ups and downs. I'd especially like to thank my former roommates, Dave Roberts and Daniel Buchmueller, as well as my good friend, Michael Roderick, who has treated me like family. Last but far from least, I am very grateful to all my family, especially my mother, Marjean, and my brother, Erik, who have given me their constant support on this journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Sensors are widely deployed in our environment, and their number is increasing rapidly. In the near future, billions of devices will all be connected to each other, creating an Internet of Things. Furthermore, computational intelligence is needed to make applications involving these devices truly exciting. In IoT, however, the vast amounts of data will not be statically prepared for batch processing, but rather continually produced and streamed live to data consumers and intelligent algorithms. We refer to applications that perform live analysis on live data streams, bringing intelligence to IoT, as the *Analysis of Things*.

However, the Analysis of Things also comes with a new set of challenges. The data sources are not collected in a single, centralized location, but rather distributed widely across the environment. AoT applications need to be able to access (consume, produce, and share with each other) this data in a way that is natural considering its live streaming nature. The data transport mechanism must also allow easy access to sensors, actuators, and analysis results. Furthermore, analysis applications require computational resources on which to run. We claim that system support for AoT can reduce the complexity of developing and executing such applications.

To address this, we make the following contributions:

- A framework for systems support of Live Streaming Analysis in the Internet of Things, which we refer to as the Analysis of Things (AoT), including a set of requirements for system design
- A system implementation that validates the framework by supporting Analysis of Things applications at a local scale, and a design for a federated system that supports AoT on a wide geographical scale
- An empirical system evaluation that validates the system design and implementation, including simulation experiments across a wide-area distributed system

We present five broad requirements for the Analysis of Things and discuss one set of specific system support features that can satisfy these requirements. We have implemented a system, called $_{SS}$IoTa, that implements these features and supports AoT applications running on local resources. The programming model for the system allows applications to be specified simply as operator graphs, by connecting operator inputs to operator outputs and sensor streams. Operators are code components that run arbitrary continuous analysis algorithms on streaming data. By conforming to a provided interface, operators may be developed that can be composed into operator graphs and executed by the system. The system consists of an Execution Environment, in which a Resource Manager manages the available computational resources and the applications running on them, a Stream Registry, in which available data streams can be registered so that they may be discovered and used by applications, and an Operator Store, which serves as a repository for operator code so that components can be shared and reused. Experimental results for the system implementation validate its performance.

Many applications are also widely distributed across a geographic area. To support such applications, $_{SS}$IoTa must be able to run them on infrastructure resources that are also distributed widely. We have designed a system that does so by federating each of the three system components: Operator Store, Stream Registry, and Resource Manager. The Operator Store is distributed using a distributed hast table (DHT), however since temporal locality can be expected and data churn is low, caching may be employed to further improve performance. Since sensors exist at particular locations in physical space, queries on the Stream Registry will be based on location. We also introduce the concept of *geographical locality*. Therefore, range queries in two dimensions must be supported by the federated Stream Registry, while taking advantage of geographical locality for improved average-case performance. To accomplish these goals, we present a design sketch for SkipCAN, a modification of the SkipNet and Content Addressable Network DHTs. Finally, the fundamental issue in the federated Resource Manager is how to distributed the operators of multiple applications across the geographically distributed sites where computational resources can execute them. To address this, we introduce DistAl, a fully distributed algorithm that

assigns operators to sites. DistAl also respects the system resource constraints and application preferences for performance and quality of results (QoR), using application-specific utility functions to allow applications to express their preferences. DistAl is validated by simulation results.

CHAPTER I

INTRODUCTION

Sensors are widely deployed in our environment, and their number is increasing rapidly. In the near future, billions of devices will all be connected to each other, creating an Internet of Things. There are predicted to be 25 billion devices connected to the Internet by the end of 2015, and 50 billion by 2020 [11]. Among these devices will be millions or billions [11] of ubiquitous sensors with myriad sensing capabilities. Some of these devices, such as vehicles, are not even traditionally thought of as computers.

However, the Internet of Things (IoT) is much more than these endpoint devices connected by a network infrastructure. They are also connected to the physical world through sensors and actuators. To truly realize IoT, these devices (sensors, actuators, and user devices) must coordinate and collaborate. Furthermore, computational intelligence is needed to make applications involving these devices truly exciting. In the present world, we already see the impact of big-data analytics, where vast amounts of statically collected data are correlated to produce new knowledge that empowers intelligent applications. In IoT, however, the vast amounts of data will not be statically prepared for batch processing, but rather continually produced and streamed live to data consumers and intelligent algorithms. We refer to applications that perform live analysis on live data streams as *Live Streaming Analysis* applications.

Live Streaming Analysis is a broad category of applications that are applicable in many contexts, including situation awareness, cyberphysical systems, and financial analysis. However, when Live Streaming Analysis is applied to bring intelligence to the Internet of Things, we call this the *Analysis of Things*.

The Analysis of Things (AoT) presents an opportunity for exciting new kinds of intelligent applications, such as traffic monitoring that combines vehicular sensors from many

vehicles, roadside sensors, and traffic cameras to optimize flow and perform live traffic engineering. Another example is using in-store sensors to provide live retail analysis, helping customers (while making more sales) in individual stores, and making inventory and marketing decisions on multi-store regional and global scales. Walmart is already experimenting with such a retail application using multimodal sensors with computer vision and analysis logic created by Shopperception [20].

However, AoT also comes with a new set of challenges beyond even traditional Live Streaming Analysis applications. The data sources are not collected in a single, centralized location, but rather distributed widely across the environment. AoT applications need to be able to access (consume, produce, and share with each other) this data in a way that is natural considering its live streaming nature. The data transport mechanism must also allow easy access to sensors, actuators, and analysis results.

Furthermore, analysis applications require computational resources on which to run. The resources should be elastic to support the dynamic nature of AoT, and should also enable resource sharing between different tenants. While these are strengths of cloud computing, and the cloud has been used effectively for big-data analytics on static data, it is not the most effective solution to stream thousands or more widely distributed sensor streams to a single cloud data center in order to analyze them. Notably, a significant portion of them may not even produce any "interesting" data at a given moment, making it wasteful to send their data to the cloud. Instead, widely distributed sensors, actuators, and user devices suggest the need for widely distributed computational resources to run analysis applications. Furthermore, an execution environment should provide convenient access to these computational resources, while facilitating the development and deployment of AoT applications.

1.1 Motivation

There is a great deal of complexity involved in creating and executing Analysis of Things applications. Systems support can help to reduce that complexity by managing difficult functionality that applications commonly require. Table 1 shows some examples of systems

that simplify development of distributed applications by providing data access functionality or managing execution on distributed computational resources. For example, systems such as the Google File System [13] and Dynamo [9] provide batch computing applications with access to data on distributed persistent storage. Meanwhile, MapReduce [8] simplifies the development of batch computing by providing a simple programming model and a system to execute applications on distributed computational resources.

Table 1: Systems Support Reduces the Complexity of Creating and Executing Applications by Abstracting Data Access and Computation Management

	Data Access	**Computation**
Batch Processing	GFS, Dynamo	MapReduce, Hadoop
Live Streaming Analysis	Stampede^{RT} / PTS, Publish-Subscribe	System S / InfoSphere Streams, MillWheel

Systems support for live streaming data access includes publish-subscribe systems [10], Stampede^{RT} [17], which provides time indexed streams to facilitate stream synchronization and correlation, and Persistent Temporal Streams (PTS) [18], which further adds the ability to provide persistent storage for streams and seamlessly access historical stream data. There are also systems that support execution of Live Streaming Analysis on distributed resources, including System S[1] [5, 12] and MillWheel [4]. However, these systems lack the support needed to address the unique challenges of the Internet of Things. Specifically, they lack the flexibility and extensibility to support any computation on any stream from any sensor. They also schedule execution only on closely distributed resources, such as a compute cluster, and provide no support for widely distributed computational resources that are required for the Internet of Things.

Data center computing, and cloud computing in particular, has become a common method for deploying applications. However, most applications that exist today, including web applications and mobile apps, operate on a client-server model, where each end device communicates with the server in the cloud and not directly with each other. The "server" in the cloud may, in fact, be a set of distributed systems (such as in the classic 3-Tier or

[1] System S is available commercially under the name InfoSphere Streams. This document will use the term System S to refer generically to both the research and commercial versions.

N-Tier models) in order to scale computation and storage with the load. However, since a data center exists at one particular geographic point, it cannot scale with the volume of data that will be provided with the explosion of devices that will be connected and participating in applications in the Internet of Things.

Fortunately, unlike traditional applications, real-world geography has significance in the Internet of Things – end devices (sensors, actuators, and user devices) will be most interested in collaborating with other end devices that are near themselves, while analyses correlating more distant sensors will tend to be more latency-tolerant and use digested forms of data. This gives us an opportunity to avoid converging all the data at one point for analysis by filtering out unimportant data locally and using data of local importance locally, thus only sending data to the cloud that is both important and globally interesting. In order to accomplish this, resources such as computation and storage need to be available locally as well as in the cloud. In other words, those resources need to be pushed to the edge of the network.

Fog computing [7] is the idea that widely distributed resources, such as computing and storage, will be available throughout the network. They will have a hierarchical arrangement that mirrors the structure of the Internet (e.g., access, aggregation, core networks). Like the cloud, these resources will be virtualized to support elasticity and multi-tenancy. The Analysis of Things can help by performing the live analysis needed by applications at local and regional scales and by analyzing sensor streams, including "rich" sensor streams (e.g., audio, video, radar, etc.), to determine which should be filtered or sent on to the cloud for global analysis. Systems support for AoT, therefore, needs to manage and schedule applications on widely distributed resources, such as those provided by Fog computing. Because applications may include local, regional, and global analysis, it is not sufficient to run independent systems at different locales in the Fog, but rather the systems must be a federated network of sites that are able to both manage local resources and work together seamlessly to support applications that span locales.

1.2 Thesis Statement and Contributions

Thesis Statement: *Systems support for Live Streaming Analysis in the Internet of Things can reduce the complexity of developing and executing such applications on computational resources and using end devices that are widely distributed at the edge of the network.*

Live Streaming Analysis applications in the Internet of Things, which we have dubbed the "Analysis of Things", can be very complex. They require deep expertise in a number of different domain areas and may involve complex structures and interactions between algorithms. They also must deal with an extremely large number of heterogeneous sensors distributed throughout the Internet, including yet-to-be-developed types of sensors. Developers must be able to find the ones they need and then access them. Meanwhile, the applications must ultimately be deployed on distributed computational resources that are spread throughout all parts of the Internet. Fortunately, appropriate systems support can aid users by abstracting away system issues at run time and facilitating their dealing with development complexity.

Contributions:

- A framework for systems support of Live Streaming Analysis in the Internet of Things, which we refer to as the Analysis of Things (AoT), including a set of requirements for system design
- A system implementation that validates the framework by supporting Analysis of Things applications at a local scale, and a design for a federated system that supports AoT on a wide geographical scale
- An empirical system evaluation that validates the system design and implementation, including simulation experiments across a wide-area distributed system

CHAPTER II

BACKGROUND

In this chapter, we provide background for system support for the Analysis of Things by discussing the Internet of Things in more detail and by considering state-of-the-art systems for Live Streaming Analysis.

2.1 The Internet of Things

The basic definition of the Internet of Things (IoT) is that applications will be aware not only of virtual concepts but also of real things in the physical world. Although the original idea was proposed using RFID to provide awareness of physical objects [6], the myriad types of sensors developed since then have added opportunities to gain much richer awareness of many more types of objects. From this base definition, what IoT actually is has been elaborated and refined in many different ways, thus meaning different things to different people. We now present our concept of the Internet of Things, which is what we mean by the term throughout this dissertation, and elaborate upon it by discussing some of the characteristics of IoT (according to our definition).

In the traditional Internet that exists today, end devices are predominately user devices, e.g., PCs, laptops, smart phones, tablets, etc. Many of these devices are mobile as well. In the Internet of Things, the proportion of user devices that are mobile will continue to increase. More to the point, however, end devices will not only be user devices but also sensors and actuators that are capable of creating digital representations of the physical world, and then using digital information to produce physical effects on the world. While user devices may be involved, IoT opens the opportunity for many machine-to-machine (M2M) interactions where a human is not necessarily in the loop. In particular, IoT can enable a vast array of cyberphysical systems, where applications can sense the environment and take action on their own.

There are a number of connectivity-related characteristics to IoT. As mentioned, mobility will be commonplace, not only for user devices but for sensors and actuators too, in some cases. Consider an intelligent, connected vehicle; such a platform may include many sensors to report the vehicle state, actuators to control the vehicle, as well as user devices – all of which are necessarily mobile with the vehicle. Furthermore, wireless connectivity will become predominant at the edge of the network, not only to support mobility but also to allow sensor deployments in environments where wired connections are difficult to support.

Wireless sensor networks (WSNs) will become commonplace, but will not constitute the whole of sensing in IoT. Indeed, we have considered the connected vehicle as a sensor platform that is not a WSN – devices within the vehicle are wired to each other, and only the vehicle as a whole is connected wirelessly to allow mobility. Many sensors and actuators may also be infrastructure-connected, rather than WSNs, such as traffic cameras that monitor the highways or security cameras in airports.

Much work in IoT is focused on various issues of connectivity. Besides the mobility and wireless issues discussed, important issues of addressing and routing need to be resolved. However, even with every device - every sensor and every actuator - able to communicate with each other, we do not consider the Internet of Things to be complete. In order for truly useful things to happen in IoT, applications must consume and process the sensor data. Therefore, components of IoT include not only end devices and connectivity, but also data access and resources for processing, i.e., compute and storage resources.

Furthermore, IoT will use a combination of simple and rich sensing. A temperature sensor (i.e., thermometer or thermostat) is an example of a simple sensor. It senses a single value and provides it in a structured form, such as a single floating point value. Compound simple sensors are also possible, such as a single package that contains temperature, humidity, and air pressure sensors and provides their data in a structured form – perhaps XML or JSON containing fields for the three values. "Rich" sensors, on the other hand, consume complex data from the environment and provide it in an unstructured form. These sensors are "rich" because they can provide a great deal more information about the environment, but at the same time, they need to be processed and interpreted in order to extract that

information from the raw sensor data. Cameras are a prime example of rich sensing. Much can be learned about the environment from video, but significant processing of the raw video stream is required in order to determine what is occurring in the scene. Other rich sensors include microphones and radar / lidar.

There is some work on providing data access to sensors, allowing elaborate queries against multiple sensors so that applications can get right to the sensor data they most need. These can even allow queries against the actual sensor data streams, but only for simple sensors. Rich sensors require advanced and custom processing in order to extract information from the sensor data that can be queried against, but such data access systems do not provide facilities for such processing of data streams as part of the query. Therefore, while these contributions are important and useful for the Internet of Things, IoT also requires facilities for custom, compute-intensive analysis of sensor streams that can extract features and events from rich sensors, and/or by correlating features/events between sensors. The ability to run useful business logic that turns these events into results, possibly driving actuators in M2M applications, is also necessary.

Therefore the Internet of Things consists of not only various end devices (sensors, actuators, and user devices) with wireless and mobile connectivity, but also data access, Live Streaming Analysis, and resources for applications to use, particularly compute and storage. Only when computational intelligence using both simple and rich sensors and advanced, custom algorithms (e.g., machine learning and computer vision) can be performed on available computational resources is our vision of IoT actually achieved.

Having thus defined the Internet of Things, we will now consider some of the characteristics of IoT. We have already mentioned the massive scale, both in terms of the number of end devices and their variety, as well as the prevalence of mobile devices and wireless access. We also briefly mentioned M2M and cyberphysical systems, which include a closed loop of sense-process-actuate (with the loop completed between actuation and sensing through the medium of the physical environment). Such interactions must involve live analysis and live interactions – the physical world proceeds in real time, and IoT applications must keep up. This further indicates the importance of low latency in IoT. Excessive delays could result in

outdated data being used to make poor decisions, or perhaps worse due to the closed loop nature of many such applications.

In addition to the massive quantity of end devices, these devices will be geographically widely distributed. While user devices in the traditional Internet are also widely distributed, this is of much greater importance in IoT due to the low latency requirement as well as location sensitivity. Unlike the traditional Internet, the location of IoT devices in the physical world matters a great deal. For example, a user (or a user's vehicle) may be greatly concerned about traffic, construction, and accidents along his/her route driving from Atlanta to New York City, but there is no need to retrieve or process data from the roads in California. Such location-sensitive circumstances will be common in the Internet of Things, so devices and applications will have to be location-aware.

It may prove helpful, however, that another characteristic of IoT applications is their hierarchical nature. Different results are desired on local, regional, and global scales. Applications will perform a great deal of local processing, using local sensor data to make local decisions and drive local actuators. Such local processing will be highly latency sensitive, whereas regional and global decision making will be progressively more latency tolerant. Furthermore, decisions further up the hierarchy will also require input from a lower density of sensors and higher-level information. This could allow local processing to filter sensor data and perform initial sensor processing to allow a smaller portion of the sensor data, as well as higher-level events, to be sent up the processing hierarchy. This may aid in stemming the tide of data from the massive quantity of sensors that otherwise would have to all be sent to a central location for processing.

As applications become more latency tolerant as they move up the processing hierarchy, so too increases the time scale on which they consider data and make decisions. Related to this, the liveliness of sensor data is also affected by the hierarchy. Specifically, data used in local processing is short-lived, while regional processing may use long-lived data and global processing will use long-lived and persistent data. The rule for exactly how long lived the data is or how latency sensitive each level of the hierarchy is is not hard-and-fast. For that matter, we discuss "local", "regional", and "global", but there are not necessarily a strict

three levels to the hierarchy, and different applications may not all use the same levels. Rather, we wish to relate the general principle that there exists some hierarchy, and moving up the hierarchy increases the time scale and decreases latency sensitivity, while moving down it decreases the time scale and increases latency sensitivity.

2.2 Live Streaming Analysis Systems

Several Live Streaming Analysis systems exist, including System S [5, 12], S4 [24], MillWheel [4], and Storm [2]. These systems are capable of performing analysis of streaming data in motion and can apply advanced and custom business logic. However, they all bear a common assumption: streams are structured and operators on streams are (mostly) selected from a set of predefined operator types. The predefined operators are often flexible, so a great variety of logic can be achieved with them, but only provided they are operating on structured data. However, unstructured data streams, such as those produced by "rich" sensors, require completely custom-built operators, as do advanced algorithms, such as machine learning. Even where such systems allow limited use of custom operators, they lack the flexibility and extensibility to support any computation on any stream from any sensor.

These systems also schedule computation on closely distributed resources, such as a compute cluster. They lack support for the geographically widely distributed and hierarchical resources needed to support the Internet of Things. Furthermore, they run statically defined applications on statically allocated resources and lack support for the dynamism needed for IoT. For all these reasons, the Analysis of Things requires a purpose-built system that can meet the particular requirements of live streaming analysis in the Internet of Things.

CHAPTER III

ANALYSIS OF THINGS DESIGN REQUIREMENTS

Having motivated the need for the Analysis of Things and for a system to support it in Section 1.1, we now consider how to approach this in detail. First we describe the design principles that we feel should govern AoT. Then we create a set of concrete requirements for a system that supports AoT and ground those requirements in our design principles. Together, these constitute a framework for the design of systems support for the Analysis of Things.

3.1 Design Principles

Based on our consideration of the Internet of Things in Section 2.1, we have identified the following design principles as key to a successful Analysis of Things environment:

1. Standing on the Shoulders of Giants (Sharing)
    (a) developers and administrators can leverage the work of others, allowing them to focus on their areas of expertise
    (b) share algorithms (operators)
    (c) share sensor data
    (d) share analysis results (output from running operator graphs)
2. Flexibility
    (a) any streaming algorithm (operator)
    (b) any type of stream
    (c) any edge device (e.g., sensors and actuators)
    (d) any operator graph (composition of operators/algorithms, edge devices, and data streams that connect them)

3. Scalability

    (a) scalable computational resources (scales with amount of computation)

    (b) scales with the number of devices

    (c) scales with geographical distribution

4. Extensibility

    (a) develop new algorithms (operators)

    (b) define new stream types

    (c) interact with new types of edge devices (e.g., sensors and actuators)

5. Dynamism

    (a) edge devices may come or go

    (b) new operators may be made available or removed

    (c) operators and operator graphs may be instantiated or terminated

    (d) computational resources may be added or removed

The *Standing on the Shoulders of Giants* principle is about the ease of developing and deploying applications in the Analysis of Things. Each user (i.e., developer or administrator) should be able to focus on his/her area of domain or business logic expertise, delegating problems in other areas to appropriate experts. Critically, no user should need to concern him/herself with system details needed to manage computational resources, deploy applications on those resources, or provide stream transport between application components running on those resources.

The *Flexibility* principle is about allowing users to develop and run any kind of Live Streaming Analysis application in the Analysis of Things. Specifically, this necessitates being able to develop operators that run any arbitrary code (with the reasonable constraint that it perform Live Streaming Analysis), and being able to compose any operator graph from the set of available operators (with the reasonable constraint that input/output types

are compatible). In addition, streams must support any arbitrary type of streaming data, even opaque, unstructured data, such as that provided by "rich" sensors. Finally, any sort of edge device (e.g., sensors and actuators) or other data source or sink for operator graphs should be able to be used by AoT applications.

