2017

Contributions to Edge Computing

Vernon K. Bumgardner
University of Kentucky, cody@uky.edu
Author ORCID Identifier:
http://orcid.org/0000-0001-9588-3501
Digital Object Identifier: https://doi.org/10.13023/ETD.2017.086

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Vernon K. Bumgardner, Student

Dr. Victor W. Marek, Major Professor

Dr. Miroslaw Truszczynski, Director of Graduate Studies

CONTRIBUTIONS TO EDGE COMPUTING

DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy in the
College of Engineering
at the University of Kentucky

By

V. K . Cody Bumgardner

Lexington, Kentucky

Director: Dr. Victor W. Marek, Professor of Computer Science

Lexington, Kentucky

ABSTRACT OF DISSERTATION

CONTRIBUTIONS TO EDGE COMPUTING

Efforts related to Internet of Things (IoT), Cyber-Physical Systems (CPS), Machine to Machine (M2M) technologies, Industrial Internet, and Smart Cities aim to improve society through the coordination of distributed devices and analysis of resulting data. By the year 2020 there will be an estimated 50 billion network connected devices globally and 43 trillion gigabytes of electronic data. Current practices of moving data directly from end-devices to remote and potentially distant cloud computing services will not be sufficient to manage future device and data growth.

Edge Computing is the migration of computational functionality to sources of data generation. The importance of edge computing increases with the size and complexity of devices and resulting data. In addition, the coordination of global edge-to-edge communications, shared resources, high-level application scheduling, monitoring, measurement, and Quality of Service (QoS) enforcement will be critical to address the rapid growth of connected devices and associated data.

We present a new distributed agent-based framework designed to address the challenges of edge computing. This actor-model framework implementation is designed to manage large numbers of geographically distributed services, comprised from heterogeneous resources and communication protocols, in support of low-latency real-time streaming applications. As part of this framework, an application description language was developed and implemented. Using the application description language a number of high-order management modules were implemented including solutions for resource and workload comparison, performance observation, scheduling, and provisioning. A number of hypothetical and real-world use cases are described to support the framework implementation.

KEYWORDS: Edge Computing, Distributed Systems, Cloud Computing

V. K . CODY BUMGARDNER
Student's Signature

MARCH 3, 2017
Date

CONTRIBUTIONS TO EDGE COMPUTING

By

V. K . Cody Bumgardner

VICTOR W. MAREK
Director of Dissertation

MIROSLAW TRUSZCZYNSKI
Director of Graduate Studies

MARCH 3, 2017
Date

To Sarah, Sydney, and Jackson

ACKNOWLEDGEMENTS

I am very thankful to have worked with a number of wonderful people over my professional and academic career. Having returned to academic pursuits after working a number of years in industry, I gained a deep appreciation of learning and the academic process. I have been fortunate to work at the University of Kentucky for over 15 years, much of that time reporting to Dr. Doyle Friskney. Without the support of Doyle and other colleagues I might have never completed my undergraduate degree, much less a Ph.D.

A great debt is owed to my advisor, Dr. Victor Marek, who gave me the gift of his time, experience, and patience. Over the last seven years of training with Dr. Marek, much was given, much was expected, and lessons far more important than academic instruction were learned. I am grateful to the members of my Doctoral Advisory Committee, Drs. Marek, Calvert, Dietz, Truszczynski, each member provided support, guidance, and instruction in their own way.

I thank my co-authors and collaborators, especially Caylin Hickey, who contributed to a number of papers and Cresco-based projects over the past several years. Caylin had the insight to pull common functions from standalone modules and package them into shared libraries, greatly contributing to the quality and consistency of the framework.

Finally, I must thank Sarah my wife, Sydney my daughter, and Jackson my son. I started graduate school before the children were born, a deficit of time is owed and will be repaid with interest. Through the stress of publication deadlines, back-to-back travel, and a demanding full-time job, I never had to worry about my family. Sarah took care of everything, including me, putting herself and her career second. We are who we are together, the effort, time, failure, and success we share.

Table of Contents

List of Tables

List of Figures

1

Edge Computing Introduction

"The value of a network increases exponentially with the number of nodes."

– Robert M. Metcalfe

For much of the history of computing data-generating resources have been consolidated in institutional data centers. End-users connect over communication networks to access central data and computational resources. With the emergence of so-called "Cloud Computing" many computational resources have moved out of institutional data centers, but the practice of direct end-user access to central data centers persists. However, with the prolific rise of smart devices such as phones, sensors, and other distributed instrumentation a great deal of data generation moved from data centers to the devices. In China alone there were a reported 9 billion devices as of 2014, with estimates of 24 billion by 2020 [1]. By the year 2020 there will be an estimated 50 billion network connected devices globally [2, 3] and 43 trillion gigabytes of electronic data [4].

Smart devices are often part of larger complex systems where actionable events are determined through the correlation of data generated from distributed devices. In 1991, Weiser described [5] a highly-connected world where devices would "weave themselves into the fabric of everyday life." By the mid-2000s Weiser's vision of

ubiquitous computing became commonly known as the Internet of Things (IoT). Whether referred[1] to as IoT, Cyber-Physical Systems (CPS) [6], Machine to Machine (M2M) [7] technologies, Industrial Internet [8], or Smart Cities [9], all of these efforts aim to improve society through the harnessing of data and resources from distributed systems. In recent years a number of CPS software platforms [10, 11, 12, 13, 14, 15] have been developed. The majority of these platforms rely on public cloud providers such as Amazon EC2 [16] or Microsoft Azure [17] to centrally process and store CPS data. The use of public cloud infrastructure in support of CPS requires that some if not all of data generation and decision making take place in remote data centers. For instance, the communication of location information between two mobile devices in close proximity might require transmission through a public cloud hundreds of miles away.

In May 2016, the Cyber-Physical Systems Public Working Group (CPSPWG), a forum established by the National Institute of Standards and Technology (NIST), published their *Framework for Cyber-Physical Systems*[18], defining CPS as "smart systems that include engineered interacting networks of physical and computational components." The CPSPWG framework described the need for "Migration of Functionality (vs Data)", a computational paradigm where processing takes place on the logical edge of networks instead of transmitting data to remote data centers. While it is common to make use of graph-based models to represent aspects of locality and hierarchy in large (global) networks [19], the term edge should not be considered a specific tier as designated in hierarchical internetworking models [20].

While there does not appear to be a generally accepted definition, we understand Edge Computing to be the intelligent processing of data throughout networks including near sources of data generation, within points of data transmission, and in collections of centralized data center resources. The author believes that the importance of edge computing will increase with the size and complexity of CPS deployments and

[1]The author does not suggest these terms are interchangeable only that overlaps do exist.

resulting data processing. In addition, the coordination of global edge-to-edge communications, shared resources, high-level application scheduling, monitoring, measurement, and Quality of Service (QoS) enforcement will be critical to address the rapid growth of connected devices and associated data.

While in its infancy, several authors have provided their interpretations of the *edge* or *fog* computing paradigm, most notably Bonomi et al., 2012 [21], Lopez et al., 2015 [22], and Varghese et al., 2016 [23]. In the following sections the author will provide his interpretation of the motivations and characteristics of edge computing.

1.1 Motivations

Motivations for the adoption of edge computing over existing distributed computational paradigms include challenges such as operational constraints, gains in efficiencies, and better control of data and privacy. The following subsections describe the motivations for edge computing in detail.

1.1.1 Resource Constraints

There exist an increasing number of CPS applications where the traditional computational paradigm of central data center control, data collection, and processing are impractical. There are computational constraints on smart devices, where limited hardware or software prevent on-device analytics and communications constraints prevent cloud-based offloading of decision making. For example, oil pipelines are typically found in remote areas where satellite communication is used to communicate sensor status to Supervisory Control and Data Acquisition (SCADA)[24] monitoring systems. Historically, SCADA systems acquire sensor data directly from Programmable Logic Controllers (PLCs) that convert sensor signals to digital data. Typically, in order to communicate large amounts of oil flow sensor[2] data to a central SCADA sys-

[2]Oil flow sensors are used in leak and theft detection.

tem, expensive broadband satellite system must be used. However, recently companies [25] have employed edge computing techniques in pipeline monitoring to increase measurement accuracy (detect smaller leaks), decrease alert latency, and reduce communication cost. In one such example [26], highly sensitive pressure sensors are used in place of oil flow sensors, sensor data is analyzed before transmission, and inexpensive low-bandwidth satellite communication is used to communicate the status of the local monitoring station in place of transmitting distinct sensor samples to the central SCADA monitoring system. Pushing computational resources to the edge enables decision making to take place near sources of data.

1.1.2 Predictive Requirements

The SCADA example in the previous section describes a scenario where both connectivity and local analysis capabilities were limited. While satellite communication serves as an extreme example, variations in traffic impact the most highly connected communication networks, especially those participating in the public Internet where utilization can change rapidly without warning. The term low-latency is relative to the context of specific applications, where latency must fall within an acceptable range. Real-time systems must guarantee processing within specified time constraints. In order for real-time systems to operate in a distributed capacity both predictive computation and network latencies are necessary. Applications that require predictive latency can neither rely on the public Internet for communications or public cloud services for computation as guaranteed resources. However, resources with known performance characteristics, including communication latency to sources of data generation, can be used. Satyanarayanan et al., 2009 [27] proposed the idea of cloudlets, small-scale clouds on the edge of networks, to support resource-intensive low-latency applications. For example, systems that coordinate real-time video, such as those from wearable cameras must transmit, encode, decode, and retransmit video streams in less than 25-50ms, or video latency is detected by users [28]. The latency to

communicate from a wireless transmitter through the public internet and back alone can easily exceed real-time video requirements, so in the current paradigm one must choose to operate in a standalone mode that lacks significant video analysis resources and content distribution or in a remote data center mode with unacceptable latency. A real-time video system operating in an edge computing environment can coordinate video streams locally, while providing enough analytic capacity to detect interesting events from one or more video streams, which the edge environment can act on or direct to cloud resources for further processing.

1.1.3 Heterogeneous operations

Self-contained applications rely on tightly controlled vertical stacks of infrastructure, devices, and software resources. Edge computing techniques can be used to ensure application constraints are satisfied throughout the operating environment. As in the previous example, real-time Smart Cities applications [29], such as traffic management [30], require predictive processing on street-intersection, sub-city, and city-wide scales. In addition, the interoperation of external federated participants, such as mobile devices and vehicles required localized decision making. Not only must external participants interoperate with distributed edge analysis services, the coordination of autonomous Vehicle-2-Vehicle (V2V) interactions [31] between vehicles and Smarter City infrastructure need to be accommodated through city-based edge [32] computing environments.

1.1.4 Devices and Data

Smartphones potentially contain dozens of sensors with the ability to perform limited processing, storage, and transmission of data generated on the device. Vehicles contain networks composed of hundreds of sensors and provide real-time processing, storage, and potentially transmission capabilities for data generated by and in proximity to vehicles. Smart homes might contain a number of sensor networks representing

thousands of potential data sources, with computational, long-term storage, and high-speed communication resources. Commonly, on-device sensor data is confined to the devices and systems where it is generated. While not all sensor data is relevant in isolation, coordination of data streams from independent devices results in aggregated streams of data that can yield new detectable events. For example, coordination of smartphone data is used in sensing spatial and temporal personnel coordination [33] and disaster relief operations [34]. However, even the most robust IoT cloud platform is unable to centrally collect and process all on-device generated data for large collections of interacting devices. Edge computational resources can be used to perform intermediate operations, such as aggregated processing or filtering, between end-user devices and higher-order analysis services. For instance, data probes placed in a network exchange[3] for cybersecurity monitoring communicate high-rate network flow traffic to edge devices for analysis [35]. Anomaly detection is performed on edge devices through the aggregation of probe data. When local anomalies are detected, data is propagated to regional or national services for aggregated correlation. Likewise, if specific network flows are requested from end-devices, edge resources are used to filter and propagate only the data that is needed for higher-order analysis. The previous examples demonstrate how edge computing techniques are used to correlate and manage data that would otherwise not be available.

1.1.5 Smart device capabilities

Smartphone data management is especially challenging due to their natural distribution and ability to generate large amounts of data through human-device interactions. While cellular networks have long been used to address corresponding challenges of mobility and data transmission, no comparative edge computational component exist. Smartphones and other devices must either operate with central coordination or use ad-hoc device-to-device methods. Both cellular and wired telecommunication

[3] A facility where many independent networks connect to (peer) with each other.

network services are provided through telephone exchanges or (as more commonly referred) central offices (CO). COs provide telephony and data services for relatively small geographic areas, which makes them ideal locations to place edge computing resources. Currently, efforts are underway as part of the *Central office re-architected as a data center* (CORD) [36] project to replace proprietary central office hardware with open source software and commodity hardware. One use case of CORD is to simplify the process of providing Content Distribution Networks (CDN) [37] resources on the edges of networks. While typically end-device interactions between CDN and other similar services are centrally coordinated, pushing such resources down to the edge allows for CO-to-device service discovery. As with CDNs, the availability of general purpose edge resources in COs allows for the end-user coordination of CPS, P2P, M2M, IoT services. General purpose shared edge resources, like those proposed by CORD efforts, are critical to realization of Smart Cities.