Scalability naturally suggests scaling with the amount of computation being performed by scaling out the amount of computational resources available. However, in addition to this, the scale of the Internet of Things demands that AoT applications be able to scale with the number of edge devices (sensors and actuators), and since those devices are spread out broadly across the physical environment, they must also scale with the geographically distributed area.

Extensibility means that new things (resources, sensors, algorithms, etc.) can be added to the system without restriction. This is necessary to support the wide variety of future applications and yet-to-be-developed algorithms that may come about. It is also necessary to support the variety of present and future sensors that will become available in the Internet of Things.

Dynamism reflects the fact that the application environment changes over time, while AoT applications are running. Edge devices, such as sensors and actuators, may be added or removed, operators providing new algorithm implementations may also be added or removed, and new operator graphs providing analyses may be started or terminated. In addition, this interacts with the Scalability requirement as new computational resources may be made available or removed. The Analysis of Things must behave gracefully in the face of this turmoil, and any AoT system must allow for such changes to occur at runtime.

3.2 System Requirements

Inspired by the AoT design principles of the previous section, we have come up with the following requirements for systems support for AoT. Each requirement (or sub-requirement) has the relevant design principles noted in parentheses, according to the principle's number in the list in Section 3.1.

1. Abstract away system-level issues such as managing distributed resources, deploying applications on resources, and stream transport between application components (DP1)

2. Component-based design

   (a) operators can include arbitrary, novel code (DP2)

   (b) operators can leverage third-party code, such as libraries (DP1&2)

   (c) operators can be reused and shared (DP1&4)

   (d) operators can be shared and unshared at run time (DP5)

   (e) operator graphs (or parts of them) may be the input for other operator graphs (DP1)

3. Execute applications on distributed resources (DP3)

   (a) take advantage of the natural parallelism in these applications (DP3)

   (b) execute arbitrary operator graphs (DP2)

   (c) allow starting and terminating operator graphs at run time (DP5)

   (d) Allow adding and removing resources at run time (DP5)

4. Provide efficient and scalable stream transport in the distributed environment (DP3)

   (a) streams contain any type of data, including opaque, unstructured data (DP2)

   (b) a stream's multiple consumers may include different operators in the same operator graph, as well as other operator graphs (DP1)

5. Enable stream registration and discovery (DP1&4)

   (a) allow dynamically adding and removing stream registrations (DP5)

   (b) allow describing and querying for streams with arbitrary, novel types (DP2&4)

6. Enable connecting operator graphs to arbitrary external (i.e., not operators) inputs and outputs (e.g., sensor and actuators) (DP2&4)

7. Widely distributed architecture, specifically a federated architecture

   (a) provides scalability over a large number of external devices and a geographically distributed area (DP3)

The design principle of Standing on the Shoulders of Giants is satisfied by both alleviating developers from having to deal with system-level issues and by allowing them to share the effort of developing complete applications. Specifically, system requirement 1 alleviates the need for developers to deal with system-level issues. System requirement 2.c allows operators to be shared and reused, allowing operator developers to focus on their own areas of domain expertise and alleviating operator graph developers from having to understand the implementation details of any individual operators. Requirement 2.b further allows operator developers to rely on the work already done by others in the form of third-party code, such as shard libraries. For example, an operator developer working in C++ may wish to use the boost library, or one developing a video analysis operator may wish to use the OpenCV computer vision library. System requirements 2.e and 4.b together allow synthesis of applications by combining operator graphs. The output of an operator graph may be an analysis that is a useful input to another operator graph. The AoT system should provide a mechanism for such sharing of operator graph output, while alleviating the developer of the latter graph from needing to know the internal details of the former one in order to make use of its output. Finally, requirement 5 allows applications to discover and use sensors that were created and deployed by others.

The Flexibility design principle suggests that an AoT system must be sufficiently generic to handle the variety of Live Streaming Analysis Applications that may be created. System requirements 2.a-b state that operators must be allowed to be created that perform arbitrary functions, including the use of third-party code, and the system must be able to deploy and run operator graphs consisting entirely of such operators. Furthermore, the system must be able to execute operator graphs containing arbitrary[1] compositions of the available operators, as stated in requirement 3.b. Flexibility further demands supporting all

[1] This is naturally subject to common sense limitations, such as meeting the operators' contracts for number and types of allowed inputs and outputs.

kinds of streaming data, including "rich" sensor and feature data that may be unstructured and therefore opaque to the system. Requirement 4.a states that the stream transport system must support such streams, while requirement 5.b states that the system must allow descriptions for all kinds of streams, so that they may be registered and queried. Finally, Flexibility demands that operator graphs must be able to use all kinds external sources or sinks (i.e., not just using other operators for input and output) regardless of what protocol or other mechanism is needed to communicate with them (requirement 6).

The design principle, Scalability, requires the computational resources and stream transport system to be scalable to support increasingly heavy loads from operator graphs. Operator graph scaling may result from the number of graphs running simultaneously, the size of the graphs (i.e., number of operators and complexity of their interconnections), or from the use of more heavy-weight operators in the graphs. To this end, system requirement 3 states that scalable, distributed resources should be used for the execution of operators, while requirement 4 states that the streaming system must be scalable, specifically, with the number of concurrent streams and the number of consumers on each stream. However, the Internet of Things provides an additional scalability requirement for an AoT system, due to the immense number of edge devices (e.g., sensors, actuators, and end-user devices) and the geographical distribution of those devices. In order to support devices in far-flung areas working together in a single Analysis of Things application, an AoT system must be widely distributed, as expressed in requirement 7.

The Extensibility design principle means that the system must accommodate stream types, algorithms, and devices not only from a finite set of known ones, but in fact from all possible ones that do exist or might in the future. In short, one must be able to "extend" the system with new stream types, algorithms, and devices. Requirement 2.c allows creating new algorithms in the form of operators that can be used by applications run within the system. Requirements 5 and 6 allows extending the system to connect to any kind of device, even yet-to-be-invented ones. Finally, the system can be extended to support any type of stream by the ability to describe any new stream type for registration and discovery (requirement 5.a).

The Dynamism design principle demands that all the various elements managed by the system be able to be added and removed at run time. The system should not need to be manually reconfigured or restarted in order to allows such additions or removals. Requirement 2.d means that operators can be added and removed while the system runs. Requirement 3.c allows applications to be started and terminated, while 3.d allows adding and removing the compute resources for applications to run on. Finally, requirement 5.a means streams can be created and registered, or deregistered at run time.

CHAPTER IV

SYSTEM SUPPORT FOR THE ANALYSIS OF THINGS

In order to validate our framework, presented in Chapter 3, we designed and implemented a system system that enables Analysis of Things applications within a limited geographic scope, which we call $_{SS}$IoTa[1]. We now discuss how $_{SS}$IoTa fulfills the requirements discussed in Section 3.2 and describe the system's components and mechanisms as necessary. Discussion of requirement 7, "widely distributed architecture," is omitted from this chapter as it is the subject of Chapter 5.

4.1 System Design

Our system design is based on our earlier work [22, 27], but also includes additional work performed since then. The system described here is our latest design.

4.1.1 Requirement 1: Abstract away system-level issues

Requirement 1 is to, "abstract away system-level issues such as managing distributed resources, deploying applications on resources, and stream transport between application components." This is achieved through a combination of our programming model and the system's execution environment.

4.1.1.1 Programming Model

Applications in our AoT system are expressed as compositions of operators[2]. Examples of such operator graphs are shown in Figures 1 and 2. Operators are individual algorithms, represented by code that conforms to a standard interface and optional state, that take streaming inputs and produce streaming outputs. The operator graph for an application, therefore, specifies how operator outputs are connected to the inputs of other operators

[1] "systems support for IoT analytics"

[2] In [27] and [22], operators are referred to as "transformations" or "transformers". However, in this document we have updated the terminology in order to be consistent with our more recent work.

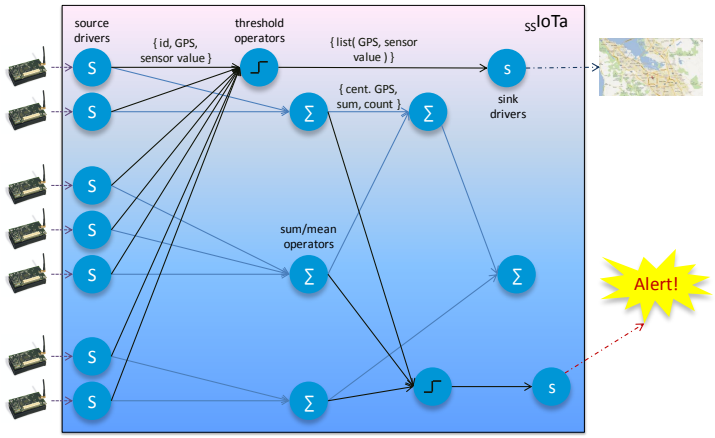

Figure 1: Example AoT Application: Sensor Thresholding – circles are operators, which include summation/mean and thresholding; text in braces describes stream contents; external sensors and actuators are outside the SSIoTa environment

with matching output/input stream types. A particular operator can be used multiple times in an application (or different applications), representing different instantiations of the algorithm, each with its own inputs and run time state.

An application is specified in a multi-line string or text file, with each line representing an operator instance, including the operator name, input streams, and output streams. A short example of such an application file is given in Appendix A. While we have not developed one, we imagine that a graph-builder GUI or IDE could be used to aid in the development of applications, outputting the correct text representation for the operator graph that was designed graphically.

Operators are developed as reusable code modules, which allows them to be developed independently of the applications that use them and facilitates sharing and reuse. Specifically, operators are dynamically linkable code modules (e.g., .so, .dll, .jar, etc.) that implement a system-provided interface. They contain declarations for run time state, an initialization routine that allows some initial state or configuration to be provided, and a handler routine to run the operator's algorithm. This handler will be called by the system

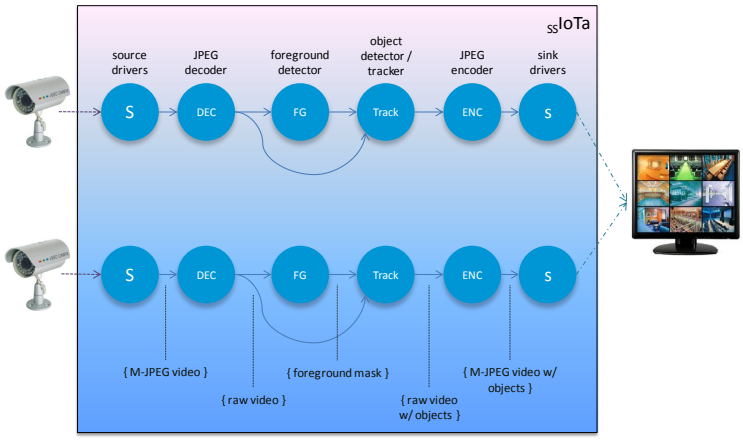

Figure 2: Example AoT Application: Video Object Tracking – circles are operators, which include video decoding and computer vision; text in braces describes stream contents; external sensors and actuators are outside the SSIoTa environment

each time a synchronized set of inputs[3] is ready. Helper functions may also be defined and used within the module.

To create an application, all that needs to be done is to create a set of algorithms in the form of operators, specify how to compose those operators with each other and with sensor inputs, and finally run the application. Neither operator nor application developers need to be concerned with managing distributed computational resources, deploying the operator instances on those resources, transporting data streams to or between operators, or synchronizing the several input streams to each operator. These issues are all handled by the execution environment and streaming subsystem.

4.1.1.2 Execution Environment

The execution environment consists of two main components: a *Resource Manager* and several *Worker Nodes*, which are shown in Figure 3. Worker Nodes (or "Workers" for short) are conceptual machines - running on a native OS on a physical machine or a guest OS on a virtual machine - and provide the computational (CPU, memory, etc.) resources for running

[3] Note that "synchronization" of zero or one input is essentially a no-op.

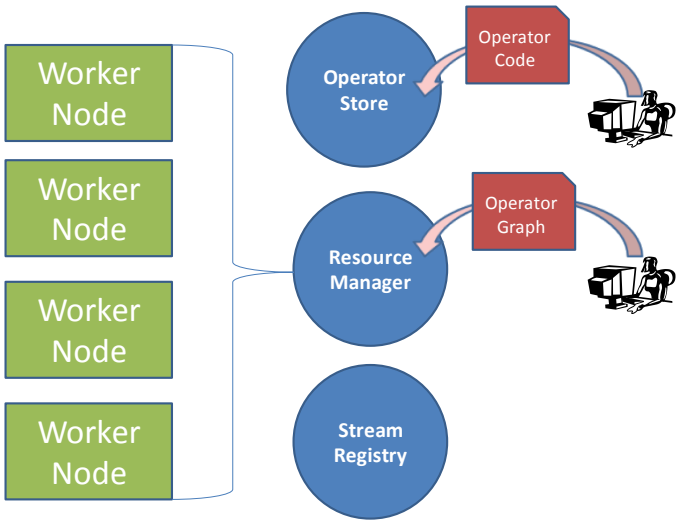

Figure 3: SSIoTa System Architecture – blue circles represent system processes, green squares represent machines or VMs, and red objects with a cut out corner represent developer-provided application logic

operators. The Resource Manager is a server process that is responsible for managing all the Worker Nodes, scheduling and deploying operator instances on the Workers, and instantiating streams in the streaming subsystem (described below) to connect the operator outputs and inputs.

The Resource Manager exposes a public API through a remote procedure call (RPC) mechanism[4] that allows users to run and terminate applications (i.e., operator graphs). When it receives an application to run, it chooses Worker Nodes to provide the computational resources, decides which operator instances in the application should run on each of those Workers (referred to as "scheduling"), creates streams in the streaming subsystem for the outputs of all the operator instances, and then instructs each Worker to start executing its assigned operators.

Worker Nodes are shown in more detail in Figure 4. Each Worker Node has a system-private RPC interface that allows the Resource Manager to control it. When a Worker

[4] our implementation uses Apache Thrift for RPC, but this could easily be replaced with another RPC implementation, or another mechanism that allows stateless procedure calling, e.g., REST interfaces

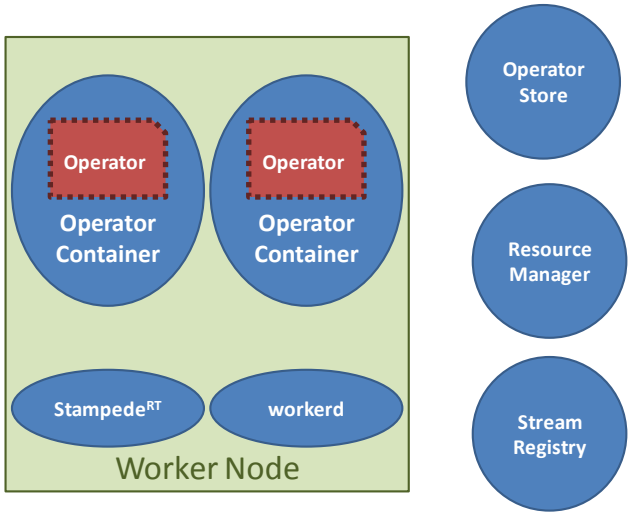

Figure 4: SSIoTa Worker Node Design – blue circles/ovals represent system processes, green squares represent machines or VMs, and red objects with a cut out corner represent running operator instances; the operators' dotted borders indicate that there is *not* a process boundary; two operator container processes are shows to indicate that any number may be run concurrently on a Worker

receives instructions from the Resource Manager to start an operator instance, it first downloads the operator module, i.e., the dynamically linkable shared object, from the Operator Store (described below). Then it creates endpoint connections to the input and output streams for the operator, and begins executing an instance of the operator. Execution of an operator consists of creating the instance, calling it's one-time initialization routine (into which will be passed any initialization data specified in the application description), and then running the operator instance's main loop: 1) system code reads a synchronized set of inputs from the input streams (waiting for input if necessary, e.g., if the operator code is faster than the input stream rate), 2) the developer-provided operator handler is called, and passed the input data, 3) the operator algorithm is executed and the handler code may contain calls to the system API to place items in the output streams, 4) repeat.

Thus the Resource Manager and Worker Node handle a lot of system-level details and keep the application developers' focus on the overall application logic and on creating the individual algorithms. Furthermore, most of the work is performed at setup time, with only

minimal involvement by the system in the critical path during steady-state execution of an application. This provides good application performance, despite the system not knowing the internals of the application algorithms or the contents of the streams.

4.1.1.3 Stampede^{RT}: the Streaming Subsystem

We have thus far described without detail how the system uses a streaming subsystem to abstract away the details of stream transport across a distributed system. For this subsystem, we have implemented the core functionality of the Stampede^{RT} stream abstraction [17], which we will now describe briefly.

Stampede^{RT} provides applications (which is SSIoTa, in this case) with a software abstraction for data streams and a standard API for accessing those streams. The API may be invoked to access any stream from any Stampede^{RT} node in the distributed system, and stream transport is handled transparently behind the scenes.

Streams are represented as ordered sequences of data items, each of which is stored as a binary object (i.e., byte array) and indexed by wall-clock time. Using a binary representation leaves the responsibility for interpreting the stream contents, as well as serialization and deserialization of data items, to the application. The advantage of this approach is that the system is not dependent on the structure of the data in the streams and can support any arbitrary type of streaming data, including unstructured data, such as that from "rich" sensors (e.g., video, audio, etc.). Consumers of a Stampede^{RT} stream are not restricted to receiving each data item in sequence, but may request items that occurred at a certain time, or all items that occurred in a certain time interval. However, using wall-clock time rather than virtual time is important for the ability to synchronize streams, such as sensor or event streams that need to be correlated. For example, analyzing a scene using stereo cameras requires correlating two video frames that were captured at the same instant, and detecting a certain situation may require detecting that two different events in different event streams both occurred within a certain time interval. For further details, we refer the reader to the Stampede^{RT} literature [17, 18].

4.1.2 Requirement 2: Component-based design

The ecosystem of third-party code for traditional applications (e.g., software libraries) has served those applications well. We foresee needing a similar ecosystem for the Internet of Things that allows the creation and use of third-party IoT code. Furthermore, we believe the right level of abstraction for third-party code in IoT is the operator. Therefore, our system aims to provide mechanisms that can support such an ecosystem of operators.

At the same time, we wish to continue to leverage the existing ecosystem of third-party libraries in the development of individual operators. For example, an operator that extracts features from a camera stream may wish to use the libjpeg (e.g., to decode a M-JPEG stream) and OpenCV libraries. Since operator code is simply executed as a shared object library, it has full access to use any third-party libraries. Furthermore, this allows operators to do anything that can be expressed in the language in which they are written[5]. This satisfies requirements 2.a-b.

Requirement 2.e says that the output of an operator graph should be able to be used as the input to another – thus allowing operator graphs to also be composable. Since all operator graphs use the same Stampede^{RT} streaming substrate, there is no barrier at the streaming level - an operator in one operator graph may produce on a stream from which an operator in another graph consumes. The only barrier, therefore, is discovering available streams produced by other operator graphs from which they may consume. This is supported by the Stream Registry, which we will describe in detail below, when we address requirement 5.

4.1.2.1 Operator Store

The remaining question, posed in requirements 2.c-d, is how to enable the operator ecosystem by allowing operators to be shared and discovered. This is addressed by the Operator Store, which is shown in Figure 3.

Once a developer creates an operator, e.g., one that implements an analysis algorithm in

[5] Currently, our implementation only supports C++ code shared as a .so shared object library. However, in principle, we could implement support for any language or type of dynamically linkable object.

their area of expertise, and then compiles it into a reusable code module with the appropriate interface, he/she can upload it to the Operator Store. Along with the code module itself, the developer who uploads the operator gives it a unique name and a description of the types of streams that it takes as inputs and produces as outputs. The way to describe streams will be addressed in more detail below, when we discuss the Stream Registry.

Once an operator is stored in the Operator Store, others can query the operators based on name and/or input and output types. This facilitates discovery of available operators by application developers. Once discovered, the code module implementing an operator can be retrieved from the Operator Store. However, this will not normally need to be done by the application developer him/herself. Instead he/she specifies the operator name in the application description (i.e., operator graph), and the Worker Node running an operator instance will automatically retrieve the code module prior to executing the operator, as described earlier.

4.1.3 Requirement 3: Execute applications on distributed resources

The distributed Worker Nodes in our execution engine enable the natural parallelism in streaming operator graphs. Furthermore, since each operator instance is run within its own process, parallel resources (such as multi-core / multi-processor) can be exploited. This also means that Worker Nodes on different VMs can provide parallel execution even if those VMs are running on the same hardware (assuming that hardware provides parallel resources for the VMs to run on). The separate process model for operators also means that pipeline parallelism (sequential operators) as well as task-level parallelism (parallel operators) in the operator graph may be exploited. This fulfills requirement 3.a.

Another advantage of the separate process model is that any valid[6] operator graph can be executed on the Worker Nodes. This allows requirement 3.b to be met.