1.1.6 Decision making and security

While edge computing extends far beyond telecommunication service delivery, the potential for disruption in a historically tightly controlled and highly regulated industry is of great importance to the privacy and security of CPS. The migration of service discovery and delivery from central cloud to CO coordination represents a paradigm shift in decision making from the service provider to the end-devices. While the outcome will be determine by implementations and policies, CO-level coordination provides the capability to push control and trust down to the human-level. More often that not, users are unaware [38, 39, 40, 41, 42, 43] of the information that is being shared by their devices. Even if users are aware of privacy policies, with application control in the public cloud end-users must trust infrastructure, device, and application providers to enforce privacy policies. For instance, thermostats, garage openers, and other smart devices detect if users are away from their residence, but this ability creates to potential for criminals to learn this information as well. With decision-making

related to data sharing and processing pushed down to the device-level, users have more options to control what data is shared and how it is processed. Network Function Virtualization (NFV) [44] is a concept that decouples network functions from proprietary hardware allowing services to run in software. Edge resources, like those provided by the previously mentioned CORD, are capable of running user-defined NFV functions. With decision making pushed down to the device, with capabilities for devices to provision NFV functions, data processing pipelines with individualized privacy and security policies are possible. For example, suppose for privacy purposes one wished to limit location data from their personal mobile devices to their home automation system. Currently, mobile devices and home automation systems must coordinate communications through public infrastructure and cloud services. Sensitive data is transmitted alongside unprotected data by the device through the communication provider and the central applications must provide security and privacy enforcement to all end-points. However, with decision making pushed to the end-device policies can be applied from the bottom-up to enforce security and privacy. For instance, an end-device can discover services provided by their current CO, which could include the users home automation system. If communications were required between COs, edge coordinators or end-devices could provision NFV services to enforce user-defined policies. In this mode of operation, no only could users define their own policies, they also could specify the methods and technologies used in security and policy enforcement. The same reasoning can be applied for privacy of biometric sensor data from personal mobile devices to private analysis services. For privacy purposes biometric is restricted until an actionable event is detected, which at point the best-fit medical personnel are alerted and streams of biometric data are pushed to medical services.

The previous examples describe decision making in the context of users and end-devices. However, it is unlikely that most end-users will design services using edge resources and NFV building blocks on their own. End-users and those they interact

with can negotiate policy-enforcing specifications, which can be implemented by edge computing technologies. For example, with smart grid [45] technology energy usage and scheduling is shared with power companies to make scheduling decisions. As previously mentioned, knowledge of residency occupation in the wrong hands is not desirable. Likewise, for privacy reasons, users might not want their home devices communicating directly with power companies. Edge computing could be used to broker agreements between power companies and home automation systems, where users have control over what data is shared and potentially where data is processed. In a brokered scenario, data might be sanitized at the users edge or perhaps necessary demand response calculations performed in the residence or a local CO.

The sharing of healthcare information is an extremely complicated topic, where additional information in the right hands can be beneficial, but in the wrong hands highly undesirable. In the event of an emergency, possibly without end-user identification, we want critical life-saving information to be shared with medical personnel. In addition, many people would agree to share personal healthcare metrics for prediction analysis and alerting purposes to avoid potentially life-threatening events. However, ubiquitous medical record data, at least in part, is not a reality due to the concerns surrounding the misuse of personal medical data. With edge computing, data sharing, communication, and alerting policies between healthcare providers and individuals could be enforced across devices, communication, and healthcare provider networks. Edge computing technologies provide a platform for users to make decisions about their data.

1.1.7 Workload Optimization

Perhaps the most compelling argument for edge computing is the ability to gain efficiencies by intelligently matching workloads, resources, and data sources. The previous sections described examples of where moving computation to data enables new features and benefits not otherwise possible. While there are intrinsic efficiencies, as

we have described, by moving computation to the edges of networks, there are broader impacts to resource management working in the edge computing paradigm. In order to understand what computation should exist on the edges of networks, one must first have an understanding of the functional components of applications and there relation to infrastructure. For instance, application components that require transactional access to synchronous data sources are highly impacted by increases in latency. Conversely, components that provide data to reporting systems asynchronously are more tolerant of increases in latency. A single threaded process used for anomaly detection is less tolerant of processor oversubscription than the long-running batch processes used to generate the detection models. Resources on the edges of networks, such as those found in COs, provide limited computational resources in comparison to public clouds. One can think of edge resources as prime real-estate, where resources cost is a premium. From a business model prospective, edge resources are a store front, while public clouds function as warehouses and production facilities. As with any business, high-level views of operations must be continuously evaluated to align needs with appropriate resources. Edge computing workloads can be optimized in various ways including cost [46], network characteristics [47], computational performance [48], and power consumption [49] to name a few.

In several examples, we have described edge computing as resources that exist on the logical boundaries of networks providing intermediate resources between end-devices and public clouds. However, edge computing techniques can be applied throughout the infrastructure-application hierarchy. For sufficiently large and complex services, especially those that are distributed globally, a top-level logical edge of the network could be at a point of international network peering, with multiple lower-level edges extending down to the end-device. One such global service is the Netflix CDN, which uses Amazon Web Services (AWS) [50] infrastructure. The Netflix CDN provides video services for over 86 million subscribers in over 190 countries [51]. Estimates place the global AWS server count in the millions [52], with AWS

regions typically containing 50-80K servers, over 80K fiber connections, and over 100Tb/sec of network capacity provisioned per data center. Netflix must maintain an acceptable customer experience with minimum cost, through management of globally distributed CDN running on a vast AWS infrastructure for which they have no low-level visibility or control. Netflix must determine the real-time viewing experience for for millions of simultaneous data streams across the globe, detect unacceptable performance, and adjust resource configurations throughout the service delivery pipeline to maintain services. Netflix must be able to change consumer-side settings such as lowering stream bit-rates or redirecting request to alternative data sources. Likewise, producer-side changes such as the provisioning of additional edge computational or network resources on the continental, regional, data center, or CO levels must be predictive and reactive based on expected usage patterns and observed needs. Edge computing technologies allows a platform to predict, detect, correct, and optimize distributed applications.

1.2 Characteristics

In the previous section we described motivations for adopting edge computing practices. In this section we will provide a characterization of edge computing. Edge computing implementations, such as one presented in this dissertation, should represent at least a subset of edge computing characteristics discussed in this section.

1.2.1 Proximity

Edge computing is placement of computational resources in close proximity to points of data generation. As previously mentioned, edge computing shifts the paradigm of moving data to computation, to moving appropriate levels of computation to data. Moving computation to sources of data creates a natural requirement to support decentralized systems. Data is generated all over the world, and as such edge computing

technologies must support geographic distribution. While often described as computational resources at the logical network boundary between communications networks and end-devices, edge computing techniques can be extended to any number of logical boundaries in software or infrastructure architectures. Logical boundaries in systems often represent technical demarcations, where one level of the system is dependent on some other level. For example, downstream network providers depend on one or more up stream providers. Likewise, an application may depend on a virtual machine, which depends on a hypervisor, which in turn depends on underlying hardware system. We discuss additional details pertaining to infrastructure virtualization in Section 1.3.3, *Global Observation*. The natural hierarchy that exist in distributed systems necessitates that on a high-level edge computing frameworks must to function in a hierarchical fashion.

1.2.2 Communications

In previous examples we had discussed the benefits of edge computing for predictive and low-latency applications. Edge computing reduces communication and computational latency by moving resources closer to points of data origination. A hierarchy of edge computing resources distributed in communications infrastructure allows for the measurement and monitoring of network characteristics between two edge-enabled network boundaries. Observation of network conditions by edge resources can be used by Software Defined Networking (SDN) [53] controllers to manipulate low-level communications infrastructure data paths to improve communications. Edge computing technologies must participate actively in communications networks, discovering edge-enabled network topologies, relaying link characteristics between edge nodes, and by reporting observed network characteristics to low-level communication control systems.

In addition to communication performance, edge computing technologies must function as data exchanges between heterogeneous networks and devices. Functioning

on the application-level, edge resources must support simple high-level edge-to-edge messaging abstracted from lower-level multi-protocol communications. In addition, edge computing must provide methods to establish and control data plane operations between underlying application and infrastructure components. Similar to the separation of control and data planes in the SDN OpenFlow [54] protocol, high-level control messages are communicated between edge nodes and devices to provision data paths and exchanges from low-level infrastructure to edge-managed applications. Edge technologies must be able to establish and maintain appropriate communication channels between data generators and consumers.

There are twice as many reported devices in China alone than total number of public IPv4 network addresses (4 billion). Edge computing frameworks must support communication protocols such as IPv6 (3.4×10^{38} addresses) with large address spaces. With the high prevalence of wireless devices in CPS networks, protocols such as 6LoWPAN [55] and ZigBee [56] must be supported. Edge frameworks must be able to provide both low-level protocol routing and high-level data translation services between devices, intermediate, and back-end processing resources. Where necessary, edge technologies must be able to translate low-level protocols and high-level application data between end-points. For example, an edge device might use the low-level ZigBee protocol to acquire data from a low-power low-bandwidth wireless sensor, which is then communicated to a cloud-based IoT application using the high-level AWS IoT API [10].

1.2.3 Target Devices

Currently, edge computing efforts focus on end-devices such as smart phones, tablets, standalone IoT devices, and supporting edge resources such as virtual servers. While existing device and data processing requirements can benefit from edge computing techniques, future requirements to support an increasing diverse array of data generating devices stand the most to gain through the adoption of edge technologies.

Advances in centrally provided computational and communication capacities have not yet grown to meet future demands in industrial, agriculture, transportation automation, healthcare, smart homes, campuses, and cities. A single autonomous car is capable of generating 4 terabytes of data daily [57], as a result the 260 million registered vehicles [58] in the US alone are capable of generating over a quintillion (1×10^{18}) bytes of data on a daily basis. Likewise, cities like Chicago are deploying general purpose sensor arrays to "track the city's vitals" [59]. The so-called sensor *Array of Things* deployed in Chicago will be used to generate both information for research and also city-wide decision making. While data generated from general purpose sensor arrays is multipurpose in nature, the policies pertaining to usage will vary greatly based on application. For instance, aggregated air quality information used to detect real-time threats to public safety should have a higher computational and transmission priority than the same information used for a long-term climate change study. However, while it is possible to provide traffic priorities between applications, existing end-to-end infrastructures typically don't differentiate between intra-application communications and processing priorities. Edge computing frameworks must be able to manage sensor arrays, communication networks, and computational resources in large data-intensive heterogeneous environments. In addition, such frameworks must extend operations beyond interactions with end-devices to communication and computational infrastructures providing end-to-end policy enforcement. The net result is a requirement for edge frameworks to support a wide range of devices, including resource components used in the provisioning of infrastructure and services. There is a need for standards and protocols to support edge computing. Our work is a step in this direction.

The next section describes several key challenges that must be addressed by edge computing frameworks.

1.3 Challenges

Above we discussed the motivations for and characteristics of edge computing. In this section we discuss several challenges presented by modern cyberinfrastructure (CI) that must be overcome by edge computing frameworks.

1.3.1 Comparison

Generally, with every layer of computational abstraction usability is increased, while the underlying complexity is also increased. For example, it is much easier to write an application in a higher-order language than in a machine-specific language. However, this abstraction makes things like tracing specific hardware operations, and their order of execution on lower abstracted levels, more difficult.

CPS-supporting infrastructure includes a wide-range of physical devices distributed across geographic regions. Public, private cloud, and telecommunication resource providers make use of CI frameworks to manage resources. CI frameworks, such as OpenStack [60, 61], provide access to logically separated layers of compute, storage, and network infrastructure. Most often hardware virtualization techniques are used to provide dynamically reconfigurable resources. Popek and Goldberg [62], defined the term Virtual Machine (VM) as "an efficient, isolated duplicate of a real machine." As with VMs, methods for infrastructure virtualization have been extended to network, storage, and even mobile [63] devices.

From the perspective of device operating systems (OS) and applications running under the OS, the presence of CI management, and related virtualization layer, should be transparent. However, the performance of virtual resources are not necessarily representative of the capabilities of the underlying hardware they represent. For instance, virtual hardware representations can exceed the total capacity of underlying hardware, resulting in resource oversubscription. Even simple tasks such as determining application resource needs from the physical infrastructure layer are often

more difficult, due to this further abstraction of the hardware layer. When using virtual CI one might have no information as to where the underlying physical hardware supporting specific virtual machines resources are geographically located, much less have visibility into its operating state. In this paradigm one would have little or no visibility into the underlying infrastructure layer. Performance of provisioned resources from identically sized virtual machines vary based on the characteristics of the underlying infrastructure. Consider the case where two identical virtual machines are running identical workloads. Despite running identical instructions and observing similar resource utilization metrics from the OS-level (inside each VM, not the host), one workload may finish much sooner than the other. Likewise, two virtual network circuits connecting islands of infrastructure can be presented as virtually equivalent resources, but in practice the performance of the two circuits may greatly vary. This variation could be related to inherent differences in the performance of the underlying infrastructure or could be a direct result of resource limits imposed on virtual resources, where one resources is assigned a greater share of the underlying infrastructure than the other. Even if the underlying infrastructure and related configurations are identical, contention for resources may create performance variations. In this context, *resource contention* is the reduction in performance due to resource request exceeding available resource assignments. Even more troublesome, CI resource costs are typically based on the size of the reserved resource and the duration of reservation. So, assuming the two VMs in our example have the same unit cost, the slower execution actually cost more money, not less. In order to compare the performance of resources, including those that are reportedly equivalent by underlying infrastructure managements systems, we will perform micro-benchmarks and report point-in-time evaluations to the framework. With resource performance and financial cost information, resource can be acquired, decommissioned, or exchanged, based on resource value, thus creating a market.

1.3.2 Identification

In modern infrastructure many layers of virtualization abstract resources from not only underlying hardware, but also from other virtual resources. NFV, as discussed in Section 1.1.6, *Decision Making and Security*, is commonly used to provide network performance enhancement, firewall, intrusion detection, and other security services that isolate groups of back-end application from external networks. Even if one could query these isolated back-end resources, the resources themselves might not be able to provide information needed to identify their relationship between external networks and dependencies.