Requirement 3.c is met trivially by the previously described public interface to the Resource Manager that allows starting and stopping applications (operator graphs).

Finally, requirement 3.d demands that additional resources can be added (or removed)

[6] Operators' input streams must exist or be provided by other operators in the graph, and the stream types of connected operator outputs and inputs must be compatible.

at run time. Worker Nodes are aware of the Resource Manager that manages them, and when they start up they automatically register themselves with the Resource Manager; when they shut down they similarly deregister themselves[7]. Thus the Resource Manager can keep track of the available resources even as they are added or removed at run time.

4.1.4 Requirement 4: Provide efficient and scalable stream transport

The streaming subsystem was described previously, and we refer you to the related literature on Stampede^{RT} [17] and Persistent Temporal Streams (PTS) [18] for experimental results validating its performance and scalability, including the scalability of streams with multiple consumers.

As mentioned earlier, these streams contain arbitrary, opaque binary data, and neither Stampede$^{\mathrm{RT}}$ nor our system needs to understand the structure of the stream contents. It is left to operator developers to understand the structure of the streams they consume and produce. However, our system does provide flexible and extensible stream descriptions that allow developers to know if streams are compatible with their operators, as will be described in the next subsection. This satisfies requirement 4.a.

Requirement 4.b is satisfied by this stream sharing capability combined with the ability to discover streams produced by other operator graphs using the Stream Registry, which is described in the following subsection.

4.1.5 Requirement 5: Enable stream registration and discovery

Requirement 5 states that stream registrations can be dynamically added and removed at run time, and that they may describe streams with arbitrary and novel types. Furthermore, these stream descriptions may be queried at run time to allow stream discovery. These stream descriptions provide information about the stream, but do not necessarily describe the stream structure. Specifically, this means that the system does not require a description such as "the first 32 bits are an integer describing this, and the next 64 bits are a floating

[7] Deregistering is not without its dangers and should be used with caution - specifically, bad things happen to applications that happen to be running on that Worker at the time. While the system can check for this condition in some cases, such as the graceful shutdown of the Worker daemon process, it cannot prevent, e.g., someone unplugging the hardware on which a Worker is running.

point number describing that," but rather just describes the stream contents, such as, "video at 720p resolution with H.264 encoding." The way to handle stream contents based on the provided description is left as a matter for standards to dictate. This is helpful for complex streams, such as video, where it is easier to describe the stream and leave it to a standard to specify the byte-wise structure of that stream than to require AoT users to have to describe "the first four bytes are the first pixel, the next four are the second pixel, etc." We also note that some existing standards, such as that for H.264-encoded streaming video, may be leveraged.

4.1.5.1 Stream Registry

The Stream Registry is shown in Figures 3 and 4. A stream may be registered with the Stream Registry by specifying the handle (i.e., the connection information needed to produce or consume data on the stream), a unique name, the stream type, and the sensor type. Application developers may then query the name, stream type, and sensor type to find streams (from sensors or analysis results) that are of interest to them.

Stream types describe the contents and format of the stream, whereas sensor types describe the properties of the sensor producing the stream[8]. For example, a camera may produce a stream whose type is video with a resolution of 640x480, at 10 frames per second, MPEG encoded, etc. The sensor type for the same camera may describe properties such as the sensor's type (e.g., camera), capabilities (e.g., color and PTZ[9]), and location. The operator descriptions use the same concept of stream type when describing the input and output streams. In fact, the output stream type of an operator can be copied directly when registering the stream type of its output stream.

A very large variety of sensors exist in the Internet of Things, including many yet to be invented. Therefore, it is important not to define the specific properties used to describe stream and sensor types, but rather to provide an extensible stream and sensor description language. We originally proposed using extensible collections of name-value pairs for this

[8] For fusion analysis results, the sensor type would be some derivative of the sensors whose data contributed to the analysis.

[9] pan-tilt-zoom control

purpose [22]. However, we now use JSON because a JSON object is, at its core, a set of extensible name-value pairs. However, it also provides some limited data typing for values and allows non-primitive types, such as lists, arrays, and nested JSON objects.

Extensible attributes create the problem of how to interpret a description. For example, a video stream's resolution could be represented in JSON as $\{resolution : 640x480\}$, $\{resolution\_x : 640, resolution\_y : 480\}$, or $\{resolution : \{x : 640, y : 480\}\}$. We envision the community of sensor and other device manufacturers, developers, and other users creating standards for what names and structures should be used to describe different types of sensors and streams, and how to interpret their values. Our design goal is simply to provide the extensible property naming system that enables such standards to be created and used.

4.1.6 Requirement 6: Enable connecting operator graphs to arbitrary external inputs and outputs

There are a wide variety of heterogeneous sensors and actuators available today, and even more variety coming in the future. Requirement 6 states that AoT applications should be able to connect to arbitrary external inputs and outputs (i.e., that are not other operators). This is necessary to provide interoperability with the variety of sensor and actuator devices, as well as other data sources and sinks. To support heterogeneous external devices, we take a cue from traditional operating systems that help applications to deal with the wide variety of different types of computer hardware including different models from different manufacturers, in other words, device drivers. A hardware manufacturer can provide a device driver to an operating system for their hardware, which gives the operating system the ability to interact with the hardware and present applications with standard interfaces for using hardware of different types. We propose a similar model of "drivers" for AoT, where the creators of external sources and sinks (e.g., sensors and actuators) can provide code modules that allow AoT applications to interact with them[10]. However, we do not need to create a new mechanism for these "drivers" - rather, a driver is simply a particular

[10] There are also sensors and actuators that use one or another protocol or standard, but even then there is no single standard that all IoT devices use. We anticipate the community-driven creation of AoT "standard" drivers that allow AoT to interact with devices using common standards or protocols.

kind of operator and need not be treated any differently by the system than other operators, as will now be described in detail.

4.1.6.1 Source and Sink Drivers for External Sensors and Actuators

It is simple for an operator graph to take input from sources that use the same streaming system (i.e., Stampede^{RT}). The sources merely need to register themselves with the Stream Registry, and then the operator graph description specifies those stream names as operator inputs. Likewise, if a sink can read data from the streaming subsystem then it is simple to connect it to an operator graph's output. The output stream will automatically be registered with the Stream Registry, so a sink can either use the stream name or perform a query to find the stream to consume from.

However, it is also possible for an operator graph to receive input from sources (e.g., sensors) and provide output to sinks that use other communication methods. Since operators can run arbitrary code provided by the operator developer, special operators may be created that use custom code to connect to other inputs or outputs that do not use the streaming system. We call such operators *drivers*, and specifically, operators that connect to sources are *source drivers* and operators that connect to sinks are *sink drivers*.

For example, if a sensor provides data using a custom protocol over UDP, an operator could be created that takes zero inputs[11]. The operator container will call the operator method repeated in a loop, and the developer-created method could receive a UDP datagram, parse the custom application-level protocol to extract the desired information, serialize it into a data item, and submit the item to its output stream. Since the output stream is registered with the Stream Registry, this additionally allows the sensor data to be shared. The use of such driver operators is demonstrated in the example applications shown in Figures 1 and 2.

Since operators can be shared easily, we envision that the community will create publicly available source and sink drivers that use standard protocols. Furthermore, sensor, actuator, and other device manufacturers may develop AoT drivers to provide to customers along with

[11] meaning zero input *streams*, because it takes it input by another method

their hardware, just as many device manufacturers do for OS drivers today.

4.2 Experimental Results

Our previous work [22] presents experimental results on the computational performance of the MediaBroker. However, we do not present those experiments here because they are somewhat dated in that the hardware used is old by today's standards, and the implementation used at the time is not the current implementation, described in Section 4.1. Still, the new code base is not sufficiently different from the MediaBroker in relevant ways so as to make repeating these experiments intellectually interesting. Therefore we merely suggest that the general trends of the MediaBroker experiments still hold, and briefly summarize them now.

The results from our previous work [22] show that performance scales well (nearly linearly) as more operator graphs[12] are executed simultaneously, up to the point where the computational hardware is saturated. After that point, performance degrades gracefully according to how overloaded the hardware becomes. The results further show that exploiting the natural parallelism in operator graphs improves performance up to the point where the degree of parallelism exceeds the available hardware resources, at which point performance becomes only marginally better than executing each operator graph's operators in serial. Finally, we showed that sharing computational results can greatly improve performance when it is possible to do so, as it reduces the total computational load by avoiding repeated, identical computation in different operator graphs.

We now present experimental results for the performance of the Stream Registry. We note that this again is not the same code base described above, and in particular has some features (namely, continuous queries) not included in the current code base due to their being a unnecessary for the primary purpose of this work. It also used a RESTful interface rather than RPC. Therefore, we suggest that it is fully reasonable that the current code base, being a simplification of the one used in these experiments, has equal or better performance.

These experiments were performed using the Amazon Elastic Compute Cluster (EC2).

[12] operator graphs were called "dataflow graphs" in the MediaBroker

We used 64-bit Linux systems running on Extra Large Instances (4 virtual cores with 2 EC2 Compute Units (ECUs)[13] each, and 15 GB system memory) [1]. The server ran on one Extra Large Instance, while the clients ran together on another Extra Large Instance. Running the clients together facilitated the communication of timing information that had to be correlated in order to find the total delay for callbacks, as well as allowing the rapid execution of large numbers of calls.

Although the sensor registry system can be run as a RESTlet server, exposing REST APIs to the clients for maximum interoperability, we instead chose to run the system as a Java RMI[14] server with clients calling RMI APIs. This choice was made due to the relative performance of RESTlets and RMI. RESTlets have a significantly higher mean delay and variance (on the order of hundreds of milliseconds) that would dominate the results, masking the true overhead and scalability of the backend system with that of RESTlets. Although applications may submit RESTful callbacks to the system, we instead chose to use RMI callbacks for the same reason.

In our first experiment, we measured the time required to make query API calls to the system. The experimental client first registered a variable number of sensors' metadata then measured the mean time to submit 4096 dynamic, continuous queries.

Figure 5 shows how the mean time to make the API calls in milliseconds scales with the number of sensors registered in the system. Each data point represents the mean of 4096 calls, and each has a 95% confidence interval of ± 0.023ms or less. The figure shows that query API calls scale linearly with the number of sensors registered with the system ($R^2 = 0.997$). We omit any discussion of the "Update Query" and "Stop Query" APIs, as they pertain to the continuous query feature that is present in the earlier version but not in our current implementation.

For our second experiment, we measured the time required to make sensor metadata registration API calls. The experimental client first started a variable number of continuous queries. Next, it measured the mean time to register 4096 sensors. Then it updated each

[13] "EC2 Compute Unit (ECU) – One EC2 Compute Unit (ECU) provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor." [1]

[14] Remote Method Invocation

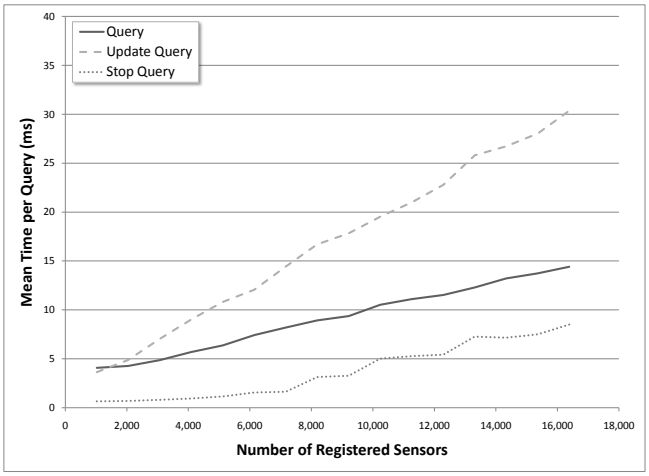

Figure 5: Time to Query the Stream Registry

of the sensor's metadata and measured the mean time to make the update calls. Finally, it measured the mean time to make the call to remove the sensor's metadata, before stopping all the continuous queries running in the sensor registry. Since our current implementation does not use continuous queries, we direct your attention to the data with the minimal number of continuous queries running.

Figure 6 shows how the mean time to make the API calls in milliseconds scales with the number of continuous queries running in the system. Each data point represents the mean of 4096 calls, and each has a 95% confidence interval of ± 0.0092ms or less. The figure shows that all three registration API calls scale linearly with the number of continuous queries registered with the system ($R^2 = 0.985$, 0.990, and 0.960 for the Register, Update, and Remove calls, respectively).

The Register and Update sensor calls take time proportionate to the number of continuous queries running because both operations require a full check of the sensor metadata (or newly updated sensor metadata) against all the continuous queries. Removing a sensor registration is nearly a constant time operation because it does not need to run the continuous queries. In our experiment, it does scale very slightly with the number of continuous queries only because each sensor matches a number of continuous queries proportionate

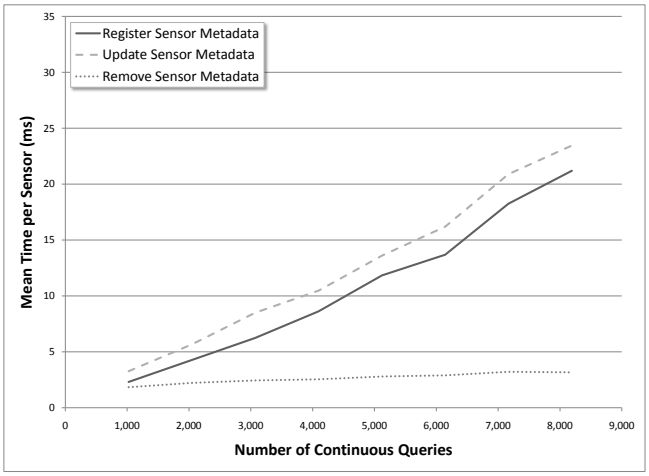

Figure 6: Time to Register / Update / Remove Sensor Metadata in the Stream Registry

to the number running, and the Remove call needs to send a callback to each matching continuous query.

In summary, these experiments show that the sensor registry system can scale to thousands of continuous queries and tens of thousands of registered sensors, while maintaining acceptable performance. Specifically, delays are on the order of tens of milliseconds for both API calls and callbacks, before accounting for the overhead of RESTlets.

In our current implementation that does not include continuous queries, we expect to see constant performance similar to the data present for the minimal number of continuous queries. For these operations, as well as the query operation presented earlier, the performance of the underlying database management system (DBMS), MongoDB in both our implementations, is the limiting factor.

CHAPTER V

FEDERATED ANALYSIS OF THINGS

The system presented in Chapter 4 is a useful tool to provide tightly-coupled distributed resources, as in a data center, for the Analysis of Things. However, in order to make the Analysis of Things possible within the Internet of Things, issues of scale must be addressed. Not only does the system need to scale with the computational load, available computational resources, number of operators available, and the vast number of sensors in IoT, it must also scale geographically. In this chapter, we present our design for a federated system to support AoT over a widely distributed area, thus fulfilling the final requirement 7.

5.1 Motivation

The Internet of Things is not something that will occur in just one location, nor in several discrete locations isolated from each other, but rather it will be pervasive. Sensors will be abundant in all environments and locations, and it will be useful to incorporate sensors in different areas into the same computation.

For example, data from the various sensors in the many cars on the road, along with roadside sensors may be analyzed on a local stretch of road to detect hazardous situations and warn drivers (or perhaps even allows cars to automatically apply the brakes). This will not happen only on one small stretch of road, but on every mile of most of the roads around the globe. In addition to these local decisions, sensor data may be analyzed across an entire region to provide dynamic traffic engineering, such as controlling signals and instructing vehicles to take different routes or drive in different lanes to improve the overall flow. Furthermore, data may be analyzed across entire metro areas, counties, or states to assist departments of transportation, urban planners, and government officials in making decisions about transportation maintenance and improvements.

Consider also a retail analysis system, such as the one being deployed in Walmart [20].

Sensors can monitor store shelves to observe which products customers are looking at, interacting with, and ultimately purchasing. Local analysis could, for example, determine when a customer is considering a product and use an on-shelf display to inform the customer that a similar item has a coupon available and even display a QR code allowing the customer to collect the e-coupon for the item. Store-wide analysis can help with inventory management, as well as determining popular items. Store-wide and regional analysis can help determine product placement strategies (e.g., which products should get the prime shelf space) and product synergies (e.g., customers who buy this item also tend to buy that item). Company-wide analysis, correlating data from all stores nationally or globally, can help determine corporate product strategies, advertising strategies, and so on.

These examples typify the Analysis of Things applications that will be present in the Internet of Things, and they illustrate several important points. First, sensors will be incredibly numerous and will be everywhere. Furthermore, their number ensures that the amount of data produced will be overwhelming, when taken in aggregate across all sensors. Second, the environment is hierarchical, with AoT applications performing analyses that correlate sensor data across local, small regional, large regional, and national and global areas. Third, live analysis on live streaming data must produce timely results. For example, the retail application must offer the customer a coupon while the customer is standing in front of the shelf, making a purchasing decision. More critically, a vehicle safety application must alert drivers to problems in time for corrective action to be taken (e.g., braking).

For these reasons, it is important to perform computation locally, near the sensors, actuators, and user devices, while still allowing regional and global analyses to consider sensor data from diverse locales. To illustrate the need for local processing over centralized processing, we present the following experiment showing the latency to transport data to the cloud for computation, and the results back again.

5.1.1 Network Latency Experiments

One potential roadblock to using a public cloud for the Internet of Things is the latency of sending streaming data over the Internet to reach the cloud. Streaming even heavy-weight

data, such as video, across the Internet has become commonplace. However, while many services such as Netflix and YouTube stream a great deal of video every day, they are prerecorded streams. Even in the case of "live" streams intended for viewing, the primary QoS metrics are jitter and quality – things that affect the viewing experience. A small delay of a few seconds is permissible even for "live" streams, if it allows some buffering that reduces jitter. In contrast, the primary QoS metric for IoT applications is latency. In this way they are more comparable to interactive streaming applications, such as video calls and video conferences, where a delay in the video stream can make the interaction cumbersome. However, video conferences normally consist of a small number of streams, whereas each IoT application may scale out to hundreds or even thousands of streams. In this way, they have a unique characteristic among streaming applications in that they combine latency sensitivity with scaling out to large numbers of streams. To investigate the performance of these applications on current networking systems, we ran two experiments with a video application to quantify the streaming latency to Amazon EC2 and found that the latency for streaming to the cloud is directly related to the raw ping-time to the cloud, and scales gracefully both with video size and the number of streams. Thus, we conclude that the ping latency is an accurate predictor of the streaming latency.

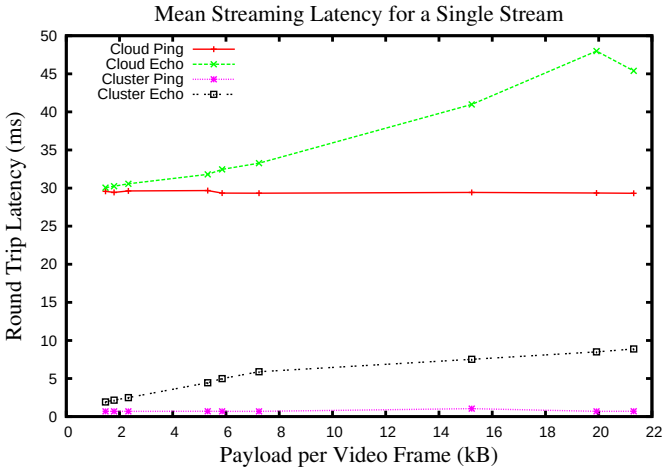

Figure 7: Latency Scaling with Size of Video

We instantiated a Cluster Compute instance in Amazon's Virginia data center (us-east-1a region) for these experiments. To test the effective latency, we ran an experiment streaming our actual application data from the Georgia Tech campus in Atlanta to both the cloud and to a local cluster on campus. Streaming to the cloud traverses the public Internet, and streaming to the local cluster remains on the campus network and traverses two router hops. To get a baseline, we first performed a simple ping, which can be seen for both the cloud and the cluster as the horizontal lines on Figure 7. Next, we created a simple process that receives streamed video frames and immediately returns them to the sender. At the video server, we computed the round trip time per frame. Video was sent at 10 frames per second. The results of this can be seen in Figure 7 as "Cloud Echo" and "Cluster Echo". As the graph shows, latency per frame to the cloud scales up with the size of the video frame in much the same way as it does when streaming to the local cluster.

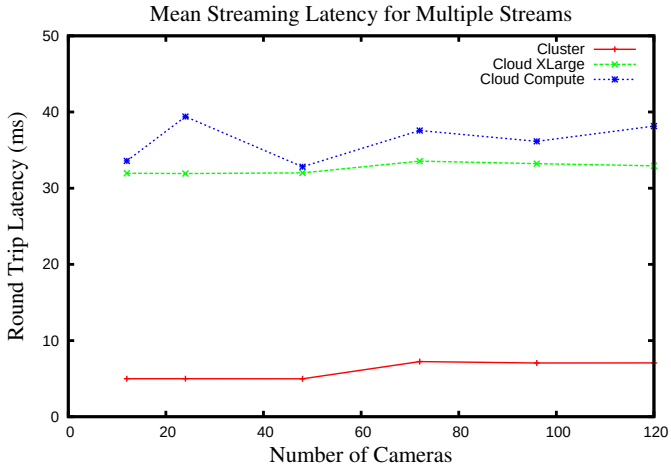

Figure 8: Latency Scaling with Multiple Streams

Next we wanted to examine how the latency is affected as we scale up the number of video streams. For this experiment, we hold the video size constant (just under 6 KB per frame and 10 fps, or 480 kbps bitrate) and scale up the number of concurrent video streams. We also scale up the number of receiver nodes proportionately to the number of video streams, so that the bandwidth of the receiver hosts is not the scaling bottleneck.