Consider the case where a request is made to a CI framework to provision two back-end virtual machine (VM) resources to be used in separate applications. The two VMs are requested from CI with networks in the same network address range (e.g. 192.168.1.0/24), but the VMs are for two isolated applications and should not be on the same logical network. This request would be invalid if the specified address range was public [64], since this would be a duplication of assigned address space. However, it is perfectly acceptable, and very common to duplicate private [65] ranges. In our example two separate OSI Layer 2 (L2) [66] networks are required to separate the two computers. Unfortunately, two networks with overlapping address ranges can not be assigned to a single host. So, the CI framework must either use separate physical machines or further abstract resources using *namespace isolation* [67]. Namespace isolation, as the name suggest, allows for the kernel-level isolation of resources, including network devices. This means that single nodes can host many types of compute, network, and storage configurations without worry of namespace (devices, networks, storage, etc) conflicts. The drawback of this method is that it creates yet another layer of abstraction between infrastructure and virtual resources. In this example, if a single node provides computational resources for both networks using namespace isolation, neither networks will be directly accessible from the node without explicitly specifying the namespace of the desired network. Figure 1.1 shows

two physically connected nodes on the same address range as requested for the virtual machines in the previous example. In the figure below there are also two VMs per physical node, which also share the same address range as the physical host. The VM networks are isolated using separate namespaces on the individual hosts, which prevents conflict with the network of the physical host.

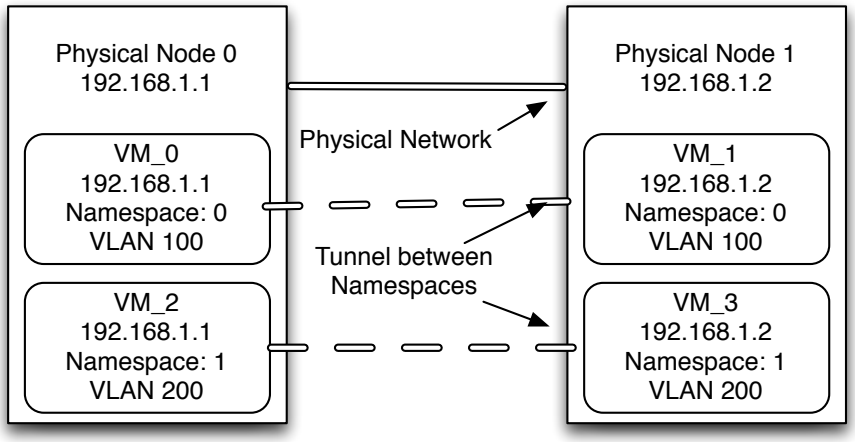

Figure 1.1: The same network range in three separate Linux Namespaces.

It is common for virtual machines to be distributed across many physical nodes. When these virtual machines and their connected networks are namespace-isolated, tunnels linking network namespaces must be created. Typically these network namespaces are connected using fully connected "mesh" tunnels between participating physical nodes. From the standpoint of the virtual machines, they are all connected to the same L2 (VLAN with same ID) network. However, outside of this namespace-isolated network, these virtual machines are completely unreachable. To provide connectivity to networks external to the namespace-isolated network, virtual interfaces are created to bridge traffic between networks. However, this gets us back to the original problem of separating networks with overlapping address ranges. To get around this problem, private ranges are typically one-to-many (one address representing many other addresses) Network Address Translated (NAT) [68] from within the namespace-isolated network. In the telephony context, this is analogous to extension numbers being

assigned to internal nodes, where the extensions are part of a single phone number. This allows traffic originating from within the network to reach external sources. If network services on the namespace-isolated network need to be exposed to externally originating traffic, typically a one-to-one *floating address* is assigned by the CI framework to the NAT interface. Once again using the telephony analogy, this would be like assigning a *direct inward dial (DID)* (phone number) number to a specific extension. Figure 1.2, shows the assignment of floating addresses to namespace isolated VM networks.

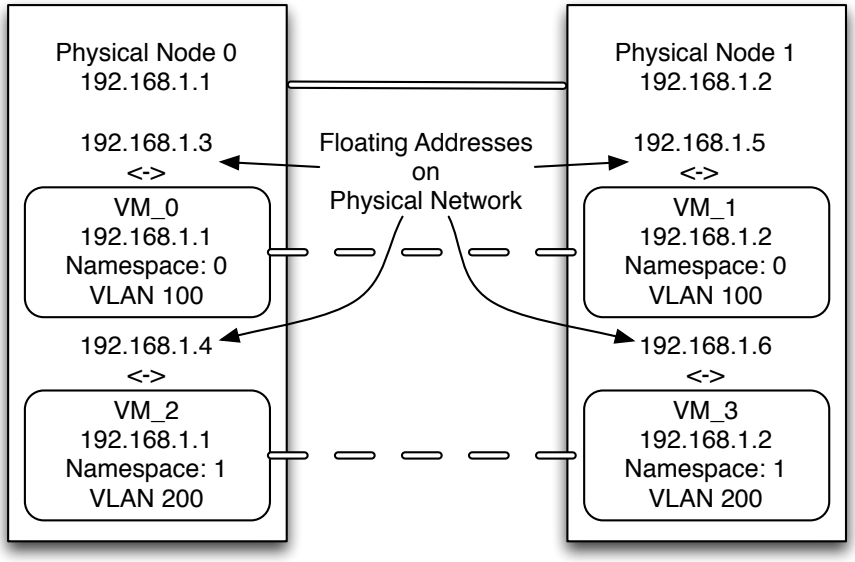

Figure 1.2: Linux Namespaces Network with Floating Addresses

One-to-many and one-to-one NAT can provide connectivity to namespace-isolated VMs. However, on the OS-level a virtual machine will be unaware of the external address representing its services.

Techniques used by CI frameworks to provide network services can leave resources inaccessible beyond CI regions.

Due to the lack of direct external connectivity to resources provided by CI, communication channels must be established to directly manage otherwise unobservable resources. In addition, network discovery methods must be implemented outside the

CI framework to provide topology information the necessary to correlate isolated pools of resources.

1.3.3 Global Observation

Most CI frameworks provide programatic methods for the collection of resource utilization in the form of the quantity of resources actively provisioned. This occupancy utilization is needed for reporting and billing purposes. A number of CI frameworks, from the infrastructure standpoint, provide visibility into the discrete utilization of compute, storage, and network utilization for resources under their control. However, as we previously discussed, utilization metrics of virtual infrastructure is not always sufficient to determine performance characteristics of underlying CI infrastructure. Even if CI utilization metrics were comparable between regions, these metrics would only be useful in evaluating performance for individual resources operating in a specific region. While one might be able to correlate application performance between resources on separate CI regions, this inter-regional relationship is not directly observable using resource utilization metrics provided by existing CI frameworks. Consider the case where an application is exclusively deployed in $region_a$. This application requires front-end (Web node) and a back-end (database node) regional resources to function. Suppose furthermore the same application is independently and exclusively deployed in a region identical to $region_a$, called $region_b$. The utilization metrics of the individual regions would be comparable, for comparable workloads. This is to say, if workload performance is understood based on the performance characteristics of $region_a$, we can expect the same results in $region_b$. Now, suppose we configure the application to allow for provisioning of interoperating resources across regions. For example, a front-end node could consume services from a database node either in the same region or a remote region. In this scenario workload performance can not be directly interpreted based on the CI utilization reported by individual regions. Due to the lack of observable performance metrics between CI regions, application-specific

metrics must be generated to determine relational resource evaluations, as previously mentioned in this chapter. Edge frameworks must address the evaluation, communication, and analysis of metrics globally. These evaluations can be used in scheduling optimization calculations.

In the next chapter we present the architectural model for an edge computing framework.

2

The Architectural Model

In this chapter we describe the architecture model for a edge computing framework we call *Cresco* [69]. First, in Section 2.1, *Functions*, we cover what we believe to be functions required of edge frameworks. Second, in Section 2.2, *Computational Models*, we discuss the computational models that have influenced Cresco framework architecture. Next, in Section 2.3, *Operating Principles*, we discuss the principles that guide the architecture of the Cresco framework. Finally, in Section 2.4, *Cresco Architecture*, we cover the high-level architectural design of the Cresco framework.

2.1 Functions

In the previous, introductory chapter, we described the motivations and general requirements for edge computing. In this section, we describe specific functions required of edge computing frameworks.

2.1.1 Data Processing

Data processing functions provide the ability to communicate, exchange, and modify data in and between points of data generation.

- Real-time data operations such as data filtering, aggregation, and complex event

analytics.

- In-line data and protocol exchange, translations, and transformation.

2.1.2 Command and Control

Command and control functions provide framework intelligence and operations management capabilities.

- High-level message passing for both control and data processing operations.
- Device, CI, and global application provisioning and coordination of resources.
- High-level CI description language to be used in resource management orchestration.
- Discovery services to determine operational topology and potential resources.

2.1.3 Global Visibility and Actions

- Provide a global view of resource topologies with correlated monitoring and measurement of underlying resources.
- Provide a global view of application topologies with key performance indicator reporting.
- Provide global scheduling services, based on static and dynamic methods.

2.2 Computational Models

The Cresco framework is heavily influenced by previous work in *Actor* and *Agent-based* computational models as described below.

2.2.1 Actor Model

On an abstract-level Cresco framework processes are based on Actor-model [70] distributed concurrency. In this model an Actor is a primitive unit of isolated computation that uses asynchronous messaging to communicate with other Actors. While the details of the Actor-model are beyond the scope of this dissertation, basic operations of Actors include message-based creation of more Actors, Actor-to-Actor messaging, and the generation of state decisions applying to the next arriving message. Erlang [71], an example of a popular programming language based on the Actor-model, introduced a "let it crash" philosophy for distributed computation. Instead of focusing on defensive programming to prevent failures, using an offensive (create, monitor, and verify) philosophy one relies on Actors to supervise other Actors creating "self-healing" distributed processing environments. In addition, the isolated operation of Actors makes continuous self and supervisor reporting of KPIs across heterogeneous environments possible.

Actors operate in isolation with the exception of inter-Actor messages, making Actor communication critically important, especially in cases where resource providers and consumers are geographically distributed. The Cresco Actor-model implementation aims to address challenges related to interoperability, performance, and security related to Actor communication channels.

Actors are typically represented as critical sections of code in programming frameworks like Erlang, Scala [72], and Akka [73]. Cresco Actors can be critical sections of code executing from within the native framework or abstracted interfaces (APIs, CLIs, etc.) into external systems. For instance, a Cresco Actor responsible for aggregating data within the framework communicates with other Cresco Actors responsible for monitoring, measurement, and processing of an external sensor networks. Within the Cresco framework all components are consider Actors.

2.2.2 Agent-based Model

Agent-based modeling (ABM) [74] is a computational model used in the simulation of agent interaction. There exist a large body of research for ABM across many disciplines, including: biology, economics, social sciences, and engineering. We are not developing agent programming, which is an existing large area in the methodology of programming, but rather use an existing framework provided by V.S. Subrahmanian, et . al [75], shown below:

- An agent provides one or more useful services that other agents may use under specified conditions.
- An agent includes a description of the services offered by the software, which may be accessed and understood by other agents.
- An agent includes the ability to act autonomously without requiring explicit direction from a human being.
- An agent includes the ability to succinctly and declaratively describe how an agent determines what actions to take even though this description may be kept hidden from other agents.
- An agent includes the ability to interact with other agents, including humans, either in a cooperative, or in an adversarial manner, as appropriate.

One area of ABM research that is of particular interest is Agent-Based Computational Economics (ACE) [76], where the agents themselves participate in agent-based micro-economy. In addition, there is a body of research covering the development of decentralized agent-based markets (dispersed exchanges without auctioneer [77]). We argue that an agent participating in the global management (scheduling) of computational resources, is participating in a dispersed resource exchange, without an

auctioneer. We have adapted previous ACE research in the development of resource-based and price-based markets provided by the Cresco framework as described in Section 4.3.1, *Guilder*.

2.3 Operating Principles

In this section we identify our guiding operating principles in the development of the Cresco Framework.

- Components should interoperate directly with other components through message passing.
- Components should be arranged in such a way as to create a hierarchy of trust and delegation, where higher-level components delegate responsibilities to lower-levels.
- Components should operate across a wide variety of underlying architectures, platforms, and devices.
- The Cresco framework should support a large number of common predefined services.
- Decision making should take place at points of data generation, communication, transformation, or processing.
- Components should be self-maintained, self-reporting, and operate autonomously once directives have been delegated.
- Components should maintain operational status based on configuration directives in conjunction with self-discovery.

In the next section we provide a high-level view of Cresco architecture.

2.4 Cresco Architecture

The Cresco framework was created to assist in the development of globally distributed applications where data collection and processing take place on the network edge. Using Actor and Agent-model computational techniques, the framework provides the ability to coordinate the processing and exchange of data between networks of edge and remote data center resources. For example, the framework can be used to develop applications that process high-rate or globally inaccessible data on network edges, assign workloads to and between appropriate edge devices, and coordinate central processing of filtered, enriched, and edge-aggregated data.

Intelligent agent-based frameworks [78, 79] can be used to provide [80] reduced network communication and latency, fault-tolerance, disconnected operation, resource cloning, scalability, and serve as an ideal platform for edge computing. As previously mentioned, Cresco components are modeled as Actors. *Cresco-Agents* and *Cresco-Plugins* are used to support Actor implementations in the framework. Cresco-Agents provide a operating environment for *Cresco-Plugins*, which provide a platform for the development of Cresco framework operations and Actor implementations. The Cresco framework shares aspects of Multi-Agent Systems (MAS) [81], while providing centralized coordination and intervention.

As consistent with Actor-model concurrency, Cresco Actors communicate through the use of message passing. While a number of Agent Communication Langues (ACL) [82] and protocols have been developed we propose our own UTF-8 [83] encoded text messages protocol we call Cresco *MsgEvents*. As with other popular agent-message languages such as the Foundation for Intelligent Physical Agents (FIPA-ACL) [84] and Knowledge Query and Manipulation Language (KQML) [85], MsgEvents are based on performative (communicative verb) functions [86], where messages are defined by classes and may contain contextual parameters. The Cresco MsgEvent protocol is capable of encapsulating other ACLs including FIPA-ACL and KQML for proposes of end-device communication.