We compare the round trip latency for Cluster Compute instances (Cloud Compute), High-CPU Extra Large instances (Cloud XLarge), and our local cluster (Cluster) in Figure 8. This shows that the latency per stream holds constant as the number of concurrent streams increases in the cloud as in the cluster.

Finally, we wanted to examine the source of the added latency to the cloud to discover how much is inherently due to the cloud environment, and how much is merely due to traversing the public Internet. To accomplish this, we performed a bidirectional traceroute and found the last hop before entering Amazon's data center. We then compared ping times from our streaming source to the last hop router with ping times to our running instance inside the cloud. As expected, the difference in ping times is minimal, indicating that there is nothing inherent in cloud technology or the data center that adds to latency. The difference in streaming latency between the cloud and the cluster is merely due to the physical limits of network distance.

However, we also observed that our traffic traverses the Internet 2 to get to Amazon's data center. We therefore suggest that the 30 ms gap exhibited by our experiments is a best-case scenario. Other sources that are not as well connected will likely face higher latencies and more variance.

From this, we conclude that the latency differences for streaming to a centralized cloud data center over streaming to a local cluster are governed strictly by the raw ping time (RTT latency). The cloud data center itself does not add substantial network latency beyond that encountered getting to the data center, latency scales evenly with the number of streams, and increases gracefully with the stream size (bitrate) in the cloud as it does in the cluster.

As a result, we conclude there is little that can be done to improve the latency characteristics of the cloud. Rather, reducing latency requires reducing the network distance from sensor data to the computation. In other words, computational resources near the network edge are needed.

5.2 Approach to Federation

Federated SSIoTa consists of a number of "sites", which is an abstraction that includes one Stream Registry, one Operator Store, and one Resource Manager (with it's set of managed Worker Nodes), and has a concept of geographic location (i.e., where it is located in the physical world). Each of these three major components will be federated between sites independently, so when a user or component must interact with one of the components, it will just talk to the local component in the same site. That component will then handle federated communication and operations behind the scenes.

The federated Resource Manager assigns the operators of an application to different sites for execution using a fully distributed heuristic algorithm, named DistAl[1]. To better support multitenant applications running concurrently, the Resource Manager allows a three-way tradeoff between quality of results (which we consider as a combination of the data rate and the output fidelity), application performance, and resource consumption. Applications that are submitted for execution will include an operator graph specifying the computation to be done, a fidelity table for each operator that specifies the different fidelity levels at which they are able to run, and a utility function specific to that application to quantify how resource usage, performance, and quality of results can be traded off against each other to best meet that application's needs. DistAl finds a good schedule by attempting to maximize the total utility for all operators in all running applications in the federated system, which it does by making local decisions with partial system knowledge.

For the Stream Registry, streams will be tagged with a location and that location will be used to determine at which site to register the stream. Queries will include a region specifying a restricted area within which to query for streams. The region will be used to direct the query to sites that cover a portion of the region, and those sites will filter the results using traditional database techniques on the remaining query criteria. The Stream Registry instance at which the query was placed will concatenate the query results from all sites before returning them to the requester.

[1] DISTributed anALysis

Operators, however, do not have the same natural mapping to physical space. Indeed, there is no sure way to place operators in the sites where they will be used in the future since they do not have any properties that may give us clues. Therefore, we recommend a generic distributed database-based solution for the Operator Store, but note that temporal locality can be expected in operator access and therefore caching may be of some benefit.

5.3 Federated Operator Store

Since operators do not have any notion of geopysical location, there is no natural way to distribute them among sites to achieve better performance or scalability. Distributed Hash Tables (DHTs) are one way to scalably distribute key-value pairs among widely distributed peers [28, 29, 31], which could allow retrieval of an operator's code module using its unique name. Most DHTs are able to route a query to site that holds the data in $log(n)$ hops (for n = the total number of sites), although Content Addressable Networks [28] uses $(d/4)(n^{1/d})$ hops (where d = the number of dimensions in key space). Either option exhibits sub-linear scalability.

However, we believe that caching can help improve performance even further. Specifically, we anticipate that operators are most likely to be used again at locations 1) from which they were initially added (i.e., were the *put*() was issued, which is not necessarily where it will be stored), and 2) at which they were recently used in the past (i.e., where *get*()s were recently issued). The reasoning for #1 is that an operator may have been added at a particular site because the user wishes to use it immediately. For #2, an application may use multiple instances of the same operator, each on different input streams. Furthermore, a user may execute several operator graphs bearing some similarities (i.e., using some of the same operators) within a short span of time. Therefore, we propose to exploit this *temporal locality* by caching operators at the local site when they are 1) added to the Operator Store, and 2) returned by an Operator Store query submitted through this site.

5.4 Federated Stream Registry

Since a geographical area is used to route the query to sites that may hold some of the data, and filtering based on the remaining criteria is performed locally at the sites, a stream query

essentially reduces to the problem of being able to find the correct site(s) that correspond to any location or region. R-Trees [15] and Quad-Trees [30] are commonly used to allow spatial queries in distributed databases. However, these distributed databases are designed to support spatial queries but not to be spatially distributed themselves. That is, they are designed for distribution in a close area, such as a data center, not for wide distribution across the Internet. Specifically, they tend to use one of two solutions for storing the spatial tree structure: 1) a singleton tree in a centralized location, which doesn't scale, or 2) a fully replicated tree on each database node, which can work in a tightly-coupled environment but is not suitable for wide distribution where nodes may join and leave.

A scalable, self-organizing method of distributing the data among sites is needed. Distributed Hast Tables (DHTs) [28, 29, 31] provide these features, as well as being robust against node failures and providing load balance among the nodes[2]. However, to support a federated Stream Registry, they should also allow two-dimensional range queries (i.e., querying a geographic area) and provide *geographical locality*. On the other hand, load balance requirements could be loosened for SSIoTa since sites may (and indeed are expected to) provide unequal amounts of resources. Individual sites may use local distribution techniques, such as those mentioned above, to scale resources locally. Furthermore, load balancing is necessarily traded off against performance scalability and geographical locality. Existing DHTs ignore geographical locality in order to achieve the other two fully, but we suggest that a federated Stream Registry should instead allow a degree of load imbalance in order to fully provide scalability and geographical locality. A summary of DHT techniques, including our SkipCAN design, is compared against these requirements in Figure 9.

Geographical locality is a concept that we introduce in order to express the requirements of the federated Sensor Registry. Traditional DHTs assume that queries for any particular data item are randomly and evenly distributed, as depicted with the red dots in Figure 10. Therefore there is no performance benefit to storing the data in any one particular DHT node over another. The subset of data these queries are looking for may be stored on the node circled in blue, even though the sensors themselves are far from that node.

[2] under the assumption that all data items are equally popular

Feature	Reqs	Traditional DHT	No Hash DHT	PHT	SkipNet	SkipCAN
Self-Organizing	✓	✓	✓	✓	✓	✓
Scalable	✓	✓	✓	✓	✓	✓
Robust vs. Node Failures	✓	✓	✓	✓	✓	✓
Load Balance	✗	✓	✗	✓	✗	✗
Range Queries	✓	✗	✓	✓	✓	✓
2-Dimensional	✓	✗	✗	✗*	✗	✓
Geographical Locality	✓	✗	✗	✗	✓	✓

Figure 9: Requirements for a federated Stream Registry compared against features of Distributed Hash Tables, including our SkipCAN design

However, we anticipate queries in the federated Stream Registry to exhibit a different pattern. Most queries will be searching for sensor streams near themselves, that is, near the place where the query originated. This is shown in Figure 11, where the red dots (representing queries) are mostly near the area where the sensors they are querying are also located. We refer to this locality property as *geographical locality*. Since those queries will begin their search at the local site, significant performance and scalability gains can be achieved by storing sensor stream data at the sites near the streams' locations, such as the node circled in blue in the figure. While worst case performance may still require as many hops as in traditional DHTs, a geographical locality-aware DHT could substantially improve average case performance.

5.4.1 Related Work in Distributed Hash Tables

A naive approach to supporting range queries would be to simply not hash the keys, which would group data with similar keys together, rather that distributing it randomly and evenly. This would allow a simple range query mechanism by finding the nodes that cover

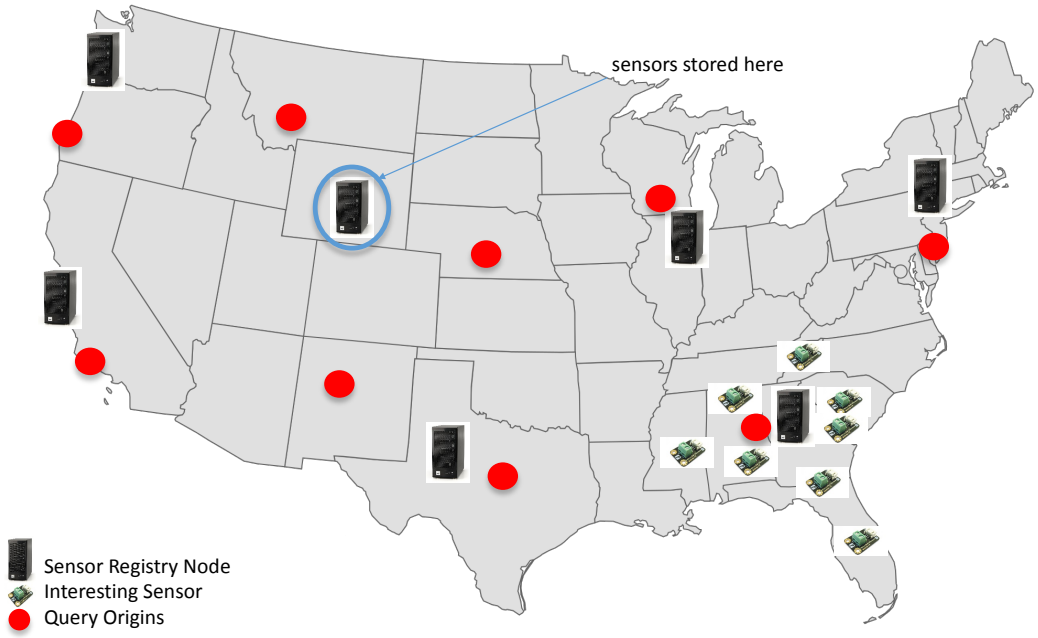

Figure 10: Distributed Hash Tables make an assumption of no geographical locality - queries for a particular set of data are randomly and evenly distributed

the needed range, but load balancing is lost when keys are not hashed, as indicated in Figure 9 by the "No Hash DHT" column.

Prefix Hash Tree (PHT) [26] is a solution to allow range queries in a DHT without sacrificing load balance. A prefix tree, or trie, is built to index the keys. Each key is stored at the trie leaf node representing the prefix of that key. Since this is based on actual keys, not hashed keys, nearby keys are stored in the same trie leaf or an adjacent leaf (by in-order traversal of the trie leaves). The trie is maintained dynamically as keys are added and removed so that leaves are bounded by a maximum of B keys (where B is a configurable parameter). The trie leaves are then assigned randomly and evenly to the DHT nodes by hashing the prefix identifier of the trie leaves, thus achieving load balance. The trie data structure is distributed, allowing queries to be routed by the trie in $log(n)$ time (for n total DHT nodes). Since trie leaves hold pointers to the next and previous leaves (creating a doubly-linked list of leaves), a range query can be performed by finding the leaf containing the beginning of the range and traversing leaves in order to the end of the range. Alternatively, branches of the trie could be traversed in parallel to reach all

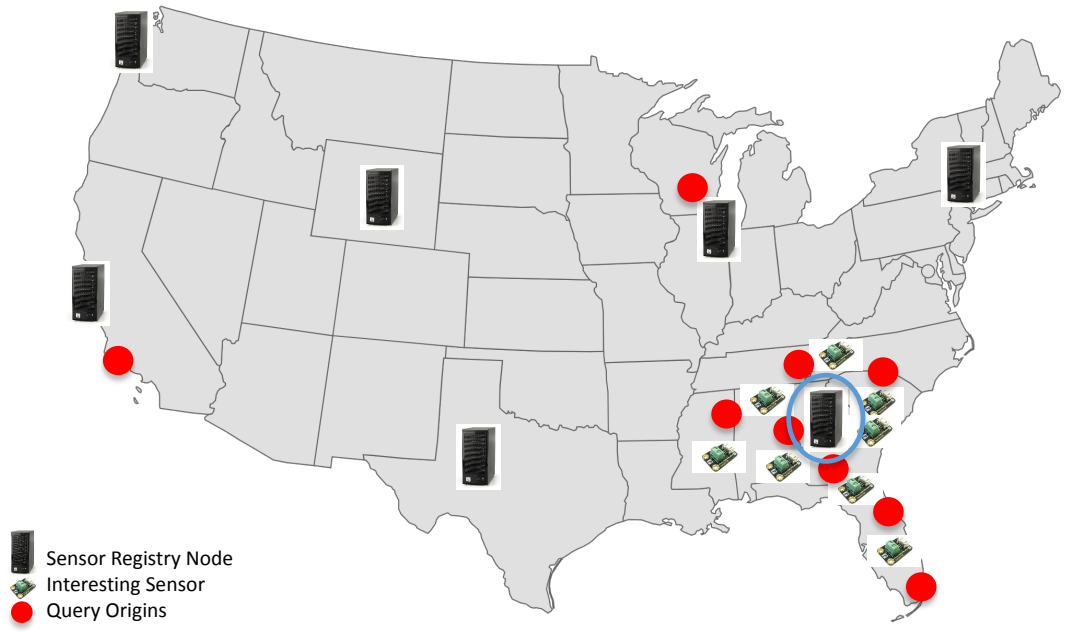

Figure 11: The federated Sensor Registry is expected to exhibit geographical locality - most queries will request data about streams near the query origin, i.e., most queries for a set of data will originate near that data

leaves that overlap the range, thus allowing range queries to be satisfied in $log(n)$ steps along the critical path.

However, since Prefix Hash Trees assign the key-containing leaves randomly to DHT nodes by hashing, PHT exhibits no geographical locality. Furthermore, it is not designed to handle two dimensional data. The authors briefly mention that linearization techniques (e.g., space-filling curves) could be used to collapse multiple dimensions into one in order to store them in PHT, but no detailed design is presented. Although the authors do not discuss multidimensional tries, we believe a quadtrie would be a reasonable way to index two-dimensional data such as geospatial coordinates, though this approach may not scale well to many-dimensional data.

SkipNet [16] is another DHT that supports range queries. Keys are assigned directly to DHT nodes without hashing, allowing easy range queries at the expense of load balance. Efficient lookup is achieved by using a distributed skip list, where each node knows the

next node at each level of the skip list, thus allowing retrieval in $log(n)$ hops. The primary objective of this work is to allow queries to retrieve keys that are part of their own administrative domain without the messages of the lookup protocol leaving that administrative domain. (This is desirable in a number of enterprise scenarios, for example, where a company may not want it's keys or get/put messages containing its keys traveling to or through another company's network or DHT nodes.) To achieve this, nodes also have an identifier in the same domain as keys. For example, both keys and nodes may be identified by a Uniform Resource Identifier (URI), and keys may be stored on a node who's URI is a prefix of the key's URI. The authors do not discuss geographical keys or locality, but if one were to use geographical coordinates as the domain for keys and nodes (rather than URIs), SkipNet would thus incidentally achieve geographical locality. However, support for multidimensional data and range queries is not addressed by SkipNet.

Content Addressable Networks [28] is one of the original DHT works from 2001, but unlike the others (Chord [31] and Pastry [29]), CAN natively supports multidimensional data. It works by assigning blocks of multidimensional key space to DHT nodes as follows: When a node joins the system, it chooses a random point in the multidimensional key space (thus ensuring nodes are probabilistically evenly distributed) and contacts the node currently holding the block in which that point falls. The two nodes then split the block between themselves. Keys are assigned to points in the multidimensional space through hashing, thus ensuring keys are evenly distributed and achieving load balance. Locating the node that covers a point is achieved by contacting any CAN node, and from there the request will be forwarded to a neighbor node along one dimension so that it is one step closer to the requested point. This is repeated until the destination node is reached. Although this superficially appears to be less performant than other DHTs, routing can be achieved in fewer steps if the number of dimensions is increased, which has the effect of bringing all nodes closer together by providing more dimensions to route through. Thus a query can be performed in $\frac{d}{4}(n^{1/d})$ hops (for d dimensions), which can achieve similar performance to other DHTs, depending on the number of dimensions. Since the multidimensional space is an arbitrary key hash space with no physical analog, dimensionality can easily be increased.

5.4.2 SkipCAN

We suggest a combination of CAN and SkipNet techniques for the Stream Registry. Since CAN naturally supports multiple dimensions, a two-dimensional key space can be used to represent geophysical space. However, since it now represents a physical concept, keys should not be hashed, but placed in the space according to their location coordinates. Similarly, nodes should use their own location rather than choosing a random one when they join the system, although splitting areas to add new nodes may otherwise proceed as in CAN. These changes allow us to have geographical locality but present two complications: Adding nodes and keys this way breaks the load balancing features, although this is an acceptable tradeoff, as discussed earlier. Also, we cannot easily increase the dimensionality since the space now represents a naturally two-dimensional [3] physical concept.

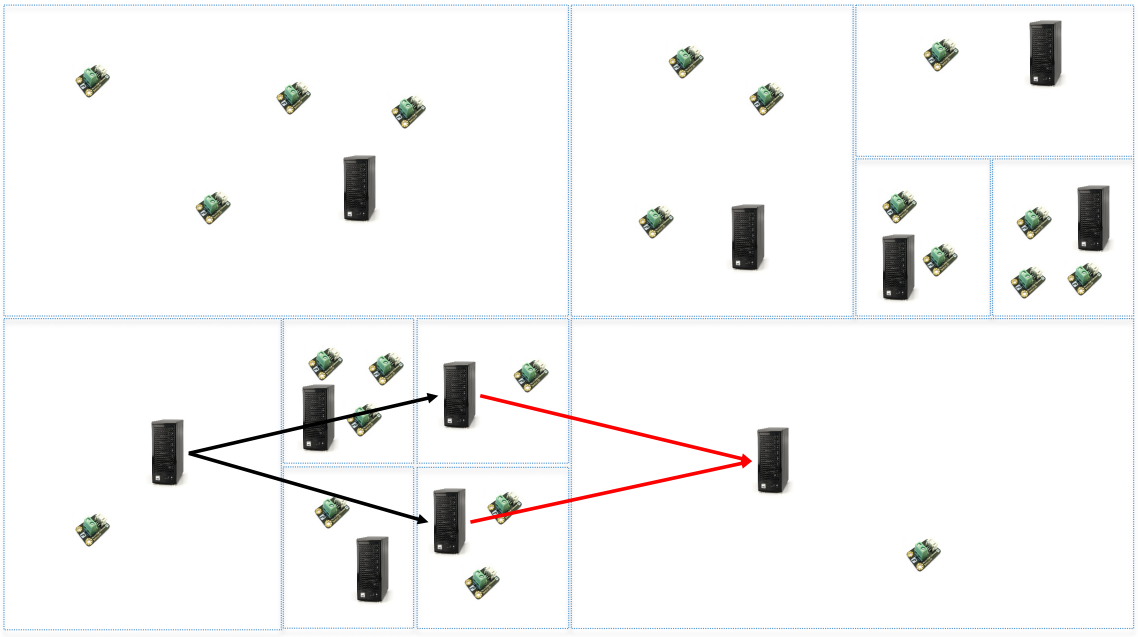

Figure 12: Multidimensional queries in SkipCAN may result in query splitting (black lines) and split queries may merge later (red lines), thus necessitating a unique query identifier for duplicate query removal.

Although leveraging geographical locality alleviates some of the performance scalability

[3] or possibly three-dimensional, depending on the spatial representation

concerns, we would nonetheless like to better bound the worst-case performance. To accomplish this, we suggest using a distributed skip list, as first proposed by SkipNet. Each node maintains one skip list for each dimension, thus allowing $log(N)$ performance per dimension. Thus each line that varies along only one dimension in the key space is essentially a SkipNet. Two special notes are warranted, however. First, the space covered by a node in $D-1$ dimensions may be covered by multiple nodes further along one of the skip lists, as shown in Figure 12, so it may have multiple neighbors at some level of its skip list. In this case, it simply forwards queries in parallel to all neighbors that overlap the query region in any dimension (but to a minimum of at least one if none overlap). Second, as Figure 12 shows, the reverse situation is also possible, so the parallel queries could be rejoined in the future. To prevent query multiplication, each query can be tagged with an identifier that allows nodes to identify and ignore duplicates of the same query. To ensure uniqueness, a query tag could be the hash of the node where the query originated, the timestamp when the query was issued, and the query expression itself. This ensures that equivalent queries can be issued from different nodes at the same time, or issued again from the same node at a later time, without being discarded.