Working with agents in a hierarchical model makes it easier to solve issues related to system scalability. Actors are naturally hierarchical [87] with every Actor being a child of another Actor. In the Cresco framework we define three primary operational agent hierarchies including *Agent*, *Regional*, and *Global*. While additional levels of Actor hierarchies may exist within the defined agent hierarchies, agent discovery, communication, and security isolation is enforced within each level. Cresco hierarchies (Agent, Region, Global) are not strictly related to geographic distribution and all hierarchies could exist within a single location. However, there are a number of applications, where Cresco hierarchies are related to geographic distribution. For instance, consider a potential Cresco smart grid application where information from devices (power consumers) within a home influences decisions made by regional and potentially national power grid management systems. In addition, information from national and regional power producers can influence decisions made by home automation systems. In this context, the Agent-level hierarchy would existing within the home, potentially with a Cresco-Agent embedded within a smart power meter [88] functioning as a smart home service gateway [89], using Cresco-Plugins to interface with various lower-level protocols and related devices. In such an arrangement there might exist billions of plugin-managed devices, millions of homes supported by agents, thousands of cites managed as regions, and a small number of global entities with a national view of the power grid. An example of Cresco components used in a smart grid application, arranged in the Cresco hierarchy, is shown in Figure 2.1.

It is sufficient to think of the Cresco Actor-model implementation as a graph of text-defined and text-communicating primitives describing edge-focused distributed applications, implemented as a system of intelligent agents.

2.4.1 Cresco Edge Characteristics

We believe the Cresco framework addresses the following characteristic challenges inherent to edge computing, as defined by Bonomi et al., 2014 [90].

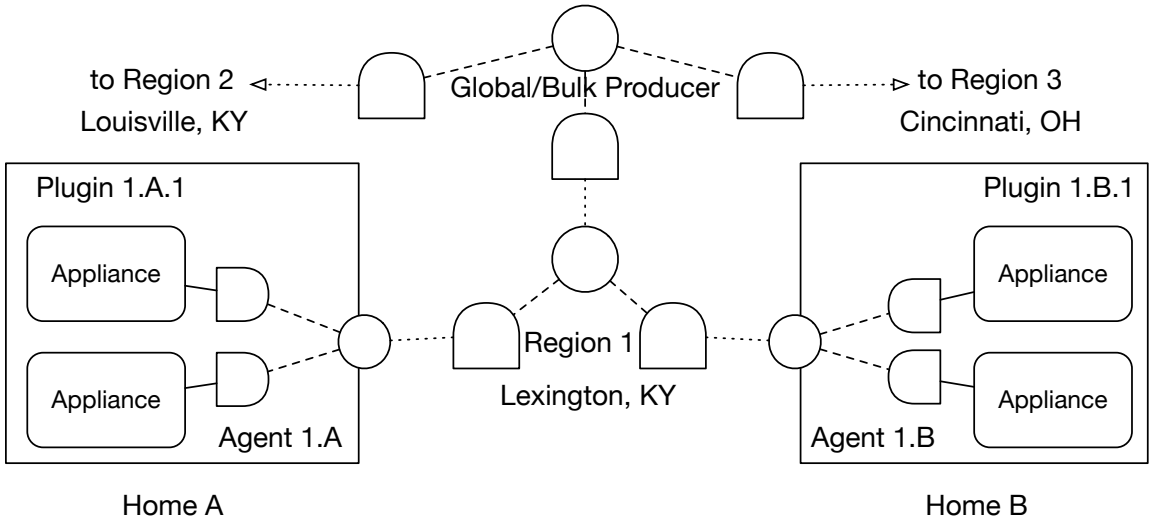

Figure 2.1: Cresco Smart Grid Hierarchy

Low latency and location awareness: There is a tradeoff between moving data to processing resources or moving resource to sources of data generation. The Cresco framework provides a global view of distributed resource performance allowing for the programatic management of both low latency network edge processing and data center processing. The Cresco Agents and Agent-Plugins operate in a hierarchical role-based structure where all Actor assignments and locations are known globally.

Wide-spread geographical distribution: The Cresco framework was developed to operate on a globally distributed level. Global topologies composed of resource producers, consumers, and application components is maintained by Cresco. Cresco components are arranged in a global, regional, agent, and workload (plugin) hierarchy. Using our hierarchical agent architecture, we provide fine-grained control over the structure, communication patterns, and security of distributed systems.

Mobility: Cresco provides high-level messaging services allowing Agent-based Actors across a wide-range of devices and environments to communicate. In our framework Agents can be deployed directly on mobile devices or Agent-Plugins can be used to represent individual external devices or networks of devices. Our text-based mes-

saging protocol allows Cresco participating Agent-based Actors to be implemented in many languages or directly in hardware.

Very large number of nodes: Actor-model concurrency provides great scalability through a natural hierarchical structure of node processes. In our hierarchical management model, higher levels in the hierarchical structure are responsible for the reporting and communication of lower level components. An Actor implemented in an Agent-Plugin is managed by an Agent, which is managed by a hierarchy of agent controllers. Information is propagated between between levels of the hierarchy only as needed, such as changes in topology and Key Performance Indicators (KPI). While the practical limit of nodes will vary based on infrastructure, the current technical limit for nodes and edges stored in the global graph database is $2^{78} - 1$ records.

Predominant role of wireless access: Cresco Agents can be deployed in wireless agents and participate as part of a wireless network. Additionally, Cresco can be used to construct distributed applications using common, light-weight wireless protocols such as MQTT [91].

Strong presence of streaming and real-time applications: Cresco originated from the need to deploy interconnected, but possibly geographically distributed resources for streaming and real-time applications. Using Cresco, large distributed applications can be deployed for real-time event processing and data enrichment, cross-region aggregation, and central stream analysis.

Heterogeneity: A key capability of the Cresco framework its ability to interface with heterogeneous computational, network, and operating environments through text-based configurations and messaging.

A detail description of Cresco component implementations are provided in Chapter

3, *Cresco Implementation.*

3

Cresco Implementation

Cresco [69] is a free and open-source distributed agent-based resource management framework available under the Apache Version 2.0 [92] license. Initial project goals are to provide solutions for resource and workload comparison, performance observation, scheduling, and provisioning.

Cresco is composed of six primary components:

- *MsgEvents*: Protocol used in the communication of Cresco components.
- *Cresco-Agent*: Endpoint resource management unit in charge of orchestrating Cresco-Plugins state, messaging, and monitoring aggregation.
- *Cresco-Plugins*: Work units providing communication channels at the agent, region, and global level, performance monitoring, as well as custom work units through extension of the Cresco-Plugin-Library.
- *Cresco-Library*: Core functions shared by *Cresco-Agent* and *Cresco-Plugins*, such as configuration management, message processing, event logging.
- *Cresco-Plugin-Library*: Common functions used by *Cresco-Plugins*, such as plugin state management and interfaces to core Cresco Library functions.

- *Cresco-Controller-Plugin*: Special plugin used to manage agent topology, discovery, and message communications. The Cresco-Controller-Plugin is required by all agents to establish agent, regional, and global management planes.

Core Cresco components are shown in Figure 3.1.

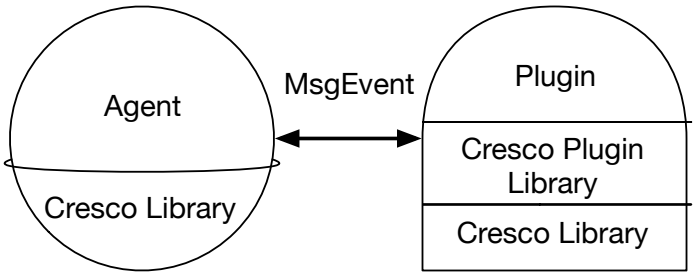

Figure 3.1: Core Cresco Components

The following sections will describe the implementation details of Cresco components.

3.1 MsgEvent Protocol

The Cresco MsgEvent protocol is an Agent Communication Language (ACL) used by all Cresco components for framework communication. These messages are used in both the transmission of data and in the remote execution of functions across the Cresco components. MsgEvents are represented as class objects or as a textual representation of class objects. In this context, class objects refer to programmatic instantiations containing both attributes of state and behavior. MsgEvents can be thought of as objects in the context of object-oriented programming. MsgEvents are represented exclusively as object classes in all components of the Cresco framework, with the exception plugins that convert MsgEvent class objects to and from their textual representation for the purpose of external communication.

3.1.1 MsgEvent Format

Typically, communication protocols are defined as byte addressable structures. These structures are used to map defined byte ranges to locations in memory, which have been populated by an underlying communication system. Where communication performance is of concern, this is a good practice. However, with Cresco we are far more interested in communication interoperability than message passing performance. We will define the format of a Cresco MsgEvent in terms of the *header* and the *body* of the message.

- HEADER
  - *msgRegion*: In the routing process, this value is used to move values in and between regions. This value can be considered the "next-hop" value for message routing on the global-level.
  - *msgAgent*: In the routing process, this value is used to move values in and between regions and agents. This value can be considered the "next-hop" value for message routing on the regional-level.
  - *msgPlugin*: In the routing process, this value is used to move values in and between regions, agents, and plugins. It can be considered the "next-hop" value for message routing on the agent-level.
  - *msgBody*: This optional value is used to describe the body of the message.
  - *msgType*: The enumerated *MsgEventType* class of message.
- BODY
  - *params* : Is the identifier for a key-value collection of parameters.
    - *msg*: This is the value of the msgBody.
    - *dst_region*: The regional destination of a message. If null, the message destination is the global controller.

* *dst_agent*: The agent destination of a message. If null, the message destination is a regional controller, as defined by *dst_region*.
* *dst_plugin*: The plugin slot destination of a message. If null, the message destination is an agent, as defined by *dst_agent*.
* *src_region*: The regional source of a message. If null, the message source is a global controller.
* *src_agent*: The agent source of a message. If the *src_region* value is not null, this message should never be null, since a message sent from a regional controller will include both agent and plugin source values.
* *src_plugin*: The plugin slot source of a message. If this message is null, then the message originated directly from the *src_agent* and is not a plugin message.

In the following subsection we will cover *MsgEventType* values.

3.1.2 MsgEventType in MsgEvent

There are several types of MsgEvents, specified by the *MsgEventType* enumeration. A list of enumerated types and their function are shown below:

- *CONFIG*: message types that are used in the configuration of agent and plugin status. These messages are used to register agents with controller plugins, configure plugin status on agents, and configure plugin configurations on agents. These messages function on the agent, regional, and global levels.
- DISCOVERY: message types, as the name suggest, are used in the discovery of Cresco resources. DISCOVERY messages are used in agent, regional, and global resource discovery. Discoverable agent items are available plugins and plugin status, while discoverable plugin items are defined by the plugin. Plugins will be identified by their *Plugin Slot*, as reported by the agent and their *Plugin*

Name as reported by the actual plugin. At minimum, a plugin must report their current configuration parameters as a discoverable item. Plugin Slots are used to reference plugins on specific agents, plugin configurations are used to identify the function of a specific plugin. These messages function on agent and regional levels.

- *ERROR*: message types that are used to report functional errors in the Cresco system. Error reporting will generally be isolated to specific agents and regions, as errors are an expected part of operations. Errors related to active resources will be propagated from regions to global controllers.

- *EXEC*: message types are used to designate messages that contain executable functions. EXEC message types are routed to sections of code on agents and plugins that provide functions for system interactions such as starting or stopping Cresco-managed resources.

- *INFO*: message types are used to provide informational messages between Cresco components that are otherwise not covered under more specific message types. Typically, INFO messages are isolated to agents and regions. However, INFO messages can be forwarded to a global controller.

- *KPI*: Key Performance Indicator (KPI) message types are data metrics used by the Cresco framework to determine the performance of workloads in relation to resources assignments. These messages report the qualitative and quantitate metrics associated with agent and plugin related resources. KPI messages are used to determine the active operating state of Cresco components in relation to core component functions. KPI performance metrics are propagated to global controllers to provide a central view of the overall system operating state.

- *LOG*: message types are used for logging purposes on agent, region, and global levels.

- *WATCHDOG*: These metrics are generated by agent and plugin resources. WATCHDOG metrics provide uptime and agent configurations information used by the controllers to determine agent topology and health.

The MsgEventType value is used in the routing and execution of MsgEvents. In the next section we will cover the ways in which a MsgEvent is represented in the Cresco framework.

3.1.3 MsgEvent Representations

As previously mentioned, MsgEvents can exist as class objects or textual representations of class objects. All Cresco components share the same MsgEvent class definition through the use of a shared sourced code base. MsgEvent classes contain the data of their textual representations along with functions to manipulate the data. MsgEvents can be converted between valid textual representations, class objects, and back to textual representations without data loss.

The choice to use text-based messages was based on the need for a simple, portable, and flexible control protocol. Text messages can be communicated using many different underlying transports and application interfaces. The following subsections describe text-based formats of MsgEvents and their expected usage. MsgEvent formats and encoding are not limited to those presented in this document. In fact, MsgEvents can be used to encapsulate other common ACLs including FIPA-ACL [93] and KQML [85].

XML The *Extensible Markup Language (XML)* [94] format of a MsgEvent is shown in Listing 3.1. The XML message format is used when the benefits of validating MsgEvent textual representations against the XML MsgEvent schema outweigh the overhead of XML processing and communication.

Listing 3.1: XML format of a MsgEvent

```
1
2  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
3  <ns2:MsgEvent xmlns:ns2="cresco.shared">
4      <msgRegion>[next-hop region]</msgRegion>
5      <msgAgent>[next-hop agent]</msgAgent>
6      <msgPlugin>[next-hop plugin slot]</msgPlugin>
7      <msgBody>[text contained in the param=msg]</msgBody>
8      <msgType>[enumerated type of the message]</msgType>
9      <params>
10         <entry>
11             <key>msg</key>
12             <value>[text representing msgBody]</value>
13         </entry>
14         <entry>
15             <key>dst_region</key>
16             <value>[destination region]</value>
17         </entry>
18         <entry>
19             <key>dst_agent</key>
20             <value>[destination agent]</value>
21         </entry>
22         <entry>
23             <key>dst_plugin</key>
24             <value>[destination plugin]</value>
25         </entry>
26         <entry>
27             <key>src_region</key>
28             <value>[source region]</value>
29         </entry>
30         <entry>
31             <key>src_agent</key>
32             <value>[source agent]</value>
33         </entry>
34         <entry>
35             <key>src_plugin</key>
36             <value>[source plugin]</value>
37         </entry>
38         <entry>
39             <key>[somekey0]</key>
40             <value>[somevalue0]</value>
41         </entry>
42         <entry>
43             <key>[somekeyN]</key>
44             <value>[somevalueN]</value>
45         </entry>
46     </params>
47 </ns2:MsgEvent>
```

Listing 3.2: URI format of a MsgEvent

```
1  HTTP://[Global controller IP]:[Global Controller Port]
2  ?type=[enumerated type of message]
3  &region=[next-hop region]
4  &agent=[next-hop agent]
5  &plugin=[next-hop plugin]
6  &paramkey=msg&paramvalue=[text representing message body]
7  &paramkey=dst_region&paramvalue=[destination region]
8  &paramkey=dst_agent&paramvalue=[destination agent]
9  &paramkey=dst_plugin&paramvalue=[destination plugin slot]
10 &paramkey=src_region&paramvalue=[source region]
11 &paramkey=src_agent&paramvalue=[source agent]
12 &paramkey=src_plugin&paramvalue=[source plugin slot]
```

Listing 3.3: JSON format of a MsgEvent

```
1
2  {
3  "msgRegion":[next-hop region],
4  "msgAgent":[next-hop agent],
5  "msgPlugin":[next-hop plugin slot],
6  "msgType":[enumerated type of message],
7  "params":
8   {
9   "msg":[text representing message body],
10  "dst_region":[destination region],
11  "dst_agent":[destination agent],
12  "dst_plugin":[destination plugin slot],
13  "src_region":[source region],
14  "src_agent":[source agent],
15  "src_plugin":[source plugin slot]
16  }
17 }
```

JSON The *JavaScript Object Notation (JSON)* [95] format of a MsgEvent is shown in Listing 3.3. The JSON message format is used when the benefits of a compact message outweigh MsgEvent protocol-level validation. Hypertext Transfer Protocol (HTTP) POST actions use JSON format on both the submission and response (data returned by HTTP server) messages.

URI The *Uniform Resource Identifier (URI)* format of a MsgEvent is shown in Listing 3.2. The URI message format is used when the benefits of communication

interface outweigh MsgEvent protocol level-validation. HTTP GET actions use the URI message format on input (data submitted to the HTTP server) and the JSON message format on response.

This section explained the MsgEvent format and ways in which it is used. In the following sections we will cover the rest of the components that make up the Cresco Framework. A high-level relationship between Cresco components is shown in Figure 3.2.

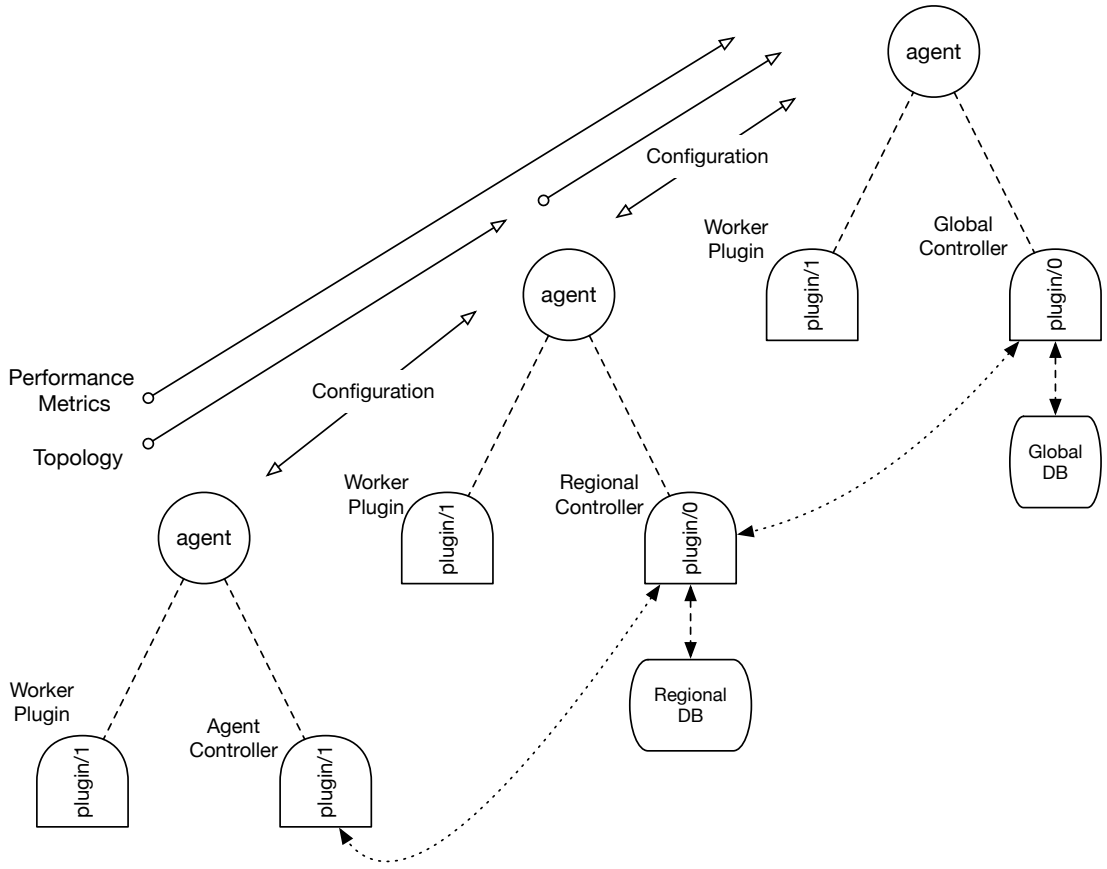

Figure 3.2: Cresco Component Relations

3.2 Cresco-Agent

On a high-level, the Cresco framework functions in the Actor-model, implemented as a multi-agent system. Cresco agents function as actors in the framework, where there primary function is to support the operational tenets of other Cresco components. The Cresco-Agent provides dynamic configuration, loading, and unloading of Cresco-Agent-Plugins, described in Section 3.3, *Cresco Plugins*. The primary role of the agent is to host and pass messages between plugins (new actors). The agent directly routes intra-agent messages to plugins through plugin interfaces described Section 3.4.1, *Plugin Interface*. Inter-agent messages are communicated through queues provided by the *Cresco Controller Plugin*, described in Section 3.6, *Cresco Controller Plugin*. Agent message routing is described in Section 3.2.3, *Agent Message Routing*. The Cresco-Agent provides the underling runtime environment for all native Cresco components.

The remainder of this section we will cover the implementation details of the Cresco Agent.

3.2.1 Agent Initialization

On startup, agents are supplied configuration parameters specified in the *Cresco-Agent.ini* file. The *Cresco-Agent.ini* configuration file must at a minimum provide a *[general]* heading, along with *generatename*, *region*, *watchdogtimer*, *controllerdiscoverytimeout*, and *plugin_config_file* values. An example Cresco-Agent.ini file is shown in Listing 3.4.

During the startup process, if the configuration file can not be validated agent will be terminated. In addition to the agent configuration file a *Cresco-Agent-Plugins.ini* file must be provided, which at a minimum provides a plugin configuration for the Cresco-Controller-Plugin. Details pertaining to Cresco-Agent-Plugins.ini configura-

tion is described in Section 3.2.2, *Plugin Loading*.

Listing 3.4: Cresco-Agent.ini Configuration file

```
1  [general]
2  #Agent name used for static operation
3  agentname = agent-b
4  #Region name used for static operation
5  regionname = region-r
6  #Automatically generate agent name 0=false, 1=true
7  generatename = 1
8  #Automatically generate region name 0=false, 1=true
9  generateregion = 1
10 #Watchdog reporting frequency in ms
11 watchdogtimer = 5000
12 #Startup delay in ms
13 startupdelay = 2500
14 #Discovery delay in ms
15 controllerdiscoverytimeout = 15000
16 #Log producer timeout in ms
17 logproducertimeout = 10000
18 #File path for plugin configuration
19 plugin_config_file = Cresco-Agent-Plugins.ini
20 #Director for plugin files
21 pluginpath = /opt/cresco/plugins
22 #Directory path for log files
23 logpath = /opt/cresco/log
24 #Platform of operation (x86_64, ARM, etc.)
25 platform = x86_64, Dell PowerEdge 720xd
26 #Operating enviorment (metal, VM, container, etc.)
27 environment = metal, standalone server
28 #Location of resources
29 location = (38.034349, -84.504381), Data Center,
30       Rack B4, 538 Rose Street, Lexington, KY, USA
31 #Features of operating enviorment
32 features = Docker, OVS, RabbitMQ
```

Once the agent has validated the agent configuration files the Cresco-Controller-Plugin is loaded. If the agent is configured for static operation the role of the agent is determined exclusively by the configuration found in the Cresco-Agent-Plugins.ini file. In a dynamic configuration agent operation is determined through a discovery process, described in Section 3.6.3, *Controller Discovery*. The example configurations describes startup parameters for a dynamic discovery agent. Once the static assignment or dynamic discovery has completed, the remainder of the plugins described in

Cresco-Agent-Plugins.ini are loaded by the Cresco Agent.

A flowchart for agent initialization is shown in Figure 3.3. The flowchart shows the workflow of the agent taken during the initialization process.

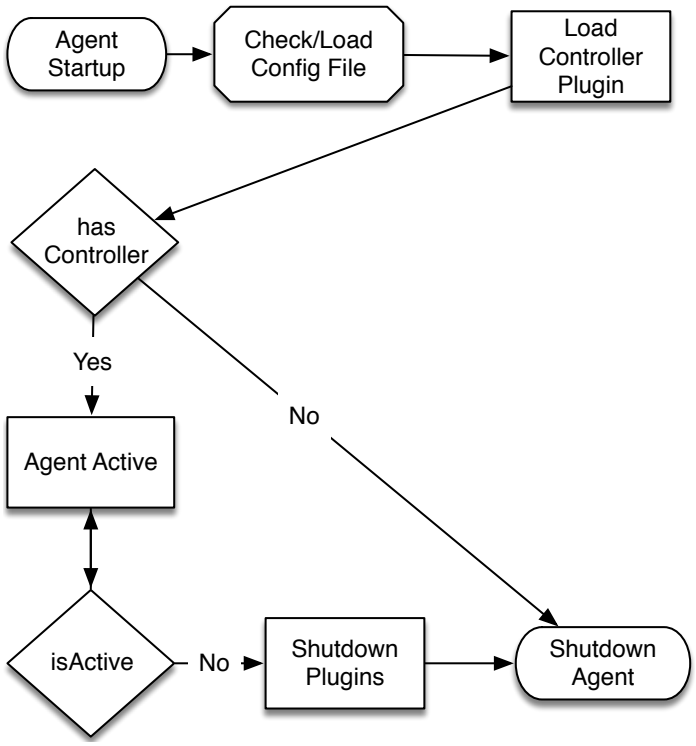

Figure 3.3: Initialization of the Cresco Agent

In the next section we will cover details related to loading plugins by agents.

3.2.2 Plugin Loading

The primary function of the Cresco Agent is to manage Cresco plugins. Within the agent plugins are are identified by their *PluginID* in the form $plugin/N$, where N is the slot number of the plugin and its associated configuration. There are no technical limits to the number of plugins managed by a single agent. An agents plugin capacity is limited by the resources provided by a single node. In practice we have shown that agents are capable of supporting hundreds of plugins. An example of the

Cresco-Agent-Plugins.ini configuration file is shown in Listing 3.5.

Listing 3.5: Cresco-Agent-Plugins.ini Configuration file

```
1 [plugins]
2 plugin/1=<disable=0,enable=1>
3
4 [plugin/1]
5 pluginname=<name of plugin>
6 filename=<plugin jar filename>
7 watchdogtimer=<time in ms>
```

The *Cresco-Agent-Plugins.ini* configuration file must at a minimum contain a configuration for a Cresco Controller Plugin, as described in Section 3.6.1, *Controller Initialization*. During the startup process if the Cresco Controller Plugin fails initialization agent startup is terminated. From the Agent prospective plugin loading is accomplished in the following six steps:

1. *Slot Check*: Verify the requested plugin slot (PluginID) is not currently in use.

2. *Plugin Validation*: The plugin file integrity is verified against the described plugin configuration, plugin object is created on the agent, and plugin object interface is assigned to a plugin slot.

3. *PreStart*: Pre-startup functions are run once in the plugin object namespace before plugin initialization. These functions include setting plugin information based on the state of the host agent or existing application components.

4. *Start*: Start function is called in the plugin object namespace during plugin initialization.

5. *PostStart*: Post-startup functions are run once in the object namespace after plugin initialization. These functions can include notifying applications of the current state of the plugin post-initialization.

Plugin loading functions implemented within the plugin namespace are described in Section 3.4, *Cresco Plugin Library*.

Once initialized, plugins will start transmitting WATCHDOG messages to their host agent. The host agent propagates topology information to regional controllers and likewise regional controllers communicate topology changes to global controllers. Detailed information pertaining to agent and plugin discovery is described in Section 3.6.3, *Controller Discovery*.

From the agent prospective, intra-agent communication is messaging between an agent and its hosted plugins. In intra-agent communication messaging the region and agent source and destination addresses are the same. Inter-agent communication is messaging between two agents or plugins hosted on another agent. In Inter-agent communication source and destination agent and potentially region addresses differ. Intra-and inter-agent communications require messages to be routed to appropriate destinations. Message routing in the Cresco-Agent is described in the next section.