5.5 Federated Resource Manager

The third and final SSIoTa component to federate is the Resource Manager. In order to federate the execution environment, all applications' operators need to be *scheduled*, or assigned to the different sites where they will execute. The Resource Manager at each site can then schedule them on specific worker nodes, as described in Section 4.1. We do not address stream transport explicitly because Stampede^{RT} is capable of transporting streaming data between sites as well as between worker nodes within a site.

5.5.1 System Model

In the context of our application space, *scheduling* means to create a mapping from a set of resources to multiple applications, where a "good" mapping does not violate any constraints (e.g., not exceeding the available CPU throughput of any resource), minimizes the end-to-end delay from application input to final results being ready, and also maximizes the quality

of the results (QoR).

5.5.1.1 *Application Model*

An application consists of a directed acyclic operator graph, $G^{app} = (V^{app}, E^{app})$, where the vertexes represent operator instances and the directed edges represent data streams from operator outputs to inputs. Each operator instance, $v_i^{app} \in V^{app}$, is a continuously running algorithm that takes one or more input streams[4] and produces one output. Each edge, $e_{i,j}^{app} = (v_i^{app}, v_j^{app}) \in E^{app}$, represents a data stream from the output of operator v_i^{app} to the input of operator v_j^{app}.

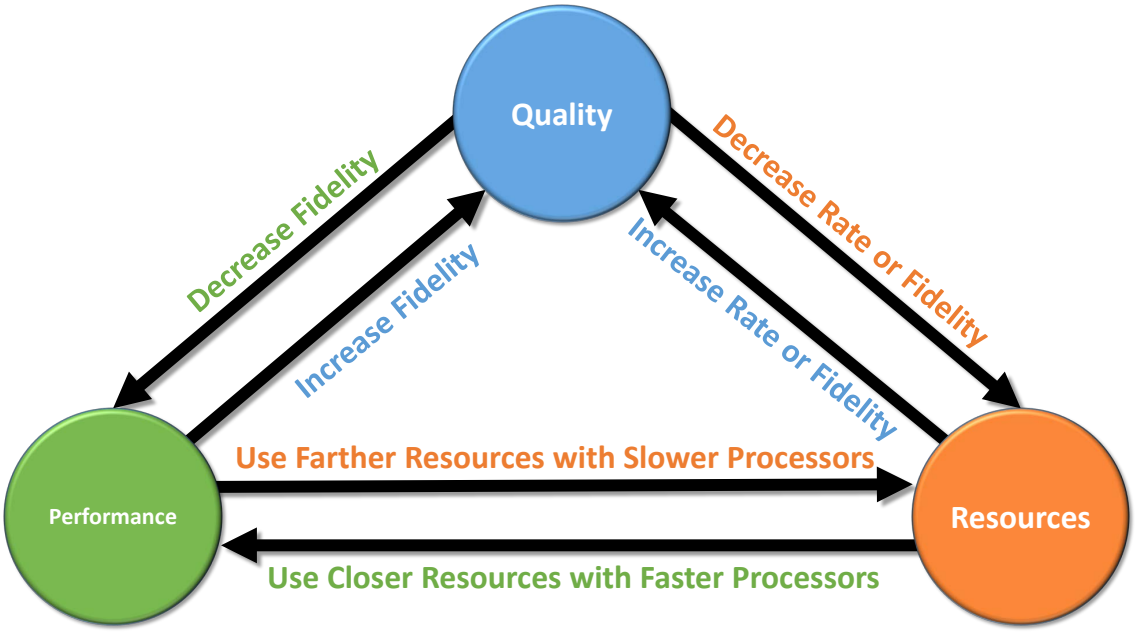

Figure 13: Tradeoffs between quality of results (QoR), application performance (end-to-end delay), and consuming fewer resources.

In addition to the operator graph, the application description contains information about the resource requirements for each of the operators. However, many operators are able to trade their *quality of results* (QoR) and performance against resource demands. For example, a computer vision algorithm may be able to process up to 30 frames per second

[4]Special *source* and *sink* operators have no inputs or no outputs, respectively. We use these as "stub" operators to represent the ingestion points for sensor data and the final destination for the application's output. In practice, these operators may be the *driver* operators for the external source and sink devices.

(fps), but could also process only 10 fps for a third the resource requirements. It may also be able to process video of different resolutions, saving resources at the expense of output quality at lower input resolutions. The quality of results is a combination of the data *rate* and *fidelity*. We assume these are adjustable for all operators and will use this fact, as described in Section 5.5.2.3, to accommodate multiple concurrent applications while meeting resource constraints. Each application also provides a *utility function* that is used to trade off performance and quality in order allow multitenancy and maintain constraints. This will be discussed further in Section 5.5.2.1.

Figure 13 shows the possible tradeoffs between quality of results (QoR), application performance in terms of end-to-end delay, and resource consumption. Resource consumption can be reduced by decreasing quality either in terms of the rate or fidelity. It can also be reduced by spreading the computation over more resource nodes, but this may come at a performance cost since more distant nodes may have to be used (adding network latency) and/or less powerful nodes (adding computational latency). Quality can also be traded to improve performance by reducing the fidelity since this will reduce the computational latency, although the data rate is not related to performance since we are considering end-to-end delay rather than throughput. In an ideal world, every application could run at full quality, finish instantly, and require no resources. However, in a real world where multi-tenant applications are sharing limited resources, some of these ideals must be sacrificed[5]. Each application may have different preferences for where it would like to fall in this triangle, and the ability to allow each application to specify its own utility function allows our system to take those preferences into account.

[5]Alternatively, admission control could help here, but one of our design goals is avoiding strict admission control.

For each operator, a table of possible fidelity levels must be provided, similar to the example shown in Table 2. This table includes the amount of resources required to processes a single input item and the fidelity value of the output for each fidelity level that the operator can support. The fidelity value is an abstract number representing the "goodness" of the operator's output. In our implementation, we used the convention that the best fidelity level always has value 100, and other levels are less, as appropriate for their proportionate qualities. Effectively, this can be thought of as "percentage of maximum fidelity". The fidelity value must necessarily be abstract because the meaning of "goodness" varies depending on what the operator does and what the data type of its output is. However, adopting a convention such as ours makes the fidelity values of different operators scale together and compare in a reasonable way.

Table 2: Fidelity Level Table for a Foreground Detector (FD) Operator

Level	*Cycles per Item*	*Value*
0	386 M	100
1	97 M	50
2	24 M	25
3	6 M	12.5

A complete application description, therefore, is $A = (G^{app}, F, u)$. G^{app} is the application's operator graph, F is the set of fidelity tables, and u is the application-specific utility funciton. $F_i \in F$ is a schedule of possible fidelity levels for operator v_i. The utility funciton will be described in greater detail in Section 5.5.2.1.

5.5.1.2 Resource Model

The resources available to run applications are represented by an undirected resource graph, $G^{res} = (V^{res}, E^{res})$, where the vertexes, $v_i^{res} \in V^{res}$, are compute nodes capable of running operators (i.e., SSIoTa sites). Each compute node has p_i processors (or cores), and each processor on node v_i runs at y_i cycles per second[6].

The edges, $e_{i,j}^{res} = (v_i^{res}, v_j^{res}) \in E^{res}$, form an overlay network between resource nodes. While we assume all nodes are reachable over the network from all other nodes for purposes streaming application data between operators, nodes that are connected in the overlay are

[6]Although we only consider homogeneous resources in a node, heterogeneous nodes may be modeled as two or more collocated nodes.

considered *neighbors* for purposes of our algorithm. This limits the scope of communication rather than nodes simply broadcasting to all other nodes in the scheduling algorithm. Finally, the network latency between every pair of nodes, $l_{i,j} \forall i, j$, is known even for pairs where there is no $e_{i,j}^{res}$ (i.e., that are not neighbors).

Since our algorithm runs continuously and responds to changes in the environment even during application run time, we do assume that resource nodes are able to perform live migrations of operators between neighbor nodes. Such migrations should be rare in the steady state. However, when a new application starts, its operators will likely experience numerous and rapid relocations until it has settled into a good operator placement. To address this, we propose simply moving references to operators rather than migrating real operators. After the application's placement has been optimized, the resource nodes with operator references can retrieve the code for the referenced operators from the Operator Store and begin executing the application. This does come at the cost of some initial delay when starting a new application, which we address in Section 5.5.3.2.

5.5.2 System Design

5.5.2.1 Utility Function

In order to allow scheduling multiple live analysis applications on the same set of resources without strict admission control, a sacrifice in performance and quality of results (QoR) is necessary. As shown in Figure 13, by reducing data rate or fidelity, resource requirements are reduced. Resource power can also be traded against application performance (i.e., using weak nearby resources vs. powerful but distant resources). To optimize the overall "goodness" of all running applications, it is necessary to quantify the value of application performance and quality of results. It is also necessary to understand how these two properties are valued relative to each other. However, each application may have different needs – fast end-to-end delay may be important to one application, while another may value accurate results more highly. To address this, we allow each application to provide its own utility function that quantifies the value (utility) of varying levels of performance and QoR. For purposes of this research, we assume all utility functions output values in the range

$[0, 1]$.

Since the system is fully distributed, the utility of an application must be calculated in a distributed manner as well, rather than at a centralized location. Therefore, the utility function provided by the application in fact computes the utility of an individual operator in that application, which allows utility to be calculated at a node using only local information. The utility of the total application is taken to be the sum of the utilities of all its operators and the total utility of the system is the sum of the utility of all operators running on all nodes. This also means that an improvement in the utility of a single operator corresponds to an equal improvement in both the application's utility and the total system utility, which is exploited by our algorithm, as will be described in Section 5.5.2.3.

5.5.2.2 Example Utility Function

An example utility function is shown in Equation 1. The four terms represent (from left to right) network latency, latency of computation, reduction in rate, and reduction in fidelity. U is the weighted average (with weights A, B, C, D) of these four terms, each of which is crafted to have a value in the range from 0 (bad) to 1 (good). Thus U itself is also in the range $[0, 1]$.

$\bar{l}$ is the average network latency (in ms) of all this operator's inputs, x/y is the time to run the operator on a single input item (x is the cycles required by the operator, and y is the processor speed), r_o/r_i is the ratio of output rate (r_o) to the rate of the slowest input (r_i), both expressed as the number of items per second, and f_o/f_i is the ratio of output fidelity value (f_o) to the fidelity value of the worst input (f_i). Both x and f_o come from the operator's fidelity table, as discussed in Section 5.5.1, and f_i comes from the input operator's fidelity table. The latencies used to compute $\bar{l}$ and the processor speed y come from the resource model.

$$U = \frac{\frac{A}{1+\bar{l}} + \frac{B}{1+x/y} + C\frac{r_o}{r_i} + D\frac{f_o}{f_i}}{A + B + C + D} \tag{1}$$

The first two terms in Equation 1 therefore represent the application performance (how much time is spent on the network and in computation, respectively). The latter two terms

represent the quality of results (data rate and fidelity). By adjusting the term weights, the relative value of these factors to a particular application can be specified. Not only can performance / quality tradeoffs be evaluated, but the relative value of data rate and fidelity are also captured, allowing DistAl to choose levels for both quality attributes as appropriate to the application.

While Equation 1 presents a function that increases smoothly with the four parameters (i.e., more is always better), it may be desirable for many applications to use a slightly more complicated utility function. For example, many applications may have a concept of "good enough" for the level of quality or performance, and therefore may want utility to be an S-curve with an inflection point around the "good enough" level. Applying $S(x) = 1/(1+e^{\frac{\alpha-t}{\beta}})$ to relevant terms should allow this variant. Furthermore, utility functions do not have to be continuous, so other variants could be created to provide sharp cutoffs (e.g., a step function). Many other variations are also possible, and this is just an example of how it is easy to customize utility functions in our system to suit different applications' needs.

5.5.2.3 Algorithm

DistAl runs continuously on all resource nodes, iteratively making one-step improvements in operator placement (i.e., by moving an operator to one of its neighbors), as shown in Algorithm 1, the high-level pseudocode overview of DistAl. (Complete pseudocode for DistAl is provided in Appendix B.) Since each step causes improvement in the total system utility, the algorithm converges over time, but since it continues to explore the space, it can dynamically adapt by finding and exploiting opportunities to increase utility when the environment changes (i.e., changes in the running applications or resource network). By using this approach, the algorithm is fully distributed, using only state known locally and to a small set of neighbor nodes that can easily be queried (line 8). This allows scaling with the number of neighbors, rather than the size of the entire resource network.

We know the algorithm converges, absent changes to the applications or resource network, because the system wide utility necessarily improves each time a change to the schedule is made, thus ensuring it always moves towards a maximum. While it is possible to

Algorithm 1 DistAl selects a local operator (line 2), queries all its neighbors to find the utility change for moving it to that neighbor (lines 7–13), and then moves the operator to the neighbor where it will give the greatest utility improvement (line 14). The delay between checks has a configurable duration (line 15).

```
 1: while true do
 2:     op ← ChooseOperator(host)
 3:     myUtilityDelta ← increase_quality_until_constraints(host)
 4:                      − UtilityFunction(op)
 5:     bestUtilityDelta ← −∞
 6:     bestNewHost ← ∅
 7:     for all neighbor ∈ Neighbors(host) do
 8:         neighborUtilityDelta ← QueryNeighborForUtilityDelta(neighbor, op)
 9:         if neighborUtilityDelta > bestUtilityDelta then
10:             bestUtilityDelta ← neighborUtilityDelta
11:             bestNewHost ← neighbor
12:         end if
13:     end for
14:     if bestUtilityDelta + myUtilityDelta > 0 then
          MoveOperator(op, bestNewHost)
15:     end if
          OptionalDelay(duration)
16: end while
```

converge on a local maximum in the short term, this condition persists only while the system state remains static. The dynamic nature of the environment over time affects the underlying optimization space, thus having the opportunity to perturb any states that are temporarily stuck. Furthermore, many classic methods of avoiding local maxima run the risk of either sacrificing scalability by increasing the number of nodes queried at each step, or losing the useful property that every step improves the total utility (i.e., there is never a "step backwards" in DistAl).

Each resource node continuously runs the loop described in Algorithm 1, where it chooses an operator to try and move, finds the neighbor that will gain the most utility by receiving the operator, and if that is more than the utility this node will lose by giving up the operator, it moves the operator to that neighbor. To find the neighbor with the most utility gain, this node queries all its neighbors to run the application's utility function on the candidate operator. A query handler on each node receives such query messages, runs the calculation, and responds with the utility change. Although our method to choose a candidate operator on each iteration uses a queue of newly arrived operators with an additional second-chance

queue, DistAl can also work with other methods of choosing an operator. A delay between iterations of different candidate operators can be used to control the rate of operator churn as well as the resource overhead required by the algorithm. The delay could even be variable, allowing more churn when new applications are being started and lots of moves are required, but reducing overhead when the situation is relatively stable and few candidate operators are expected to actually be moved.

When a node is asked to accept an operator, it may find that the new operator would exceed the resource constraints of the node. This is unacceptable since streaming analysis applications depend on throughput keeping up with their data rate. In this case, the node will adjust the quality levels of its local operators until the resource constraints are met, thus providing adequate throughput while avoiding strict admission control. Of course there is a cost to the utility when quality is reduced. Thus an iterative algorithm is again applied to minimize the utility loss while meeting the resource constraints. The node makes a one-step reduction in the rate or fidelity (but not both) of a single operator, choosing the step that provides the greatest reduction in resource pressure per amount of utility lost for the change, and repeats this until the constraints are met. The utility losses needed to keep within resource constraints are also accounted for when a node informs its neighbor of how much utility it will gain from accepting a new operator. Thus nodes with a lot of resource pressure are less likely to be given new operators since, while they may gain the utility of the new operator, they will lose utility from decreasing the quality of other operators.

Meanwhile, when a node is giving up an operator, this relieves resource pressure. It uses an algorithm similar to the one just discussed, only in the reverse direction, to increase it's operators' rates and fidelities until they are as high as possible without violating resource constraints. Each step chooses the change that maximizes the utility gain per additional resources used. Thus as operators pile up on a node and apply pressure to its resources, the probability of it giving up an operator increases since it has more opportunity to gain utility by increasing other operators' qualities when it gives up an operator.

Considering the utility function in Equation 1, the node placement algorithm affects the first two terms (network and computation latencies) by determining on which node an

operator is placed, and the quality adjustment algorithms affect the latter two terms (rate and fidelity). An application's choice of utility function therefore can affect whether it's operators prefer crowded but powerful, low-latency nodes, or nodes that are weaker and farther but have little competition for their resources. It can also affect how far away an application is willing to go in order to use a more powerful node (i.e., trading network performance against compute performance) and whether it should prefer to lose data rate or fidelity when quality must be sacrificed. Furthermore, maximizing the total utility in the system means that the needs of all running applications are collectively met as best as possible, according to the preferences expressed in each of their utility functions.

Since our algorithm is an iterative optimization algorithm, new applications need an initial placement before DistAl can begin to refine its operator placement. We use a naive algorithm to create a simple initial placement. It begins by pinning the sources to the nodes where the sensors are closest – DistAl will not move these since sensors are fixed in physical space. Similarly we pin the sink that consumes the analysis results to a node. Then operators that use the sources' data are placed on the same nodes, then operators that use those operators' outputs are placed on the same nodes, and so on until all operators are placed. Operators with multiple inputs will be placed on the same nodes as whichever of its inputs happens to come up first as the algorithm runs. As operators are placed together on the same nodes in excess of the node's resources, operator qualities are reduced until the resource constraints are met. Thus the initial placement tends to bunch operators together on the same resource nodes (incurring little network latency), tends to place them on nodes near the application's sensors, and tends to set them to lower quality levels than an optimal schedule. However, other initial placement schemes are possible and DistAl works with any valid initial placement.

5.5.3 Performance Evaluation

We have qualitatively argued that DistAl meets the desirable requirements for scheduling live streaming analysis applications on geographically distributed resources, namely (a) a fully distributed algorithm, that (b) continuously adapts to changes in the environment, (c)

meets resource constraints while placing operators, (d) avoids admission control (specifically, by trading off quality of results to meet resource constraints), and (e) uses a utility function to optimize performance and quality according to each application's needs.

To validate our design and algorithm, we implemented resource networks and applications in the OMNeT++ v4.6 simulation framework [33]. While it would be ideal to compare it to a theoretically optimal operator placement, computing such a schedule at the scale of resources and application size we're considering would have been infeasible. Therefore, we will demonstrate that the algorithm is effective by showing that it improves both utility and application performance from a naive initial state. We further consider the cost of our method by looking at the startup time required for DistAl to run to convergence.

5.5.3.1 Experimental Scenario

For our experiments, we use a randomly generated resource network and two different application structures. In the random network, 100 nodes each have a random number of neighbors with a mean of 6 neighbors (except for experiments where we explicitly varied this parameter). Computational resources are allocated to nodes by sampling processor speed and the number of cores each from a normal distribution of possible resource amounts – the mean processor speed is 2.8GHz and the mean number of cores per node varies for different scenarios as described below. Network latencies between neighbors are sampled from a normal distribution with a 100ms mean (except for experiments where we explicitly varied this parameter). All experimental results that follow are the mean results of 100 different resource networks that were randomly generated using the same parameters.

Our simulation models the resource network but not the full transport network beneath it. Therefore, the latency between two non-neighbor nodes was taken as the sum of the latencies along the shortest path in the resource overlay. This does not affect our experimental results except when varying the number of neighbors, since higher connectivity in the overlay graph thus reduces the latency between non-neighbors. We discuss the effect this has on our specific experimental results in Section 5.5.3.2.