3.2.3 Agent Message Routing

The ability to pass messages between components is a fundamental aspect of distributed systems, agent-based frameworks, and actor-model concurrency. In the implementation of the Cresco Agent a local concurrently accessible FIFO (first-in-first-out) queue named *AgentEngine.msgInQueue* is used as a message mailbox. The *MsgInQueue* process passes messages from the AgentEngine.msgInQueue to the *msgInProcessQueue* executor service. Executors are discussed in Section 3.3.2, *Executor Class*. The executor service spawns[1] a *msgRoute* thread for each incoming message. The msgRoute thread determines the delivery or execution path of the incoming message. Once the message has been delivered the execution thread is terminated.

[1]The number of concurrent threads is configurable. The default value is four threads.

The following variables are used by the msgRoute thread to determine message destination:

- *AgentEngine.region*: Region name of the routing agent.
- *AgentEngine.agent*: Agent name of the routing agent.
- *src_region*: Source region name of incoming message.
- *src_agent*: Source agent name of incoming message.
- *src_plugin*: Source plugin name of incoming message.
- *dst_region*: Destination region name of incoming message.
- *dst_agent*: Destination agent name of incoming message.
- *dst_plugin*: Destination plugin name of incoming message.

The pseudocode shown in Algorithm 1 generates *routeString*, a 6 bit[2] boolean expression $[RX_r \wedge RX_a \wedge RX_p \wedge TX_r \wedge TX_a \wedge TX_p]$ representing a contextual message type. RouteString and the resulting *routePath* integer representation values are based on source and destination message parameters in respect to agents settings. There are three possible actions for the routing engine to take for each message:

- *drop*: Discard the message.
- *getCommandExec*: Execute message on the agent.
- *sendToPlugin*: Send message to a plugin on the agent.

The route path truth table is shown in Table 3.1, where F_0 and F_1 represent *getCommandExec* and *sendToPlugin* respectively. The default route action is to drop messages, which are omitted from the table.

[2]An arbitrary number of bits (flags) can be used in the future if needed.

Algorithm 1 Determine RoutePath

Input: src_region, src_agent, src_plugin, dst_region, dst_agent, dst_plugin
Output: routePath = [Base 10 value of route code]

if $src\_region = AgentEngine.region$ **then**
  $RX_r \leftarrow 1$
else
  $RX_r \leftarrow 0$
end if
if $src\_agent = AgentEngine.agent$ **then**
  $RX_r \leftarrow 1$
else
  $RX_r \leftarrow 0$
end if
if $src\_plugin \neq null$ **then**
  $RX_r \leftarrow 1$
else
  $RX_r \leftarrow 0$
end if
if $dst\_region = AgentEngine.region$ **then**
  $TX_r \leftarrow 1$
else
  $TX_r \leftarrow 0$
end if
if $dst\_agent = AgentEngine.agent$ **then**
  $TX_r \leftarrow 1$
else
  $TX_r \leftarrow 0$
end if
if $dst\_plugin \neq null$ **then**
  $TX_r \leftarrow 1$
else
  $TX_r \leftarrow 0$
end if
$routeString \leftarrow RX_r + RX_a + RX_p + TX_r + TX_a + TX_p$
{routeString: a string concatenation representing a 6 bit binary value.}
$routePath \leftarrow Integer.parseInt(routeString, Base_2)$
{routePath: an integer representation of incoming message type.}

Table 3.1: Agent Route Table

RoutePath	RX_r	RX_a	RX_p	TX_r	TX_a	TX_p	F_0	F_1
56	1	1	1	0	0	0	1	0
57	1	1	1	0	0	1	1	0
58	1	1	1	0	1	0	0	1
59	1	1	1	0	1	1	0	1
60	1	1	1	1	0	0	1	0
61	1	1	1	1	0	1	1	0
62	1	1	1	1	1	0	0	1
63	1	1	1	1	1	1	0	1

The boolean expressions for our two actionable route cases are shown below:

- *getCommandExec*: $RX_r \wedge RX_a \wedge RX_p \wedge TX_r \wedge TX_a \wedge \overline{TX_p}$
- *sendToPlugin*: $RX_r \wedge RX_a \wedge RX_p \wedge TX_r \wedge TX_a \wedge TX_p$

It not necessary to calculate boolean expressions for simple routing rules, like those used by the agent. In the case of agent routing, conditional expressions applied to source and destination addresses are sufficient to implement route procedures. However, as we will show in Section 3.6.4, *Controller Message Routing*, for routing cases with more variables developing complex conditional statements becomes difficult and are error prone. For complex cases routePath values are used with lookup tables to determine route actions.

The *MsgRoute* workflow is shown in Figure 3.4.

Messages are either routed to a specific plugin or executed on the agent itself. The route thread terminates on route completion, which is sufficient for unidirectional messages. However, there are cases where we want a response to our messages. Typically, method execution is a blocking function, where the calling instance must wait (is blocked) for a method to complete before continuing its processing. This mode of operation is considered a synchronous operation. In a distributed system

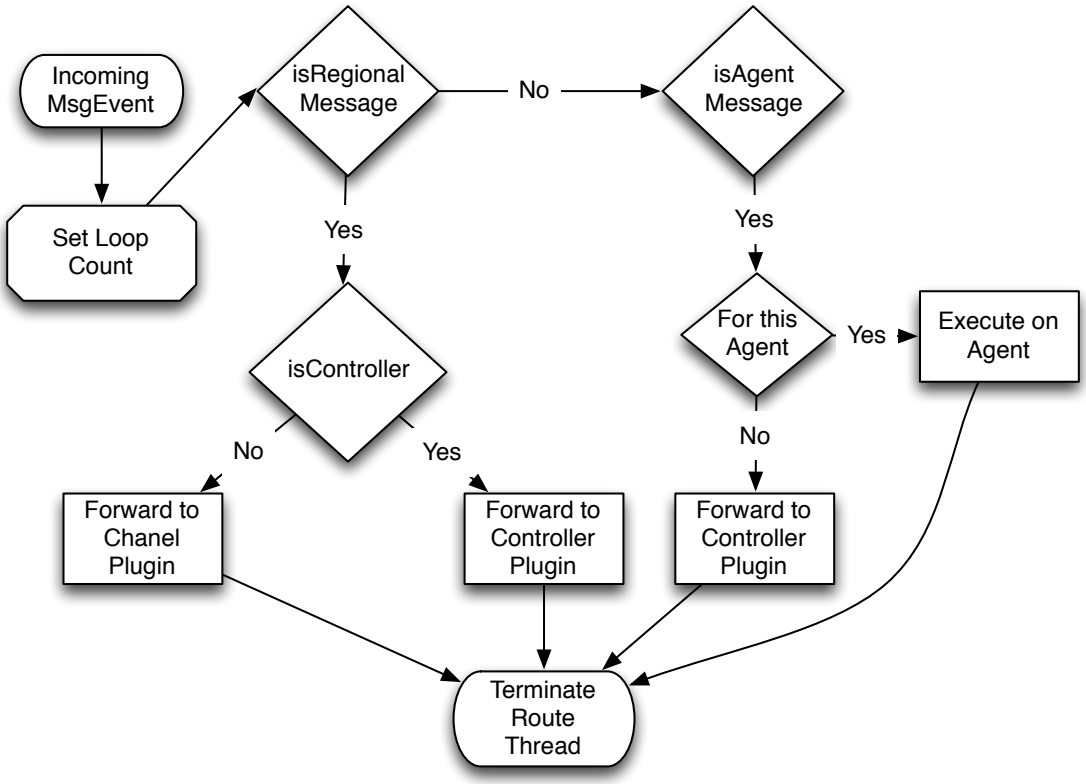

Figure 3.4: MsgEvent Routing process for the Cresco-Agent

with many agents and plugins we do not want to block the execution path, while waiting on methods to complete. The ability to call methods without waiting for the method to complete is considered an asynchronous (non-blocking) operation. In the context of the Cresco framework we define Remote Procedure Calls (RPC) as bi-directional asynchronous method executions. These calls can be performed between all Cresco components. RPC calls in the Cresco framework is explained in detail in Section 3.5.4, *Remote Procedure Call*.

As previously mentioned, MsgEvent messages destined for agents are routed to the getCommandExec process. The getCommandExec process implements the Cresco Agent API described in the next subsection.

3.2.4 Agent API

The Cresco Agent API provides functions used in the configuration and execution of plugin management functions. API functions are executed through CONFIG and EXEC MsgEvent type messages destined for the the agent. Currently implemented API functions are described below.

CONFIG

- *comm_init*: Function used to set the operating role state as determined by the a local Controller Plugin or delegated by a remote Regional Controller.
- *plugin_add*: Used to add plugin configurations to the agents local Cresco-Agent-Plugins.ini file and start a plugin.
- *plugin_remove*: Used to stop a plugin and remove plugin configurations from the agents local Cresco-Agent-Plugins.ini file.
- *plugin_inventory*: Returns the status and configuration for all plugins maintained by the agent.
- *plugin_download*: Used to transfer plugins files from external HTTPS servers as specified by URI.

EXEC

- *plugin_enable*: Start a plugin described in the agents local Cresco-Agent-Plugins.ini file.
- *plugin_disable*: Shutdown a plugin described in the agents local Cresco-Agent-Plugins.ini file.
- *show_version*: Provides runtime information, such as software version and function capabilities pertaining to the agent.

- *show_address*: Returns network interface and address information pertaining to the agents host.

In the next section, we will described the implementation of Cresco Plugins.

3.3 Cresco-Plugins

As with Cresco-Agents, Cresco-Plugins function as actors. However, with the exception of the Cresco-Controller-Plugin described in Section 3.6, *Cresco-Controller-Plugin*, plugins focus primary on actor functions outside of the operation of the Cresco framework. Cresco plugins provide communication channels for message passing, interfaces to resources, methods for information gathering, and control environments for processes running within the plugins themselves. We designate Cresco-Plugins that either natively host or provide interfaces to resources as *Resource plugins*. These plugins report resource information such as operational KPI, state, and capabilities to the host agents, which propagate plugin information to controllers. The same resource plugin implementation deployed on two agents can provide very different performance information for the same observed workload, based on variations (load, infrastructure, etc.) in the underlying operating environment. Plugins not only provide configuration and operational controls for resources, they also provide information used in global resource scheduling.

In Section 3.2.2, *Plugin Loading*, we described the process followed by the agent to load plugins. In order for plugin loading to work, a common interface must be implemented across all plugins. Required abstract classes used in the agent-plugin interface and common operational tasks are described in Section 3.4, *Cresco Plugin Library*. The Cresco Plugin Library not only provides a common agent-plugin interface, it greatly simplifies the process of developing Cresco Plugins. The three primary user-defined classes found in plugins are listed below:

- *Plugin class*: required main plugin class extending the *CPlugin* class found in

the Cresco Plugin Library.

- *Executor class*: optional class extending the *CExecutor* class found in the Cresco Plugin Library. This class is used to interface incoming messages with user-defined operational functions.

- *perfMonitor class*: optional class used to communicate KPI (local performance) messages to controllers.

The following sections describe the three primary user-defined functions found in Cresco Plugins.

3.3.1 Plugin class

Source code for an example Plugin class is shown below in Listing 3.6.

Listing 3.6: Plugin.java

```
1  import com.researchworx.cresco.library.plugin.core.CPlugin;
2
3  @AutoService(CPlugin.class)
4  public class Plugin extends CPlugin {
5      private PerfMonitor perfMonitor;
6
7      public void start() {
8          setExec(new Executor(this));
9
10         perfMonitor = new PerfMonitor(this);
11         perfMonitor.start();
12         logger.info("PerfMonitor initialized");
13     }
14
15     @Override
16     public void cleanUp() {
17         perfMonitor.stop();
18     }
19 }
```

The Plugin class is used as a main class for user-defined object creation, instrumentation, and destruction. In the actor-model paradigm, the *start()* function provisions methods that allow the plugin actor to make decisions. Starting on line 7 of the code

an example the *start()* function is shown, which sets the user-defined Executor class, creates the perfMontor object, and starts the perfMonitor services. On line 17 an optional *cleanUp()* function is implemented to gracefully stop the PerfMonitor service on plugin unload. Access to services instantiated in the Plugin class are accessed by agents and plugins through the Executor class described below.

3.3.2 Executor class

As previously mentioned, the only method of communication between agents and plugins is through message passing. In order to control the functions of plugins, an *Executor* class must be implemented to interface used-defined services in the main Plugin and derivative classes. In the actor-model the *Executor* class determines how the plugin actor responds to messages. An example executor class is shown in the Listing 3.7.

Listing 3.7: Cresco Plugin Executor

```
1  import com.researchworx.cresco.library.messaging.MsgEvent;
2  import com.researchworx.cresco.library.plugin.core.CExecutor;
3  import com.researchworx.cresco.library.utilities.CLogger;
4
5  public class Executor extends CExecutor {
6      private final CLogger logger;
7
8
9      public Executor (Plugin plugin) {
10         super(plugin);
11         this.logger = new CLogger(plugin.getMsgOutQueue(), plugin
               .getRegion(), plugin.getAgent(), plugin.getPluginID(),
                CLogger.Level.Info);
12         }
13
14     @Override
15     public MsgEvent processExec(MsgEvent msg) {
16         msg.setParam("sys-info", SysInfoBuilder.getInfo());
17         return msg;
18         }
```

In the plugin executor example, a new parameter *"sys-info"* is added (line 16) to every incoming message and the message is returned (line 17) to the sender. The

value returned by the "sys-info" key is determined by classes exposed by the Plugin class.

3.3.3 perfMonitor class

PerfMonitor classes can be very simple, reporting a single metric such as the number of active network flows, concurrent processes, or resource utilization. PerfMonitor classes can also be very complex providing low-level infrastructure measurements such as current environment state in relation to application-specific parameters. In addition, perfMonitor classes are used to monitor and measure KPI indicators between actors in a globally managed environment. For instance, suppose agent A and B with associated plugins A.1 and B.1 are running in environments that from a provisioning prospective are equivalent. However, the KPI(s) associated with the functions of plugin B.1 show higher performance than that of plugin A.1. In this case, perfMonitor data indicates that we should migrate processing from agent A's environment to agent B's. Conversely, in a more complicated case, where plugins must interact with geographically distributed agents as part of a process pipeline, there could exist measured communication performance benefits of agent A's environment that exceed general process advantages of alternative environments.

An abridged example of a perfMonitor class is shown in Listing 3.8.

Listing 3.8: Cresco Plugin perfMonitor

```
1  import com.researchworx.cresco.library.messaging.MsgEvent;
2  import com.researchworx.cresco.library.plugin.core.CPlugin;
3
4  class PerfMonitor {
5      private CPlugin plugin;
6      private SysInfoBuilder builder;
7
8  ...
9
10 public void run() {
11         MsgEvent tick = new MsgEvent(MsgEvent.Type.KPI, plugin.
               getRegion(), plugin.getAgent(), plugin.getPluginID(),"
```

```
            Performance Monitoring tick.");
12          tick.setParam("src_region", plugin.getRegion());
13          tick.setParam("src_agent", plugin.getAgent());
14          tick.setParam("src_plugin", plugin.getPluginID());
15          tick.setParam("dst_region", plugin.getRegion());
16          tick.setParam("resource_id","sysinfo_resource");
17          tick.setParam("inode_id","sysinfo_inode");
18
19           for(Map.Entry<String, String> entry : builder.
                getSysInfoMap().entrySet()) {
20                  tick.setParam(entry.getKey(), entry.getValue());
21              }
22           plugin.sendMsgEvent(tick);
23          } }
```

In the example perfMonitor class system-level parameters obtained from the SysInfoBuilder object are packaged into a KPI message and sent to a regional controller for additional processing and global resource propagation.

The described Cresco Plugin classes can be used to develop a wide-wide range of resource plugins. A number of plugins have been implemented for the Cresco Framework. Resource plugins are described in Section 3.7, *Plugin Implementations*.

Underlying abstract function used in initialization, message handling, and WATCHDOG generation are abstracted by the Cresco Plugin Library, which in turn used the Cresco Library for lower-level functions such as message handling.

In the next section we describe the implementation of the Cresco Plugin Library.

3.4 Cresco Plugin Library

The Cresco Plugin Library provides common functions for Cresco-Plugins, such as initialization, shutdown, and interfaces to core functions provided by the Cresco Library. The Cresco Library uses Apache Maven [96] for package distribution, which provides build, reporting and documentation from a central object configuration. The Cresco Library is available [97] from a public Maven repository. Low-level implementation

details are found in the Cresco Plugin Library code repository [98].

In the following sections we describe the underlying functions provided by the plugin library.

3.4.1 Plugin Interface

Plugins are units of compilation separate from agents, with respect to dependencies and build environments. Plugins are loaded by agents and reside in the memory space of the hosting agent. In order for plugins and agents to communicate, a common *Plugin Interface* is implemented. Plugin Interfaces are common code that describe the methods implemented by the plugin, which are accessible to the agent.

On plugin initialization the following objects are passed from the hosting agent to the plugin:

- *msgOutQueue*: are concurrent queue used to offer messages back to the agent.
- *configObj*: is an object representation of the this plugins configuration found in *Cresco-Agent-Plugins.ini*.
- *region*: the region name of the hosting agent.
- *agent*: the agent name of the hosting agent.
- *pluginID*: the plugin slot number of the hosting agent.

As part of the initialization process an incoming message processor (*msgIn()*), RPC handler, WatchDog, and logging services are started. In addition, any *preStart()*, *start()*, and *postStart()* functions implemented in the dependent plugin are executed. If all core and user-defined function complete without error the plugin is marked active and considered an actor in the Cresco Framework.

Figure 3.5 show the data-path interface between agents and plugins.

Additional functions provided by the Cresco Plugin Library are listed below:

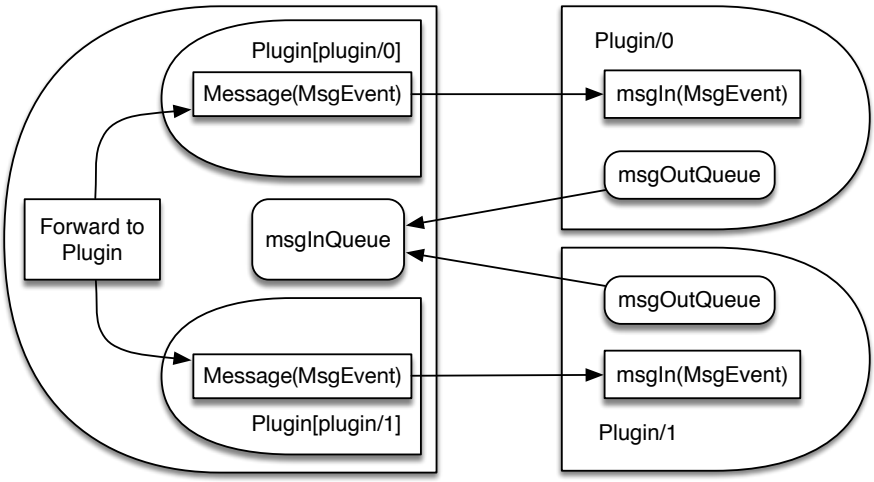

Figure 3.5: Cresco-Agent to Cresco-Agent-Plugin Interface

- *get/setName*: Function to get and set the name of the plugin.
- *get/setVersion*: Function to get and set version information of the plugin.
- *get/setRegion*: Function to get and set the region name of a plugin.
- *get/setAgent*: Function to get and set the agent name of a plugin.
- *get/setPluginID*: Function to get and set pluginID of a plugin.
- *sendMsgEvent*: Function to send MsgEvent messages from the plugin asynchronously.

There are a number of additional functions, such as those dealing with RPC communication, configuration management, and WatchDog services, provided by the plugin library that wrap underlying calls provided by the Cresco Library into the Cresco Plugin Library class. The Cresco Library implementation is described in the next section.

3.5 Cresco Library

The Cresco Library provides common core functions shared by *Cresco-Agent* and *Cresco-Plugins*, such as configuration management, health monitoring, message processing, logging, and other functionality. Providing core functions as part of a central library enforces standardization across components, while making development of new components easier. The Cresco Plugin Library uses Apache Maven for package distribution and is available [99] from a public repository. Low-level implementation details are found in the Cresco Plugin Library code repository [100].

The following subsections describe core functions provided by the Cresco Library.

3.5.1 Configuration Management

Plugin configuration are maintained for agents and plugins as part of the *Config* class. Configurations are composed of key-value pairs. Modification of runtime configuration parameters of agents (*Cresco-Agent.ini*) and plugins (*Cresco-Agent-Plugins.ini* are reflected in their respective configuration files. The Config class provides the following functions to convert configuration parameters into Java primitives and classes.

- *getBoolean*: returns requested boolean primitive from string configuration value.
- *getDouble*: returns requested double primitive (double-precision 64-bit IEEE 754 floating point) from string configuration value.
- *getInteger*: returns requested integer primitive (two's complement range from -2^{31} to $2^{31}-1$) from string configuration value.
- *getLong*: returns requested long primitive (two's complement range from -2^{63} to $2^{63}-1$) from string configuration value.
- *getString*: returns requested string object from a array of characters.

If the requested configuration file does not exist, a *null* value is returned. Additional methods exist for primitive conversion, which allow the user to supply a return value if the configuration key does not exist.

3.5.2 Health Monitoring

Low-level health monitoring is accomplished through the use of WATCHDOG messages communicated throughout the Cresco component hierarchy. The Cresco Library implements a *WatchDog* class, which is programmatically similar to the perfMonitor class shown in Listing 3.8. The WatchDog class communicates WATCHDOG messages that contain information about the Cresco component, including *runtime* and *timestamp* information (in milliseconds). Plugins communicate messages to agents, agents to regional controllers, and regional controllers to global controllers. Based on a user-defined timeout, Cresco components are considered detached from their hierarchy when downstream messages are not received. Likewise, if agents detect that they are unable to communicate WATCHDOG messages, and are not configured for static operation, a rediscovery process is triggered as described in Section 3.6.3, *Controller Discovery*.

3.5.3 Message Tools

The Cresco Library provides implementation classes for MsgEvents as described in Section 3.1, *MsgEvent Protocol*. Classes related to MsgEvent creation, object marshallings, and communication are provided.

Cresco, as with other actor frameworks, has been designed to support asynchronous messaging (message without response) between component actors. Synchronous communication (messages with expected response) is accomplished through the use of RPC described below.

3.5.4 Remote Procedure Call (RPC)

In princable synchronous messaging violates the primary tenants of Actor-model concurrency. Allowing synchronous messaging creates the potential for dead-locks created through circular message dependencies. The Remote Procedure Call (RPC) class is used to implement synchronous operations using asynchronous messages, with controls to prevent message dead-locks.

RPC messages are used in cases where a response from a remote component is needed to complete a transaction. RPC messages must be identified in the payload of the message. If a plugin or agent sends an RPC message, that message must contain a *callid* parameter in the message body. The *callid* parameters must conform to the following format: *callId-[src_region]-[src_agent]-[src_plugin]-[unique random string].* A *unique* identifier must prepend the callid, since this references a single unique message. However, the unique identifier must only be unique for the duration of the message and can be reused once the calling thread has terminated. In the case of a RPC call originating by an agent, the *[src_plugin]* field is omitted. The *callid* parameters are inserted by the sender of an RPC message allowing agents and plugins to uniquely identify and track RPC messages. The RPC *call* function spawns a new thread, which adds appropriate callId information to the message. The message is sent to an outgoing queue for routing and the thread will blocked until a return message is received or the default RPC timeout of 30 seconds is exceeded. If RPC timeout is exceeded the calling function is canceled and an error message is returned, preventing component dependency locks.

3.5.5 Logging

The Cresco Library provides unified logging services across Cresco components. The *CLogger* class implements message and file-based logging services with *error*, *warn*, *info*, *debug*, *trace* designations. Message-based logs are created with the *LOG* Ms-

gEvent type and file-based logs are stored on the host agent environment path as defined in the Agent.ini.

3.6 Cresco-Controller-Plugin

The Cresco framework can be described as a hierarchical arrangements of actors, implemented as message-passing agents and plugins. In Section 3.2, *Cresco Agent*, we defined that the primary purpose of the agent is to host and communicate messages between plugins. Likewise, in Section 3.3, *Cresco-Plugins*, we defined the primary purpose of plugins as a provider monitoring, measurement, and control of resources in the Cresco Framework. The Cresco-Controller-Plugin is a special plugin that governs the operational aspects of the Cresco framework by providing the following:

- *Discovery*: The controller provides network-based discovery services, which are used to determine operating roles within Cresco topologies.
- *Message Broker*: The controller provides message brokering services, which are used for inter-agent and plugin communications.
- *State Memory*: Depending on operating mode, the controller maintains agent, region, and global state information.
- *Services*: The controller provides decision making capabilities, such as resource scheduling, optimization, and other service delivery functions on agent, region, and global-levels.

In the remainder of this section we will cover the operational aspects of the Cresco-Controller-Plugin.

3.6.1 Controller Initialization

On agent initialization, the controller is the first plugin that is initialized. If the controller plugin can not be loaded the agent startup, process is terminated. The

Cresco-Controller-Plugin configuration is maintained in the Cresco-Agent-Plugins.ini file as with other plugins. An example controller configuration is shown in Listing 3.9.

Listing 3.9: ControllerConfiguration

```
1  [plugins]
2  plugin/0 = 0
3
4  [plugin/0]
5  pluginname = cresco-agent-controller-plugin
6  jarfile = cresco-agent-controller-plugin-0.5.0-SNAPSHOT.jar
7  #Secret key used in agent controller discovery
8  discovery_secret_agent = cresco_agent_discovery_secret
9  #Secret key used in regional discovery
10 discovery_secret_region = cresco_regional_discovery_secret
11 #Secret key used in global discovery
12 discovery_secret_global = cresco_global_discovery_secret
13 watchdogtimer = 5000
14 #Enable ssh daemon on the controller
15 enable_sshd = true
16 sshd_username = admin
17 sshd_password = admin
18 sshd_rsa_key_path = sshd.key
19 #Enable IPv6 capabilities
20 isIPv6 = true
```

In the controller plugin configuration *discovery_secret* configuration parameters are used by the discovery process to determine the *operating mode* of the controller and related agent. The discovery process is described in Section 3.6.3, *Controller Discovery*. The controller plugin initialization process is shown in Figure 3.6. Controller operating modes are described in Section 3.6.2, *Controller Modes of Operation*.