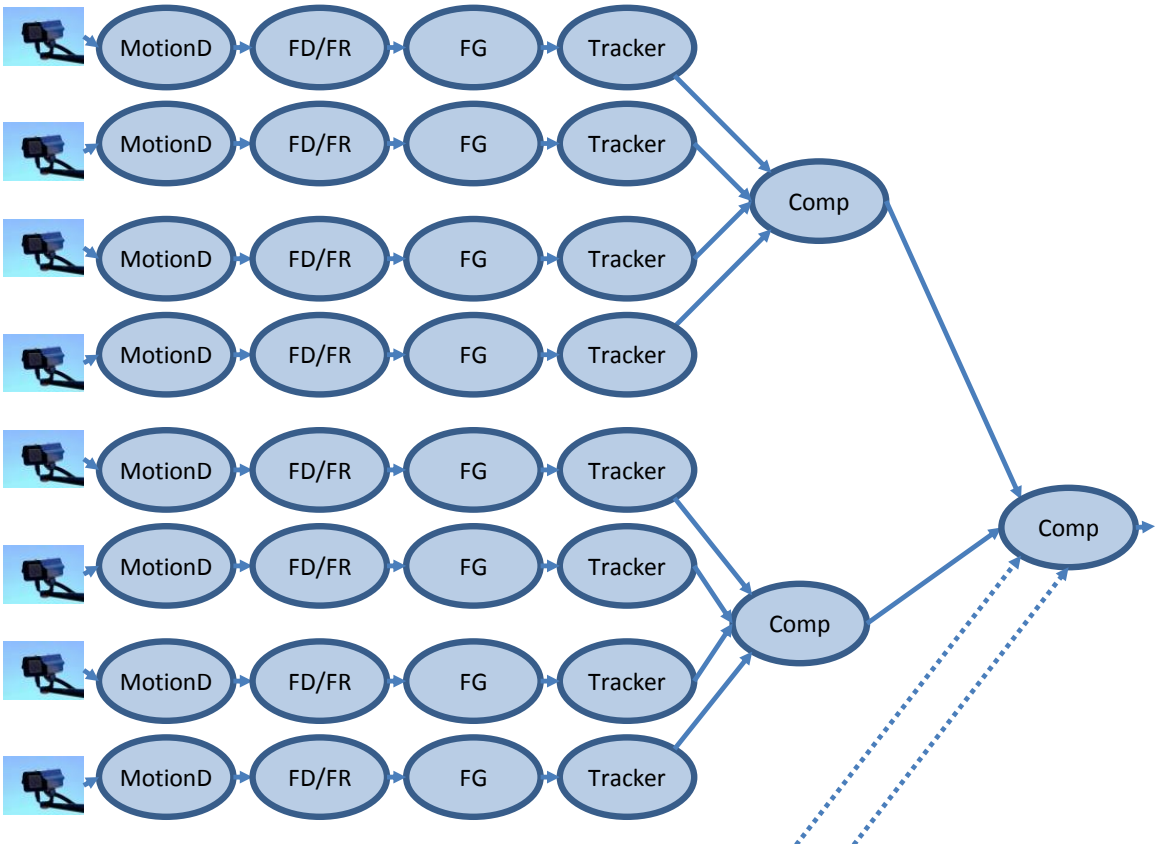

Figure 14: Kernel of "Parallel", an embarrassingly parallel suspect tracking application

For the applications, we have created two different kernels of the suspect tracking application that represent two different ways that parts of the application may be structured. The first application kernel, called "Parallel" and shown in Figure 14, analyzes video camera data in an embarrassingly parallel manner, consolidating results only as the analysis of each camera is completed. Figure 15 shows the second application kernel, called "Combining", that performs similar analysis, but attempts to consolidate the data as quickly as possible. The dotted line inputs in both figures indicate that there are additional operators at the lower levels that are not shown for readability, thus increasing the scale beyond what is shown in the figures (to a total of 124 operators in "Combining", and 171 operators in "Parallel"). Both applications use Equation 1 as their utility function, with weights $A = 3, B = 3, C = 5, D = 1$. Although we have not built a complete suspect tracking application, we believe these two are representative of the kinds of computation and application

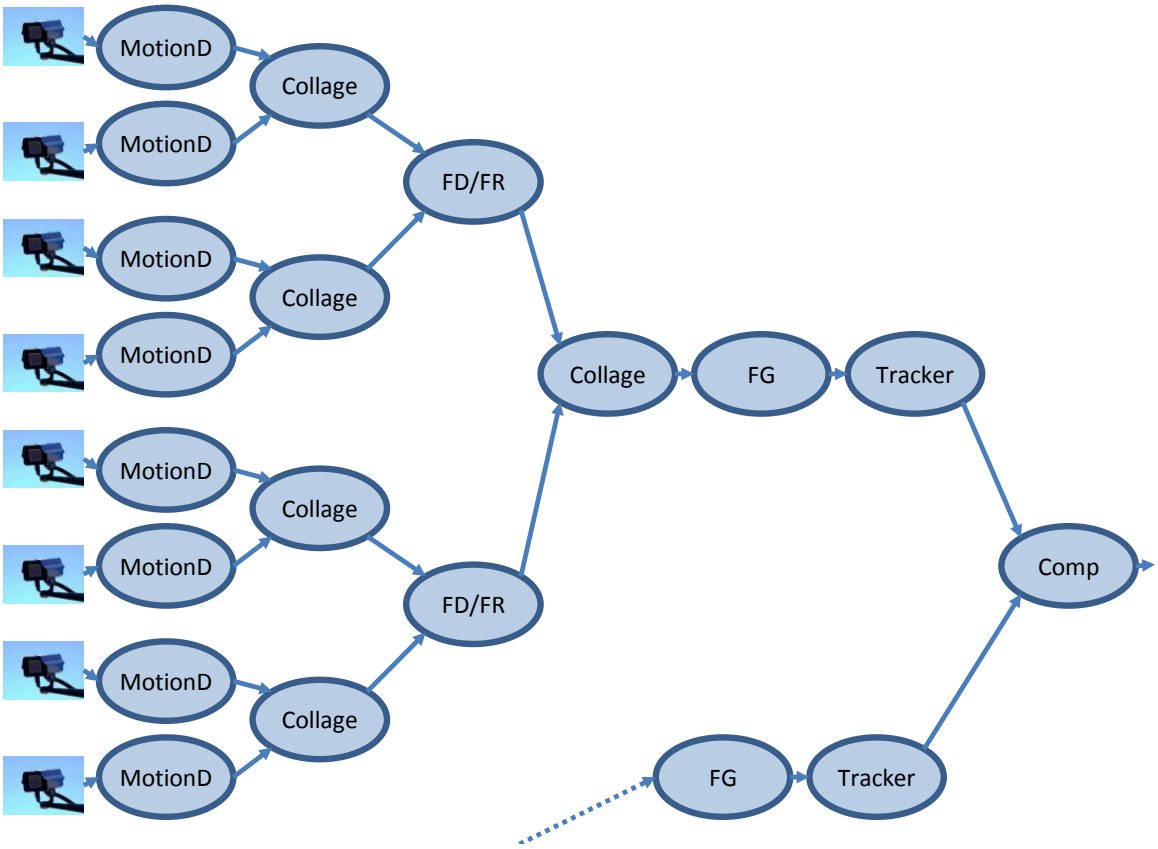

Figure 15: Kernel of "Combining", a suspect tracking application that fuses data

structures that can be expected in such applications.

In these applications, *MotionD* is a motion detector, *Collage* is a fusion operator that concatenates image data without filtering, *FD/FR* is a face detection and recognition algorithm, *FG* is a foreground detection operator that creates a foreground mask which is required for the Tracker, *Tracker* is an algorithm to track moving objects through a camera frame, and *Comp* is a comparator operator that can determine if an object in a camera's view is the same as an object recently seen in another camera's view. Table 3 shows the computational requirements to run each operator at full fidelity on a single item (i.e., video frame). An example complete fidelity table for FG is presented in Table 2, but the other operators' fidelity tables are omitted for brevity. The numbers for MotionD, Collage, and FD/FR are taken from Wolenetz et al. [34], while FG, Tracker, and Comp are from one of our previous papers [19].

We also vary the amount of resources available in the resource network according to the applications running in each experiment in order to produce three different scenarios: "Enough Local" is the scenario in which there are enough resources to run all applications' operators at full quality (i.e., rate and fidelity) at resource nodes very near the application's initial placement. For one instance of each application type running on the random network, the mean number of cores per node is 32. The "Enough Global" scenario provides enough resources globally to run all applications at full quality, but there is contention for local resources between nearby applications. There are 16 mean cores per node in the Enough Global scenario. Finally, the "Not Enough Global" scenario does not have enough total resources in the network to run all applications at full quality, so a significant quality tradeoff is needed simply to avoid admission control that would deny an application the right to run at all. Node resources were sampled from a normal distribution with a mean of 10 cores.

Table 3: Operator Resource Requirements

Operator	*Cycles per Item*
MotionD	1009 K
Collage	803 K
FD/FR	1959 M
FG	386 M
Tracker	154 M
Comp	200 M

5.5.3.2 Results

Although there are related algorithms in Grid Computing and Wireless Sensor Network research, their inability to trade off quality of results or support multitenancy with application-specific utility functions prevents them from being directly comparable with our algorithm. Therefore, to demonstrate the effectiveness of our algorithm, we show the improvement in utility, quality, and application performance from the initial operator placement (before DistAl is run) to the final placement (after DistAl has converged), averaged over all applications running. The application performance metric is end-to-end delay, defined as the longest latency path from any sensor input until the result arrives at the sink, including both the network latencies and the latency of computation for all operators.

Figures 16 and 17 show the improvement (as percent increase) in the mean per-application utility. Our algorithm is effective at improving the utility in resource-rich scenarios (Enough

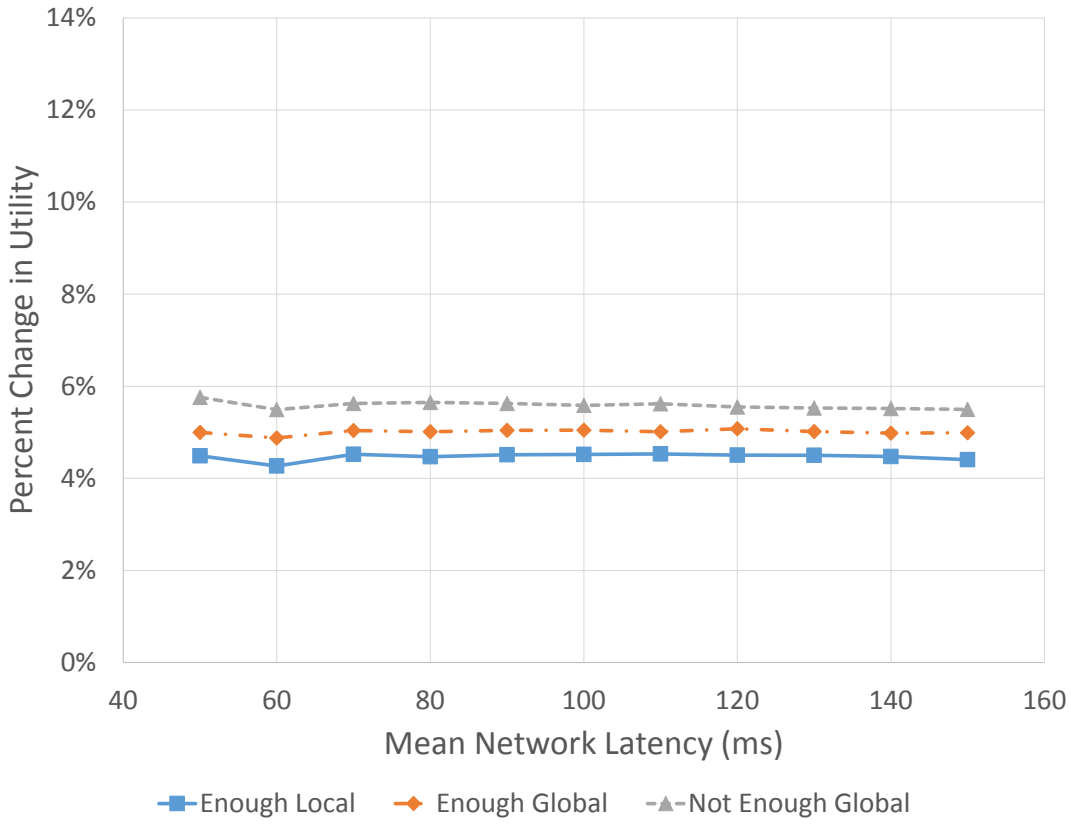

Figure 16: Percent Change in Average Application Utility from Initial Placement (before DistAl) to Final (after DistAl) as a Function of Network Latency on a Random Resource Graph with Both Applications

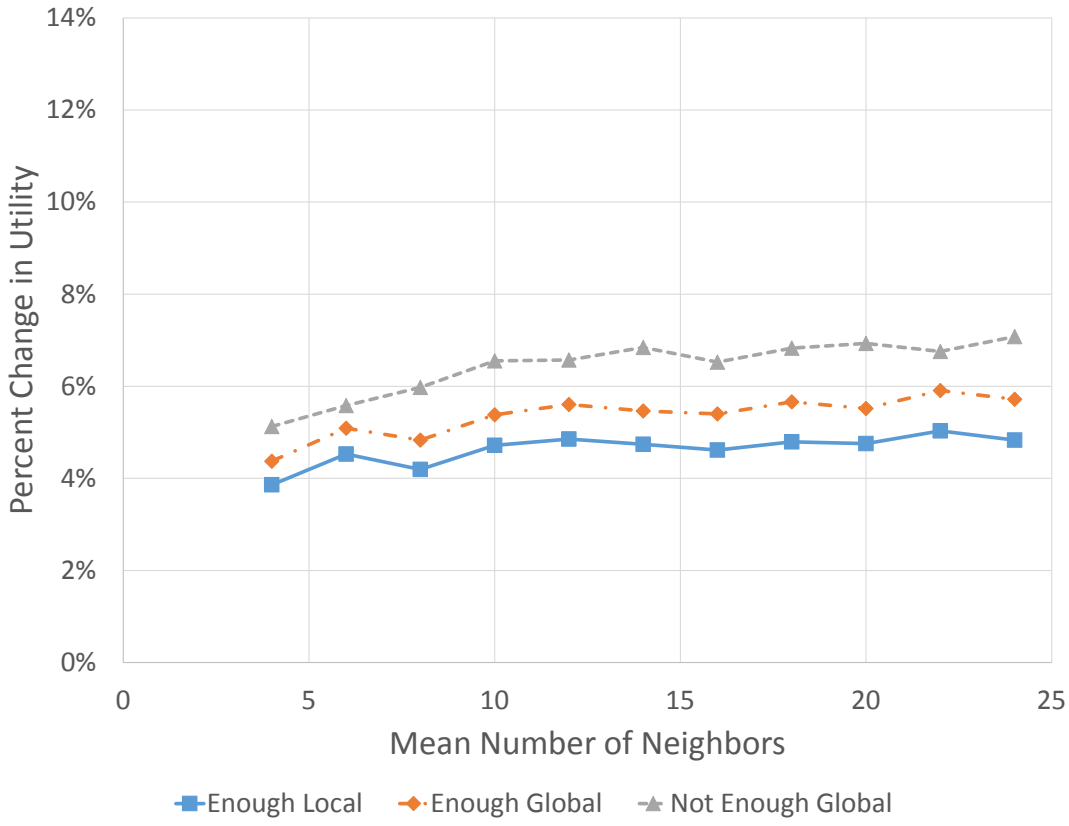

Figure 17: Percent Change in Average Application Utility from Initial Placement (before DistAl) to Final (after DistAl) as a Function of the Number of Neighbors on a Random Resource Graph with Both Applications

Local), but improves utility even more when resources are constrained (Not Enough Global). Figure 16 shows that our algorithm is equally effective at improving utility regardless of the network latency. Figure 17 shows a small increase in utility as the number of neighbors increases. This is mostly due to the increased latency improvement shown in Figure 21, as Figure 19 demonstrates that the change in quality improvement is slight.

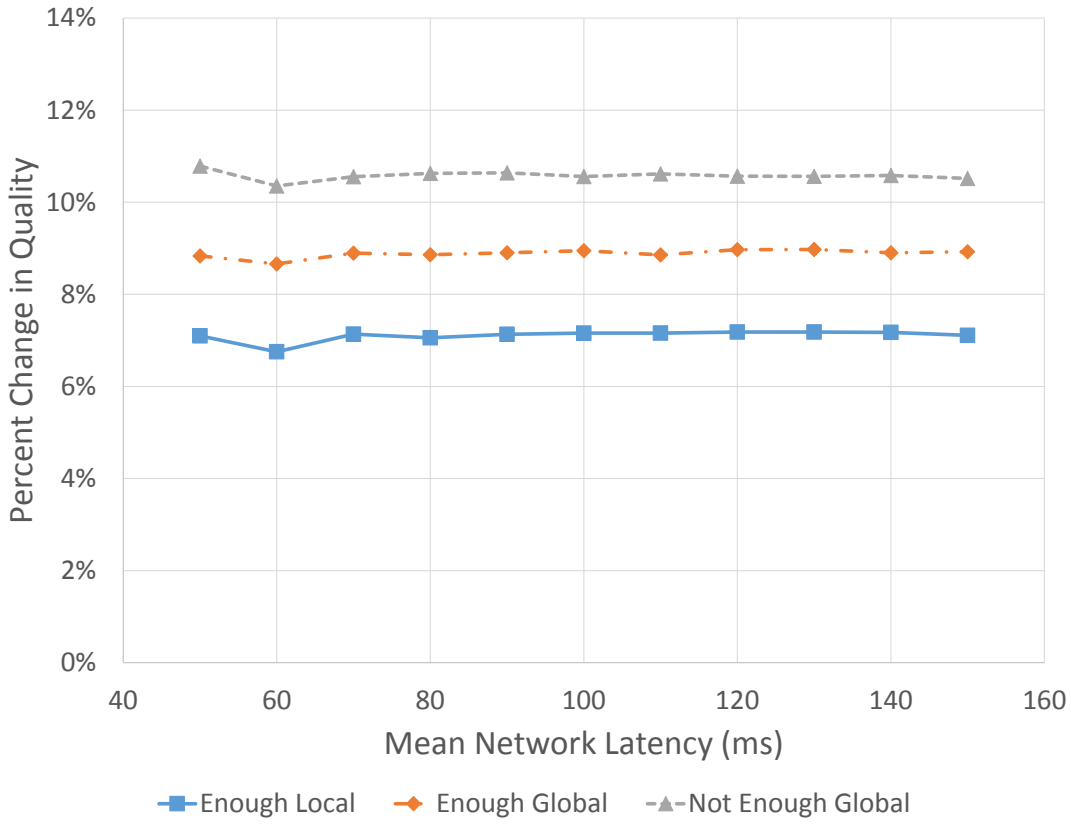

Figure 18: Percent Change in Average Application Quality from Initial Placement (before DistAl) to Final (after DistAl) as a Function of Network Latency on a Random Resource Graph with Both Applications

We considered the improvement in per-application quality by taking the rate as a fraction of maximum rate and fidelity value as a fraction of maximum fidelity value, averaged for each operator in the application. This is measured differently from the way quality is accounted for in the utility function so that this metric is not merely reflecting what we already measured with the change in utility (Figures 16 and 17), but rather is an independent

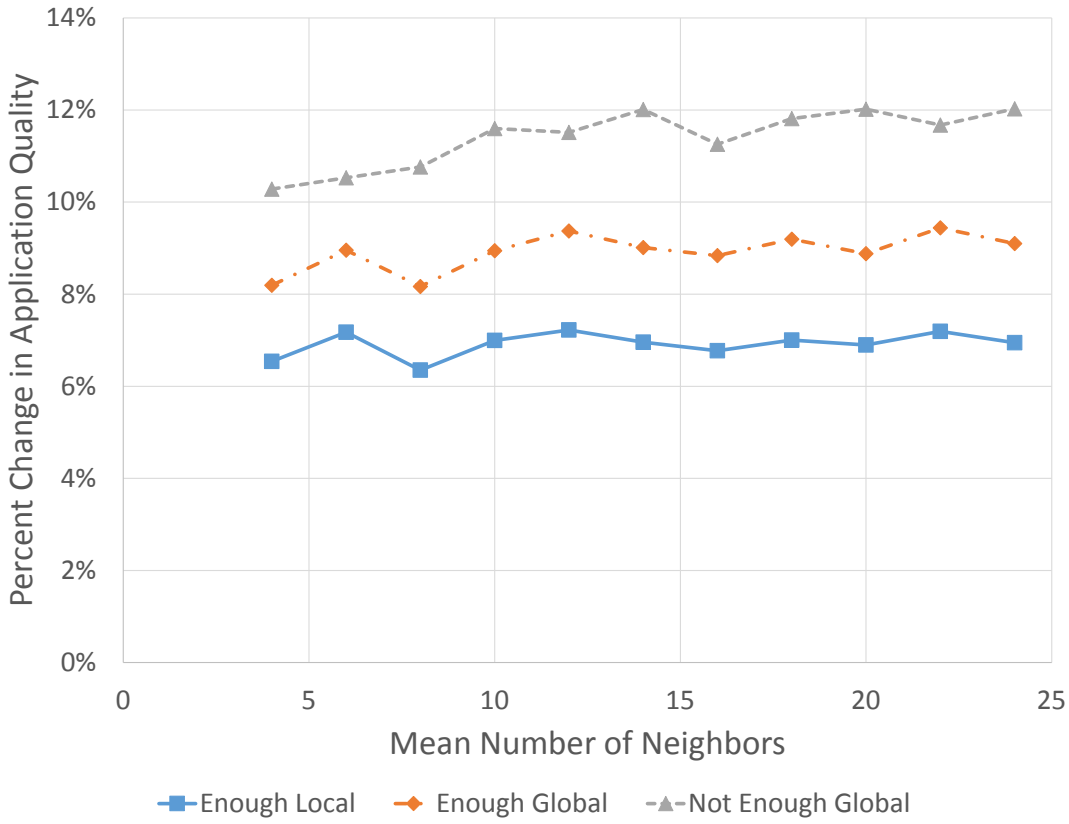

Figure 19: Percent Change in Average Application Quality from Initial Placement (before DistAl) to Final (after DistAl) as a Function of the Number of Neighbors on a Random Resource Graph with Both Applications

measure of quality. Nevertheless, we found that quality improvement very closely correlates with utility improvements, as shown in Figures 18 and 19. Unsurprisingly, network latency has little impact on quality. Figure 19 shows a small increase in quality improvement as resource nodes are given more neighbors, owing to the fact that operators are better able to find more powerful nodes with available resources, but the effect is slight.

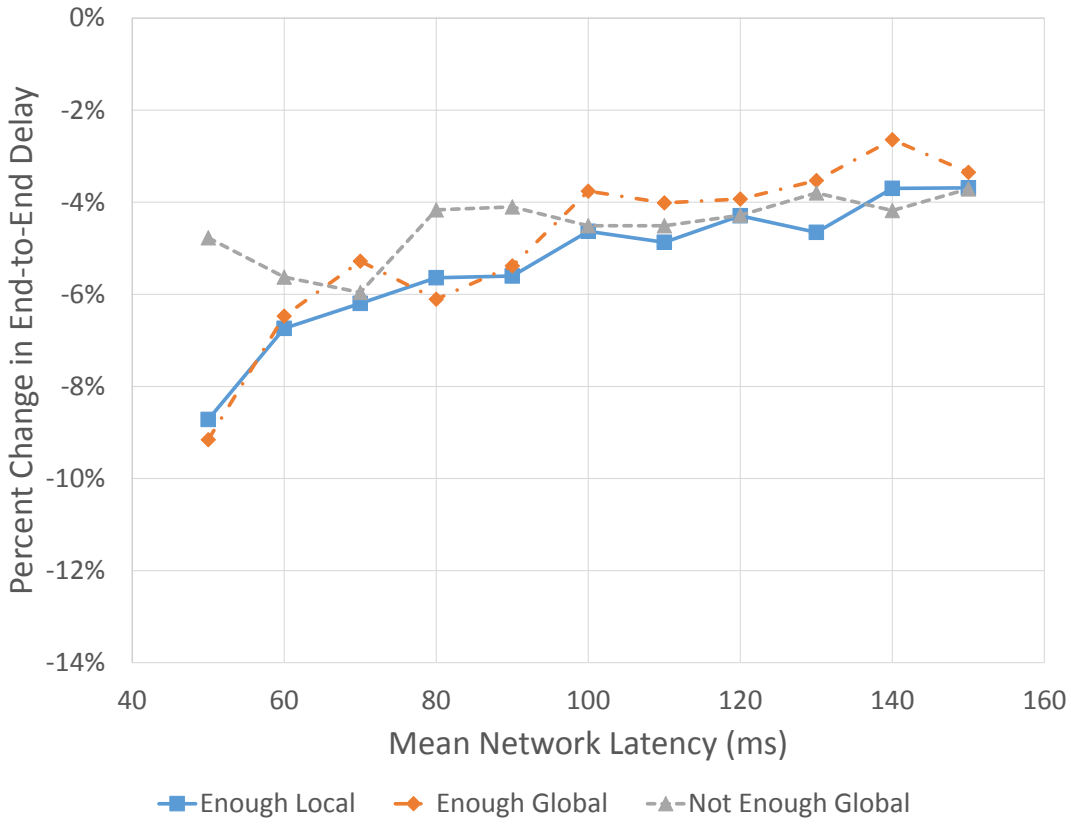

Figure 20: Percent Change in Average Application Quality and End-to-End Delay from Initial Placement (before DistAl) to Final (after DistAl) as a Function of Network Latency on a Random Resource Graph with Both Applications

Figure 20 shows that there is only slightly less relative improvement in average application end-to-end delay as network latency increases. This is because improved or worsened network latency affects both the initial and final placements proportionately, so it is canceled out in any relative comparison. The small reduction still seen is because the initial

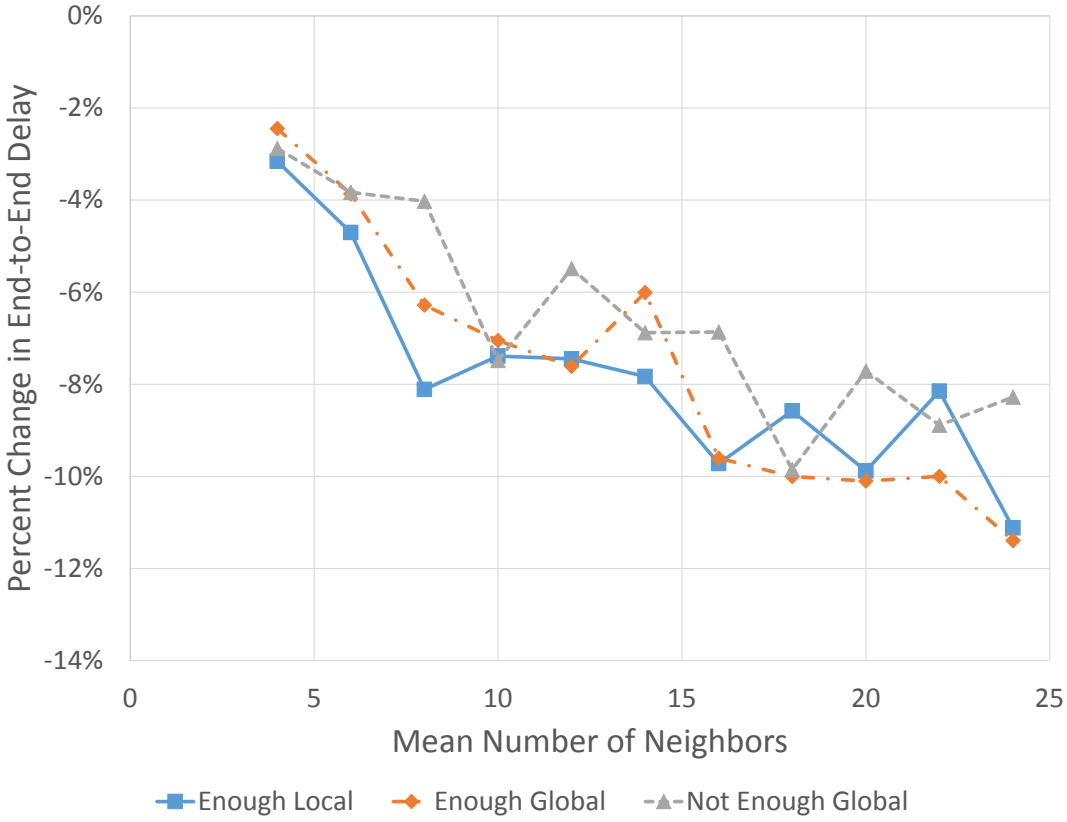

Figure 21: Percent Change in Average Application Quality and End-to-End Delay from Initial Placement (before DistAl) to Final (after DistAl) as a Function of the Number of Neighbors on a Random Resource Graph with Both Applications

placement is closer to optimal as network latency increases. Our method of choosing initial placement tends to bunch operators together, incurring less network latency at the expense of having to reduce quality to run all the operators on relatively few nodes. At low network latencies, this is a bad scheme because operators may be spread out to use more powerful computational resources at relatively low cost, reducing computation latency enough to make up for the added network latency. However, as network latency increases, the penalty for spreading out the operators becomes greater and a schedule similar to the initial placement avoids paying those steep penalties.

However, Figure 21 shows that the total end-to-end delay is affected by the number of neighbors. As discussed earlier, this has the side effect of bringing nodes closer in terms of actual network distance due to the way our simulation is formulated. This has two benefits: First, the source sensor locations and the sink location for the applications may be brought closer together, reducing the total network distance that must fundamentally be traveled. Second, when compute nodes are heterogeneous, it may bring more powerful compute nodes close by, allowing their compute power to be used without incurring as much network penalty. We see diminishing returns because as neighbor connections are added, more distant nodes are linked first becoming close nodes, and next the nodes remaining distant are linked (which were the ones originally only moderately distant).

The cost for improved performance is the additional delay from the initial placement until the application can be started. Therefore we considered how long it takes from the first operator move DistAl makes until the last operator move before convergence completes. Figure 22 shows that while this does vary somewhat with the network latency, the delay stays relatively steady between 1 - 1.5 minutes due to the number of moves needed to converge being fewer in the higher latency configurations, which compensates for the higher delay in sending messages. As discussed earlier, the optimal schedule becomes more similar to the initial placement as network latency increases due to our choice of initial placement algorithm, which tends to bunch operators together, and the increasing costs of placing operators on different nodes as network latency increases.

Figure 23, on the other hand, shows that the startup delay increases as the number

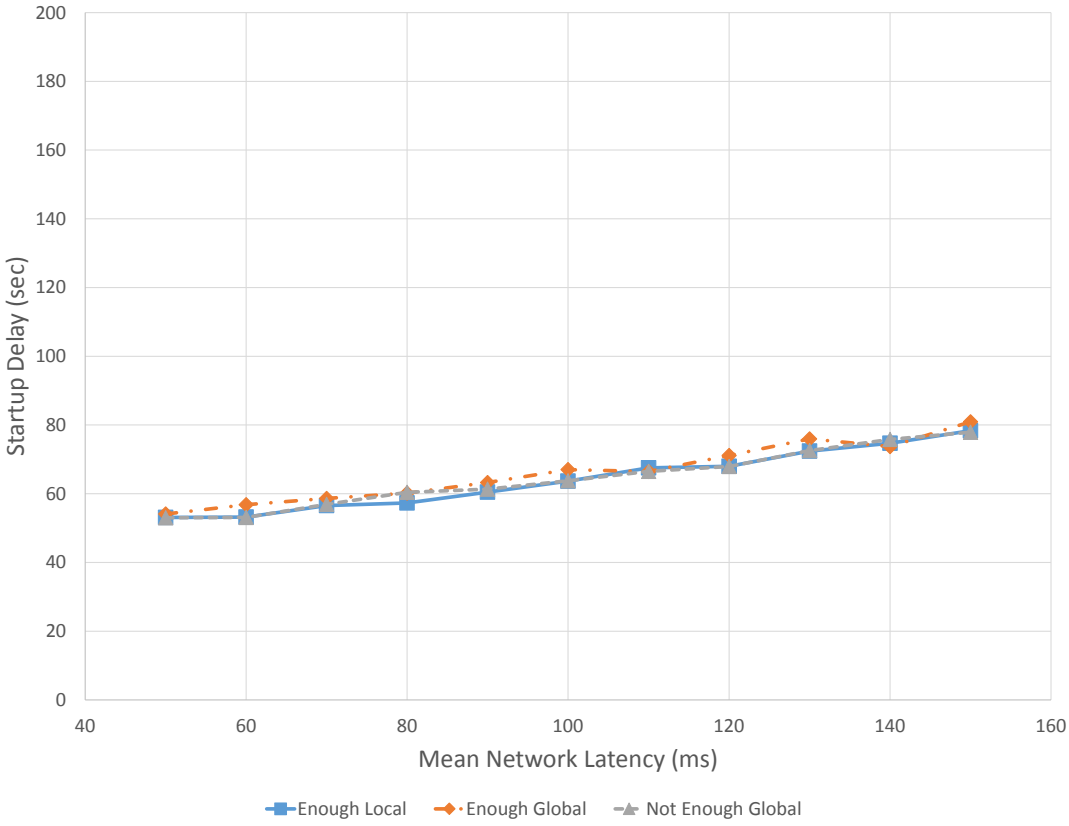

Figure 22: Startup Delay (in sec.) from Initial Placement (before DistAl) to Final Placement (after DistAl) for Both Applications (Started Simultaneously) as a Function of Network Latency on a Random Resource Graph with Both Applications

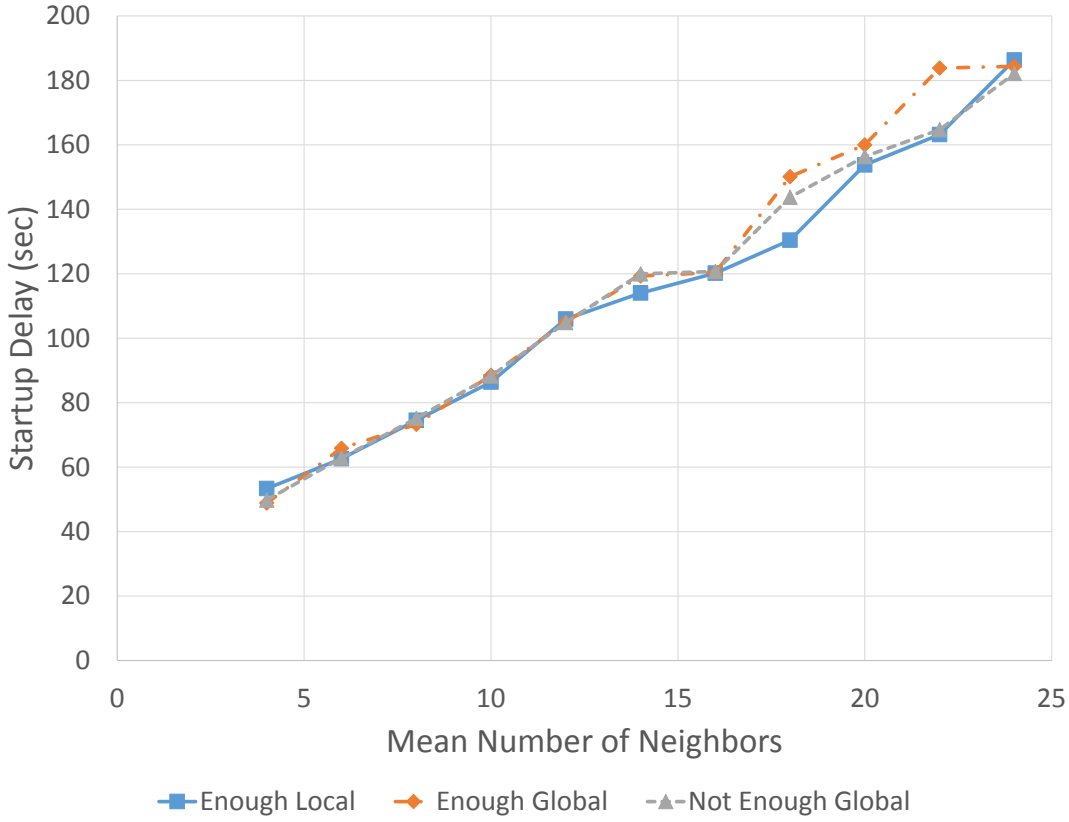

Figure 23: Startup Delay (in sec.) from Initial Placement (before DistAl) to Final Placement (after DistAl) for Both Applications (Started Simultaneously) as a Function of the Number of Neighbors on a Random Resource Graph with Both Applications

of neighbors in the overlay network is increased (although it remains around 2 min. with 16 neighbors and just over 3 min. even with 24). This may seem counterintuitive but is explained by the fact that the total work increases because the amount of work necessary to take a single step is proportionately larger. (Every neighbor is contacted for each considered move, and no decision can be made until the slowest of those neighbors has responded.) This supports our idea that each node have a limited number of neighbors in the resource overlay network, even though the underlying physical network may be fully connected.

We argue that a 1 - 2 minute startup delay is acceptable if it improves the performance and quality of continuous, long-running applications. For example, this startup delay is only a fraction of a percent overhead for an application that runs continuously for just one full day. Furthermore, this is comparable to the real startup delays involved in bringing up new VM instances in some real-world compute clouds. It should also be noted that DistAl can run continuously and make refinements even after the application starts, so it may not always be necessary to wait for the full startup delay before data can be processed, if some sacrifice in the performance and quality can be accepted (at least temporarily) in order to do so. Furthermore, since DistAl uses iterative improvement, each individual step taken increases global system utility, thus an application may be started at any point before convergence without fear that it is a worse schedule, globally, than a previous step.

Once an application is started, DistAl continues to run on all resource nodes and inspect operators for potential improvements. Assuming the distributed algorithm converged before the application was started, this will not result in any further changes to the schedule unless something perturbs the system, such as adding a new application, adding resources to a node, or adding a new node to the resource network. Even when this happens, the actual changes should be small since the new optimal schedule will only be slightly different from the previous one. It should also be noted that the overhead of the continually running algorithm is fully configurable by setting how often a resource node will choose an operator to investigate, though this comes at the cost of slower adaptation to the changing situation.

5.5.3.3 Summary of Results

In summary, our results demonstrate that:

- DistAl is able to improve both application performance and quality of results for multiple, concurrent, complex (hundreds of operators) applications on a large resource network (hundreds of nodes) vs. a reasonable, but naive initial placement.
- By trading off quality, DistAl is able to run applications in resource-constrained environments, where other approaches would have to apply admission control ("Not Enough Global" scenario).
- Startup delay is a reasonable overhead for continuous, long-running applications.
- Limiting the number of neighbors in the resource overlay helps keep startup delay low.

5.5.4 Related Work in Streaming Graph Scheduling

Prior work has investigated mapping streaming analysis applications onto distributed resources in mainly two areas: Grid computing and wireless sensor networks (WSNs).

In Grid computing, Streamline uses a list scheduling hueristic [3], Zhu and Agrawal use graph isomorphism [32], Zhang et al. use a genetic algorithm [36], and Gu and Wu [14] and SWAMP [35] both use layer-oriented dynamic programming (LDP) to maximize frame rate. SWAMP also has an alternative Recursive Critical Path algorithm that minimizes end-to-end delay. All of these have a few things in common. Firstly they are centralized algorithms that use global knowledge about the application and resource network states. This is acceptable in the Grid environment because they are only scheduling operators onto the resources that have already been allocated to the application, and therefore scale is limited. However, these algorithms are not involved in the resource allocation step, and therefore can help efficiently assign operators to particular allocated nodes but cannot ensure that the nodes have been allocated in a manner efficient for streaming applications. Furthermore, they are designed with scientific computing applications in mind and thus optimize for throughput (e.g., maximum frame rate). Only a few, such as SWAMP [35],

allow an option to optimize for minimum end-to-end delay. However, in many modern live analysis applications, end-to-end application delay is the key performance metric, and throughput is simply a constraint to be met minimally.

Role assignment in wireless sensor networks (WSNs) is similar to mapping an operator graph onto distributed resources. DFuse [21] uses a fully distributed, iterative algorithm to assign fusion nodes to sensors by making local decisions with partial knowledge. Frank and Römer [36] use a similar algorithm, but their system is applicable to a wider variety of role assignment problems. Manoj et al. [23] uses a greedy A* algorithm to search the space of role assignments, but unlike the others, this requires global knowledge and does not run continuously to adapt to the dynamic environment. These algorithms tend to put the focus on optimizing for energy, except Manoj et al. who emphasize reliability, though they still consider energy as a factor since this is a primary source of sensor failure. However, our problem does not consider energy since we are focused on infrastructure resources, and instead we focus on performance and quality of results. Furthermore, in a WSN application, a sensor node can only have one role, while a node in our resource model can run many operators at once.

Some key differences both the WSN and Grid work have from DistAl are 1) that they only consider a single application running on the resources they schedule, and do not explicitly address multiple applications, 2) they either assume adequate resources exist for the application or apply strict admission control, and 3) only consider optimizing performance, and do not consider trading off quality of results against performance within resource constraints. DFuse is also the only related work that uses a cost function, much like our utility function, to customize how an application should be optimized.

Finally, SBON [25] is a system to place operators for distributed stream-processing systems. It uses a fully distributed algorithm based on spring relaxation on the operators in a cost space. This system does explicitly support multiple running applications, but still assumes adequate resources exist and does not consider adjustable quality levels. How different factors being optimized are considered relative to each other are baked into the cost space and not customizable with a utility function as in our system.

5.5.5 Discussion

We have presented the problem of scheduling live streaming analysis on geographically distributed resources as well as DistAl, our method and algorithm for such scheduling. We have also argued for the necessity of such scheduling for applications such as situation-awareness, cyberphysical systems, complex event processing, and the Internet of Things. DistAl is unique in its ability to provide all of the following features that are desirable in such a scenario:

- Fully distributed algorithm that requires only partial system information to make local decisions
- Continuously adapts to a dynamic environment
- Creates an operator placement schedule that meets resource constraints
- Avoids admission control by trading off quality of results to meet resource constraints
- Uses application-specific criteria (in the form of a utility function) to optimize performance and quality according to each application's needs

Our implementation uses network latency and computational power to demonstrate the concepts of resource constraints and accounting for performance in utility functions. However, other resource and performance measures may be desirable. Bandwidth can be treated similarly to CPU, both as a constraint (total communication cannot exceed a node's bandwidth) and a term in the utility function (output size vs. transmission rate) to account for the additional delay. This would also require adding a per-item output size to the fidelity table for each operator. Memory can simply be treated as a constraint, comparing operators' memory requirements to the available memory on each node. Since our system is already designed to handle both resource constraints and customizable utility functions, it serves as a proof of concept and demonstrates that additional resources such as these can easily be added.

CHAPTER VI

CONCLUSION

We have presented a set of requirements and a system design in order to address the question, "What should a system that supports Analysis of Things applications look like?" To that end, we have designed, implemented, and quantitatively evaluated a system to support AoT in the local (single-site) scenario. We have also presented detailed requirements for federating the Stream Registry and Resource Manager. Based on these, we have designed the federated $_{SS}$IoTa system and presented simulation results for DistAl, the scheduling algorithm for the federated Resource Manager.

In order to create and execute an application using $_{SS}$IoTa, developers must provide the following:

- an operator graph that specifies the application's operators, input sensors, and the connectivity between them (such as the example in Appendix A)
- operator code for each operator in the operator graph
- a fidelity table for each operator
- a utility function for the application

Since the Operator Store enables component-based design, a single developer need not provide all of these elements. Specifically, operator developers can provide operator code and corresponding fidelity tables through the Operator Store, allowing the application developer to focus on the operator graph and utility function. Furthermore, while application-specific utility functions allow a custom utility function to be created for each application, we anticipate that developers will in practice settle on a few good utility functions with tunable parameters. (An example of tunable parameters is the term weights in the example utility function presented in Equation 1.) This leaves the application developer's task

to simply choosing one of the few common utility functions and adjusting the parameters appropriately.

Furthermore, the Stream Registry helps application developers to discover sensors to use in their applications and alleviates their need to manage the sensing infrastructure. Meanwhile, the execution environment (Resource Manager and Worker Nodes) alleviates the developer's need to determine one which resources to execute all the components of their application, to monitor those resources, and to manage the running application.

Therefore we conclude from our construction that: Systems support for Live Streaming Analysis in the Internet of Things can reduce the complexity of developing and executing such applications on computational resources and using end devices that are widely distributed at the edge of the network.

6.1 Future Work

This area presents many opportunities for future research. We conclude by presenting some of the most interesting opportunities, though this is by no means an exhaustive list.

While the Stream Registry helps to bring sensors into the SSIoTa ecosystem, there remains a gap between the Stream Registry and the sensors themselves that must be closed by registering the available sensors. One significant improvement would be a plug-and-play protocol that, when implemented by sensors, would allow them to automatically register themselves with the Stream Registry. The protocol should also include a pointer to where the system can retrieve any driver operators (as presented in Section 4.1.6.1) and automatically add them to the Operator Store. The goal is to allow sensors to be immediately usable by SSIoTa applications immediately upon their being connected to the network and with zero configuration required.

Another avenue for research is allowing running applications to dynamically adapt to changes in the available sensors (i.e., if sensors are added or removed). We propose three abstractions to be used in combination to achieve this. First, *stream groups* represent a collection of streams that contain the same type of data (e.g., all video), which allows the system to reason about them as a single unit. A stream group could represent streams

that all come from sensors of the same type, or a set of streams representing intermediate data after some parallel computation has been performed. An application can create a stream group by specifying a query on sensors, rather than naming a specific sensor, in the application description. The sensor streams from all sensors matching the query constitute a stream group. Second, *parallel operators* allow applications to specify an operation to be performed on a stream group. For example, a parallel map operator could specify an operator to be performed in parallel on all streams in a stream group - one instance of the operator would be created for each stream in the stream group, and the output streams of these operator instances would form a new stream group. Other computation patterns also exist, such as reduction / fusion parallel operators. Finally, the Stream Registry should allow continuous queries against its set of registered sensors. This would allow stream groups specified in the application description to be dynamically updated with new or removed streams and sensors that are added to or removed from the network. As the system is notified of changes to the continuous sensor query, it must update the stream group membership, then dynamically adapt any parallel operators using the stream group as input, then update any stream groups those parallel operators produce as output, and so forth through the running application.

As we have not yet specifically addressed fault tolerance, this is also an important area for future research. Two main types of faults need to be addressed: node failure and sensor failure. In the case of a Worker Node failure, any operators on that node would stop running, thus halting all applications to which they belong. Such failures must be detected and the operators restarted on good computational resources as quickly as possible, to minimize disruption to the applications. Furthermore, this should be done without blatant replication of all computation, since that dramatically increases the amount of resources consumed without accomplishing any additional work. Sensor failures should also be detected and the Stream Registry updated accordingly. With the aforementioned continuous query concept, applications could automatically adapt to sensor failures so long as the Stream Registry is promptly updated with the new situation.

Several enhancements to the execution engine would also be beneficial. The first is

support for heterogeneous resources on the Worker Nodes, such as GPUs and hardware accelerators. This requires giving the system an awareness of the resources available on each Worker, as well as providing a mechanism in the programming model for resource needs of each operator to be specified. Some operators may require certain resources. Others may consider them optional, providing performance benefits when they are available. The system must then be able to efficiently schedule operators with different needs on the heterogeneous Workers. The second improvement is the ability to dynamically monitor operator performance and system resources, which could alleviate the developer's need to precisely and accurately provide performance characteristics to the system. It would also help to support operators whose performance may be data dependent. Finally, the programming model and execution engine could be enhanced to support applications that dynamically spawn new computation at runtime. This could come up in applications that have situation-dependent branches in their operator graph, such as a suspect tracking applications that spawns tracking computation each time a suspect is detected.

Another area for future work is providing additional extensible options in two areas. SSIoTa automatically synchronizes input streams on behalf of operators, before calling their handler function. However, there are a number of different ways that synchronization could be performed. Therefore we propose making the synchronization mechanism an extensible module, which would allow easy addition of new methods for synchronizing streams. Also, operators are currently responsible for serializing and deserializing their streaming inputs and outputs. This puts an extra burden on operator developers, and also results in unnecessary duplication of effort since streams of the same type may be used by many operators. Therefore, the second opportunity for extensibility is to allow modules that marshal streams of different types. These modules could be registered with the type of stream they handle, and automatically called by the system when streams of that type are used by operators.

We believe that system support for the Analysis of Things is a fertile area for further research. It is our hope that these ideas and other future work will prove to practically benefit AoT applications.

APPENDIX A

EXAMPLE OPERATOR GRAPH FILE

```
> TCPSource({MYIP}:30000) > jpeg-video-0
jpeg-video-0 > JpegDecode > video-0
video-0 > DetectForeground > foreground-0
video-0 > SplitFrame > video-0-0, video-0-1, video-0-2, video-0-3
video-0-0 > DetectForeground > foreground-0-0
video-0-1 > DetectForeground > foreground-0-1
video-0-2 > DetectForeground > foreground-0-2
video-0-3 > DetectForeground > foreground-0-3
foreground-0-0,foreground-0-1,foreground-0-2,foreground-0-3 > JoinFrame > foreground-0
video-0, foreground-0 > Track > tracked-0
tracked-0 > JpegEncode > jpeg-tracked-0
jpeg-tracked-0 > TCPSink({MYIP}:30100) > sink-0
> TCPSource({MYIP}:30001) > jpeg-video-1
jpeg-video-1 > JpegDecode > video-1
video-1 > DetectForeground > foreground-1
video-1 > SplitFrame > video-1-0, video-1-1, video-1-2, video-1-3
video-1-0 > DetectForeground > foreground-1-0
video-1-1 > DetectForeground > foreground-1-1
video-1-2 > DetectForeground > foreground-1-2
video-1-3 > DetectForeground > foreground-1-3
foreground-1-0,foreground-1-1,foreground-1-2,foreground-1-3 > JoinFrame > foreground-1
video-1, foreground-1 > Track > tracked-1
tracked-1 > JpegEncode > jpeg-tracked-1
jpeg-tracked-1 > TCPSink({MYIP}:30101) > sink-1
```

APPENDIX B

DISTAL PSEUDOCODE

Algorithm 2 Message Handlers

```
1: function HANDLE_MOVE_QUERY(Node host, Node sender, Operator op)
2:     utilityDelta ← REDUCE_QUALITY_FOR_CONSTRAINTS(host, op)
3:     runOps ← RunningOperators(host)
       RollbackQualityChanges(runOps ∪ {op})
       send(sender,QueryResponse(utilityDelta))
4: end function
5: function HANDLE_INPUT_QUERY(Node host, Node sender, Operator inputOp, Node
   inputNewHost)
6:     utilityDelta ← 0
7:     movedOp ← inputOp
8:     Host(movedOp) ← inputNewHost
9:     for all op ∈ RunningOperators(host) ∩ Outputs(inputOp) do
10:        tempOp ← op
11:        Inputs(tempOp) ← Inputs(tempOp) −{inputOp} ∪ {movedOp}
12:        utilityDelta ← utilityDelta+ UtilityFunction(tempOp)
13:                     − UtilityFunction(op)
14:    end for
       send(sender,QueryResponse(utilityDelta))
15: end function
16: function HANDLE_MOVE_OPERATOR(Node host, Node sender, Operator op)
       REDUCE_QUALITY_FOR_CONSTRAINTS(host, op)
17:    runOps ← RunningOperators(host)
       CommitQualityChanges(runOps ∪ {op})
18:    RunningOperators(host) ← runOps ∪ {op}
       send(sender,MoveAck())
19: end function
```

Algorithm 3 Greedy Placement

```
1: function MAIN(Node host)
2:     while true do
3:         op ← CHOOSEOPERATOR(host)
           ATTEMPT_MOVE(host, op)
4:     end while
5: end function
6: function ATTEMPT_MOVE(Node host, Operator op)
7:     RUNNINGOPERATORS(host) ← RUNNINGOPERATORS(host) −{op}
8:     myUtilityDelta ← INCREASE_QUALITY_UNTIL_CONSTRAINTS(host)
9:                          − UTILITYFUNCTION(op)
10:    bestUtilityDelta ← −∞
11:    bestNewHost ← ∅
12:    for all neighbor ∈ NEIGHBORS(host) do
           SEND(neighbor,MoveQuery(op))
13:        queriesMade ← 1
14:        for all outputHost ∈ OUTPUTS(op) do
           SEND(outputHost,InputQuery(op, neighbor))
15:            queriesMade ← queriesMade + 1
16:        end for
17:        neighborUtilityDelta ← 0
18:        while queriesMade > 0 do
           RECEIVE(QueryResponse, sender)
19:            neighborUtilityDelta ← neighborUtilityDelta
20:                                   + UTILITYDELTA(QueryResponse)
21:            queriesMade ← queriesMade − 1
22:        end while
23:        if neighborUtilityDelta > bestUtilityDelta then
24:            bestUtilityDelta ← neighborUtilityDelta
25:            bestNewHost ← neighbor
26:        end if
27:    end for
28:    if bestUtilityDelta + myUtilityDelta > 0 then
           SEND(bestNewhost,MoveOperator(op))
           RECEIVE(MoveAck, sender)
29:        runOps ← RUNNINGOPERATORS(host)
           COMMITQUALITYCHANGES(runOps)
30:    else
31:        runOps ← RUNNINGOPERATORS(host)
           ROLLBACKQUALITYCHANGES(runOps)
32:        RUNNINGOPERATORS(host) ← runOps ∪ {op}
33:    end if
34: end function
```

Algorithm 4 Greedy Quality Adjustment pt. 1

```
1: function REDUCE_QUALITY_FOR_CONSTRAINTS(Node host, Operator newOp)
2:     FIDELITY(newOp) ← MAXFIDELITY(newOp)
3:     RATE(newOp) ← MAXRATE(newOp)
4:     utilityDelta ←UTILITYFUNCTION(newOp)
5:     while CONSTRAINTSEXCEEDED(host) do
6:         improvement ← 0
7:         changeUtility ← 0
8:         changeOp ← ∅
9:         modifiedOp ← ∅
10:        for all op ∈ RUNNINGOPERATORS(host) ∪{newOp} do
11:            if FIDELITY(op) > MINFIDELITY(op) then
12:                tempOp ←REDUCEFIDELITYONESTEP(op)
13:                tempUtility ←UTILITYFUNCTION(tempOp)−UTILITYFUNCTION(op)
14:                tempConstraint ←CONSTRAINEDUSAGE(tempOp)
15:                                − CONSTRAINEDUSAGE(op)
16:                if IMPROVEMENTFN(tempConstraint, tempUtility)
17:                    > improvement then
18:                    improvement ← IMPROVEMENTFN(tempConstraint, tempUtility)
19:                    changeUtility ← tempUtility
20:                    changeOp ← op
21:                    modifiedOp ← tempOp
22:                end if
23:            end if
24:            if RATE(op) > MINRATE(op) then
25:                tempOp ←REDUCERATEONESTEP(op)
26:                tempUtility ←UTILITYFUNCTION(tempOp)−UTILITYFUNCTION(op)
27:                tempConstraint ←CONSTRAINEDUSAGE(tempOp)−CONSTRAINEDUSAGE(op)
28:                if IMPROVEMENTFN(tempConstraint, tempUtility)
29:                    > improvement then
30:                    improvement ← IMPROVEMENTFN(tempConstraint, tempUtility)
31:                    changeUtility ← tempUtility
32:                    changeOp ← op
33:                    modifiedOp ← tempOp
34:                end if
35:            end if
36:        end for
37:        if changeOp ≠ ∅ then
38:            changeOp ← modifiedOp
39:            utilityDelta ← utilityDelta + changeUtility
40:        end if
41:    end while
       return utilityDelta
42: end function
```

Algorithm 5 Greedy Quality Adjustment pt. 2

```
1: function INCREASE_QUALITY_UNTIL_CONSTRAINTS(Node host)
2:     utilityDelta ← 0
3:     madeChange ← true
4:     while madeChange do
5:         improvement ← 0
6:         changeUtility ← 0
7:         changeOp ← ∅
8:         modifiedOp ← ∅
9:         for all op ∈ RunningOperators(host) do
10:            if Fidelity(op) < MaxFidelity(op) then
11:                tempOp ←IncreaseFidelityOneStep(op)
12:                tempUtility ←UtilityFunction(tempOp)−UtilityFunction(op)
13:                tempConstraint ←ConstrainedUsage(tempOp)−ConstrainedUsage(op)
14:                if ImprovementFn(tempConstraint, tempUtility) > improvement
15:                    ∧ WouldNotExceedConstraints(tempConstraint) then
16:                    improvement ← ImprovementFn(tempConstraint, tempUtility)
17:                    changeUtility ← tempUtility
18:                    changeOp ← op
19:                    modifiedOp ← tempOp
20:                end if
21:            end if
22:            if Rate(op) < MaxRate(op) then
23:                tempOp ←IncreaseeRateOneStep(op)
24:                tempUtility ←UtilityFunction(tempOp)−UtilityFunction(op)
25:                tempConstraint ←ConstrainedUsage(tempOp)−ConstrainedUsage(op)
26:                if ImprovementFn(tempConstraint, tempUtility) > improvement
27:                    ∧ WouldNotExceedConstraints(tempConstraint) then
28:                    improvement ← ImprovementFn(tempConstraint, tempUtility)
29:                    changeUtility ← tempUtility
30:                    changeOp ← op
31:                    modifiedOp ← tempOp
32:                end if
33:            end if
34:        end for
35:        if changeOp ≠ ∅ then
36:            changeOp ← modifiedOp
37:            utilityDelta ← utilityDelta + changeUtility
38:            madeChange ← true
39:        else
40:            madeChange ← false
41:        end if
42:    end while
       return utilityDelta
43: end function
```

REFERENCES

[1] "Amazon Elastic Compute Cloud." `http://aws.amazon.com/ec2/`.

[2] "Storm, distributed and fault-tolerant realtime computation." `http://storm-project.net/`.

[3] Agarwalla, B., Ahmed, N., Hilley, D., and Ramachandran, U., "Streamline: scheduling streaming applications in a wide area environment," *Multimedia Systems*, vol. 13, pp. 69–85, September 2007.

[4] Akidau, T., Balikov, A., Bekiroglu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S., "MillWheel: Fault-tolerant stream processing at internet scale," in *Proceedings of the 39th International Conference on Very Large Data Bases*, VLDB '13, pp. 734–746, 2013.

[5] Amini, L., Andrade, H., Bhagwan, R., Eskesen, F., King, R., Selo, P., Park, Y., and Venkatramani, C., "SPC: A distributed, scalable platform for data mining," in *Data Mining Standards, Services and Platforms*, DMSSP '06, 2006.

[6] Ashton, K., "That 'Internet of Things' thing," *RFID Journal*, June 2009.

[7] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S., "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pp. 13–16, 2012.

[8] Dean, J. and Ghemawat, S., "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, OSDI '04, 2004.

[9] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W., "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pp. 205–220, 2007.

[10] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M., "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, pp. 114–131, June 2003.

[11] Evans, D., "The Internet of Things: How the next evolution of the Internet is changing everything," white paper, Cisco Systems, Inc., April 2011.

[12] Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S., and Doo, M., "SPADE: The System S declarative stream processing engine," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pp. 1123–1134, 2008.

[13] Ghemawat, S., Gobioff, H., and Leung, S.-T., "The Google File System," *SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.

[14] Gu, Y. and Wu, Q., "Maximizing workflow throughput for streaming applications in distributed environments," in *International Conference on Computer Communications and Networks*, ICCCN '10, pp. 1–6, Aug 2010.

[15] Guttman, A., "R-trees: a dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pp. 47–57, 1984.

[16] Harvey, N. J. A., Jones, M. B., Saroiu, S., Theimer, M., and Wolman, A., "Skipnet: A scalable overlay network with practical locality properties," in *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, USITS '03, 2003.

[17] Hilley, D. and Ramachandran, U., "Stampede^{RT}: Programming abstractions for live streaming applications," in *Proceedings of the 27th International Conference on Distributed Computing Systems*, ICDCS '07, p. 65, 2007.

[18] Hilley, D. and Ramachandran, U., "Persistent Temporal Streams," in *Proceedings of the 10th International Middleware Conference*, Middleware '09, December 2009.

[19] Hong, K., Smaldone, S., Shin, J., Lillethun, D. J., Iftode, L., and Ramachandran, U., "Target container: A target-centric parallel programming abstraction for video-based surveillance," in *International Conference on Distributed Smart Cameras*, ICDSC '11, pp. 1–8, 2011.

[20] Israel, S., "How Walmart and Heineken will use Shopperception to put your in-store experience in context." `http://www.forbes.com/sites/shelisrael/2013/01/27/how-walmart-and-heineken-will-use-shoppercetion-to-put-your-in-store-experience-in-context/`, January 2013.

[21] Kumar, R., Wolenetz, M., Agarwalla, B., Shin, J., Hutto, P., Paul, A., and Ramachandran, U., "Dfuse: A framework for distributed data fusion," in *International Conference on Embedded Networked Sensor Systems*, SenSys '03, pp. 114–125, 2003.

[22] Lillethun, D. J., Hilley, D., Horrigan, S., and Ramachandran, U., "MB++: An integrated architecture for pervasive computing and high-performance computing," in *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '07, pp. 241–248, August 2007.

[23] Manoj, B., Sekhar, A., and Murthy, C. S. R., "A state-space search approach for optimizing reliability and cost of execution in distributed sensor networks," *Journal of Parallel and Distributed Computing*, vol. 69, no. 1, pp. 12–19, 2009.

[24] Neumeyer, L., Robbins, B., Nair, A., and Kesari, A., "S4: Distributed stream computing platform," in *Proceedings of the IEEE International Conference on Data Mining Workshops*, ICDMW'10, pp. 170 –177, December 2010.

[25] Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., and Seltzer, M., "Network-aware operator placement for stream-processing systems," in *International Conference on Data Engineering*, ICDE '06, pp. 49–49, April 2006.

[26] Ramabhadran, S., Ratnasamy, S., Hellerstein, J. M., and Shenker, S., "Brief announcement: Prefix hash tree," in *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, (New York, NY, USA), pp. 368–368, ACM, 2004.

[27] Ramachandran, U., Modahl, M., Bagrak, I., Wolenetz, M., Lillethun, D. J., Liu, B., Kim, J., Hutto, P., and Jain, R., "MediaBroker: A pervasive computing infrastructure for adaptive transformation and sharing of stream data," *Pervasive and Mobile Computing*, vol. 1, July 2005.

[28] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S., "A scalable content-addressable network," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, (New York, NY, USA), pp. 161–172, ACM, 2001.

[29] Rowstron, A. I. T. and Druschel, P., "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, (London, UK, UK), pp. 329–350, Springer-Verlag, 2001.

[30] Samet, H., "The quadtree and related hierarchical data structures," *ACM Computing Surveys*, vol. 16, pp. 187–260, June 1984.

[31] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H., "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, (New York, NY, USA), pp. 149–160, ACM, 2001.

[32] Tang, A., Liu, Z., Xia, C., and Zhang, L., "Distributed resource allocation for stream data processing," in *High Performance Computing and Communications*, vol. 4208 of *Lecture Notes in Computer Science*, pp. 91–100, 2006.

[33] Varga, A. and Hornig, R., "An overview of the OMNeT++ simulation environment," in *Simulation tools and techniques for communications, networks and systems & workshops*, Simutools '08, 2008.

[34] Wolenetz, M., Kumar, R., Shin, J., and Ramachandran, U., "Middleware guidelines for future sensor networks," in *Workshop on Broadband Advanced Sensor Networks*, BASENETS '04, October 2004.

[35] Wu, Q., Zhu, M., Gu, Y., Brown, P., Lu, X., Lin, W., and Liu, Y., "A distributed workflow management system with case study of real-life scientific applications on grids," *Journal of Grid Computing*, vol. 10, no. 3, pp. 367–393, 2012.

[36] Zhang, W., Cao, J., Zhong, Y., Liu, L., and Wu, C., "Grid resource management and scheduling for data streaming applications," *Computing and Informatics*, vol. 29, September 2010.