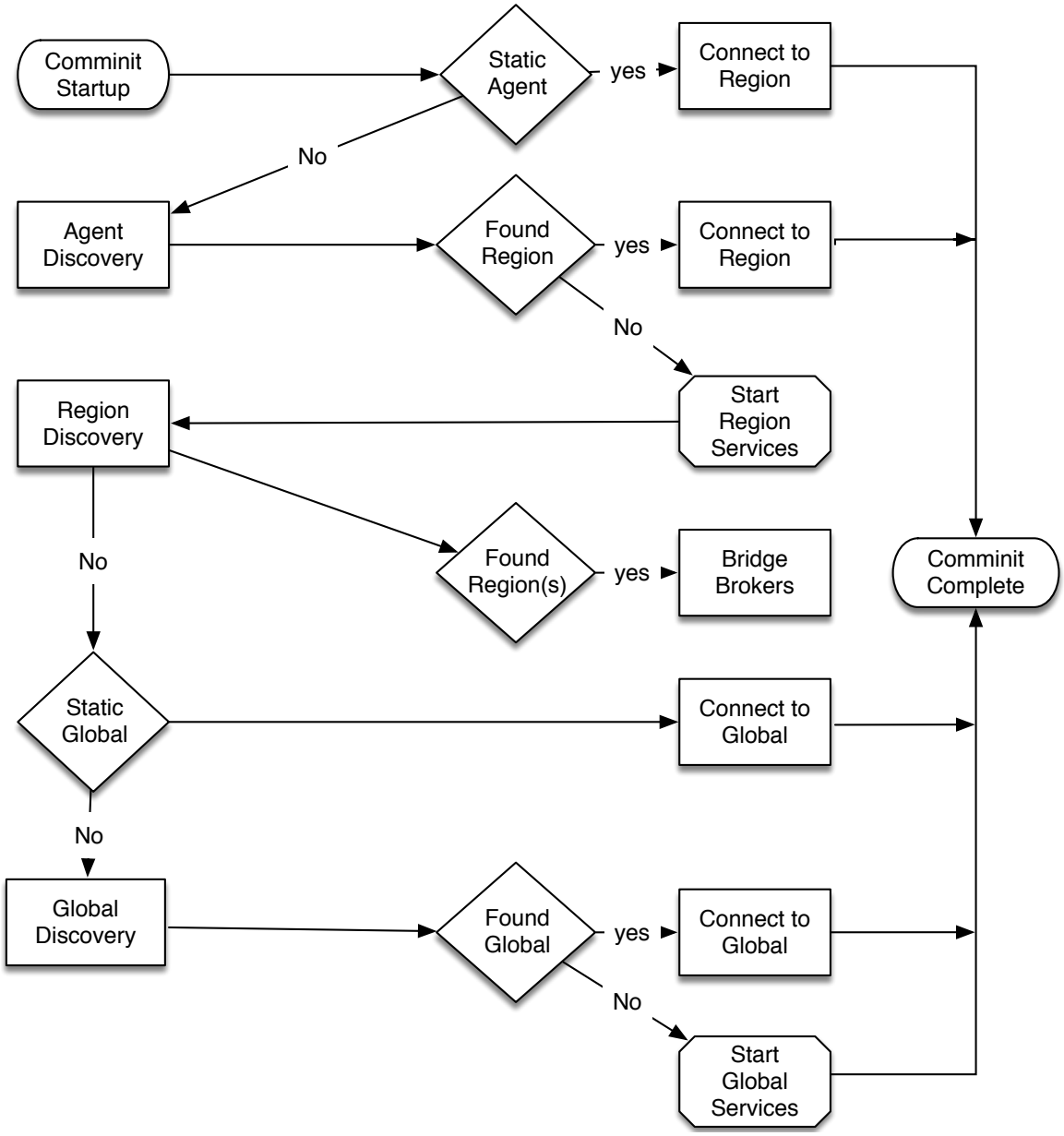

Figure 3.6: Initialization of the Cresco Controller Plugin

In the next section we describe the modes of controller and related agent operations.

3.6.2 Controller Modes of Operation

An *operating mode* defines where a controller resides in the Cresco hierarchy. There are three operating modes of a controller, as stated below:

- *Agent*: Responsible for communication with regional controller.
- *Regional*: Responsible for communication between regions and with one or more global controllers as well as intra-regional message routing.
- *Global*: Responsible for establishment of a global management plane across and inter-region message routing.

An example of a topology graph showing the Cresco hierarchy is shown in Figure 5.15.

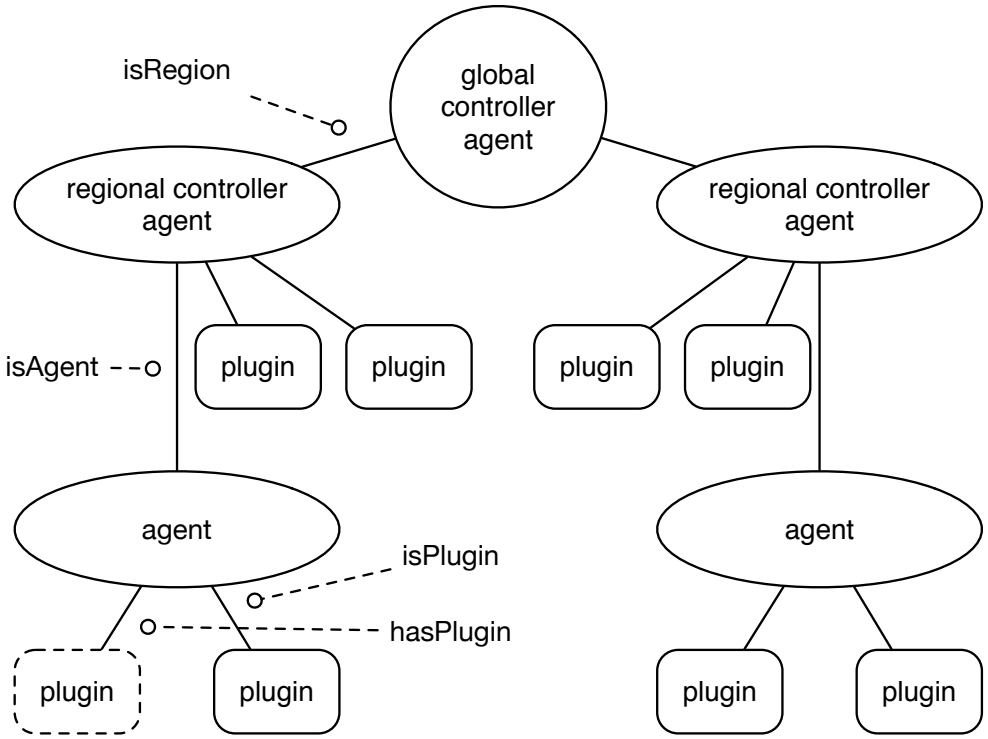

Figure 3.7: Simple Cresco Topology Graph with two regions

Agent Controllers The agent controller manages inter-agent communications to the regional controller. Agent controllers are dependent on regional controllers for all intra-node communications.

Regional Controllers The regional controller plugin manages reachable agents in its region. This plugin serves as a gateway connecting reachable intra-regional components and inter-regional components. Agent's status is detected by the regional controller plugin as changes on the agent-level are discovered. However, it is not necessary for the agent to maintain connectivity to a regional controller at all times. The system is designed with the expectation that agents and plugins can both appear and disappear without warning, so state discovery is accomplished using several methods described in the following subsection.

Global Controllers The global controller establishes a global view of resource status across regions. This view is inclusive of all resources assigned at regional and agent levels. A global resource view is established by maintaining a graph database of all known local and regional relationships. Currently, the open-source graph database OrientDB [101] is used for database services. We define all nodes and edges in our graph database to be the complete Cresco data graph. We define the *topology graph* to be a sub-graph of the complete graph, which contains nodes, edges, and labels related to arrangement and connectivity of Cresco components. We refer to arrangement in this context to be the spacial positioning of Cresco components based on regional, agent, and plugin assignments. We define the connectivity of Cresco components in this context to be the path in which MsgEvent messages are routed. The topology graph is maintained by the global controller, based on regional topology information provided by regional controllers.

We define the *resource graph* to be a sub-graph of the complete graph of the system, which contains nodes, edges, and labels related to the arrangement, configurations, and utilization of resources managed by Cresco components. In this context, configuration is related to the textual configuration of agents and plugins as described in Section 3.2, *Cresco Agent*. Utilization is defined as a set of resource-specific metrics, reported by resource managing Cresco plugins.

An example of a resource graph is shown in Figure 3.8.

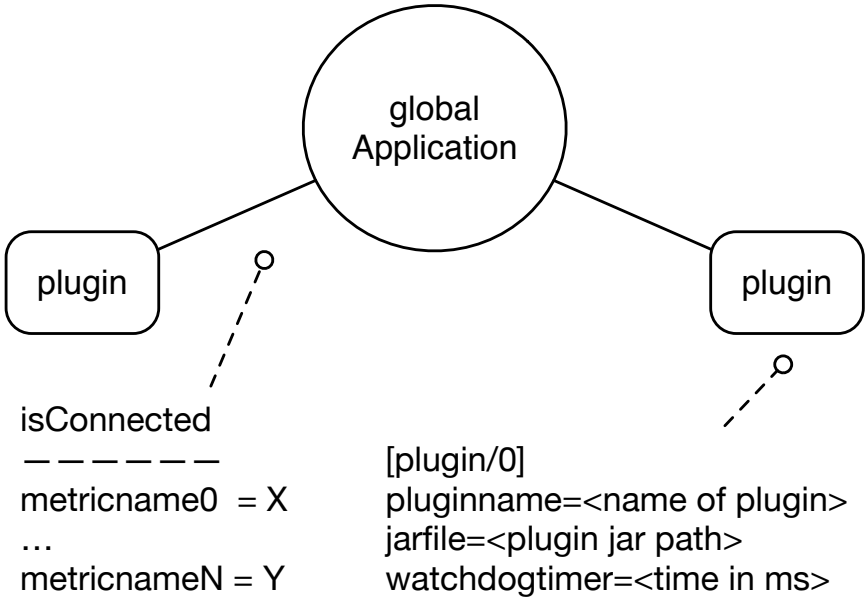

Figure 3.8: Simple Cresco Resource Graph with two assigned plugins.

The resource graph includes active plugins related to specific global applications. Information is provided to the global controller by one or more regional controllers. These plugins establish communication with the global controller and provide both regional and agent resource metrics for topology and resource graphs.

3.6.3 Controller Discovery

The controller discovery process is an important feature of the Cresco framework. The ability to statically assign a network of agents is important for applications where communication between hosts should be restricted to verified agents. Alternatively, dynamic discovery of agents, regions, and related controllers is important for operational resiliency and scale. Agents and related resources assigned to specific locations or regions should be discoverable without notification of resource creation or assignment. In this section we discuss the discovery configuration, order of operations,

process.

Discovery Modes Controller discovery takes place in the order of *AGENT*, *REGION*, and *GLOBAL*. Discovery can be static, dynamic, or a hybrid of the two modes. Discovery modes of operation are listed below. Additional methods may be added in the future as needed:

- *Static*: All configurations related to the agent and plugins are provided on agent startup via configuration file. While regional and global controllers can be used in this mode, typically static assignments are made on agents to force assignments to specific regional controllers without going through the network discovery process. Static modes of operation based on configuration are listed below:
  - *Agent-forced*: In this mode of operation the *regional_controller_name* value is provided in the agents configuration file. The agent will continue to attempt unicast network discovery for the requested region until the agent process is terminated.
  - *Region-forced*: In this mode of operation the *is_regional_controller* value is set to *TRUE* and no regional network discovery takes place. The agent assigns itself the regional name provided in the *regional_controller_name* configuration. In this mode, dynamic global discovery does not take place and unicast network discovery for the global controller specified by *global_controller_ip* and *global_controller_name* is attempted.
  - *Global-forced*: In this mode of operation the controller is already a regional controller, either through static or dynamic discovery and unicast network discovery for the global controller specified by *global_controller_ip* and *global_controller_name* is attempted.
- *Dynamic*: Agent operation is determined through the network discovery process

and component configuration can change as need in the future based on interactions with other agents and controllers. For instance, if a dynamic agent looses contact with its regional controller an agent may promote itself to a regional controller. Dynamic modes of operation based on discovery are listed below:

- *Agent-only*: Connectivity with a regional controller has been established. Regional and global communications will take place through the established regional controller broker. Agent and Plugin status is communicated to the regional controller. No further regional or global controller discovery will take place, unless connectivity with the regional controller is lost. If regional controller connectivity is lost, the dynamic discovery process is repeated.
- *Agent-Region*: Agent (discovery) connectivity with an existing regional controller could not be established and the controller has been promoted to a regional controller. Communications between agents in this region take place on this regional controller. Regional discovery can result in regional peering[3] between two regions. If regional peering has occurred regional messaging will take place between regional controllers. Connectivity with a global controller has been established. Regional status is communicated to the global controller. If global connectivity is lost the global discovery will be restarted, but existing agent and regional settings will remain unchanged. If no global controller is found the regional controller is promoted to a global controller.
- *Agent-Region-Global*: Agent and regional connectivity with existing controllers could not be established and the controller has been promoted to a global controller. Plugin, agent, and regional status will be reported to this global controller.

[3]Peering in this context, much like network peering is the sharing of reachable agents between brokers.

Controller discovery takes place in two phases. The first phase is network identification, which is used to identify potential communication brokers for connection and the second is the authenticated connection to the secure message broker.

Network Identification By default network discovery supports both IPv4 and IPv6 protocols. IPv4 discovery is accomplished through network unicast[4] and broadcast[5] [102]. IPv6 discovery is accomplished through network multicast [103]. Both discovery methods can be used simultaneously in "dual stack" IPv4 and IPv6 operation.

In network discovery a MsgEvent message is transmitted to potential hosts using one or more supported protocols and communication methods. The message contains the *discovery_type* parameter, which specifies either *AGENT*, *REGION*, or *GLOBAL* discovery. In addition, the message contains the *discovery_validator* parameter, which contains an AES-encrypted [104] discovery message. The discovery message is encrypted with a shared key configured in *discovery_secret_agent*, *discovery_secret_region*, or *discovery_secret_global* depending on discovery type. Existing controllers listen for discovery messages and if they are able to decrypt the discovery message and are otherwise not restricted[6] respond to discovery request with the *agent_count* and *validated_authenication* parameters. The *validated_authenication* parameter provides unique credentials that can be used to connect with a communication broker. The discovering controller might receive a number of responses and will choose the least loaded controller for connection based on *agent_count* values.

Broker Communication Following the network identification phase, in which secure connection credentials are generated for each discovered session, the controller will establish communication with one or more brokers. The open-source package ActiveMQ [105] is currently used to provide brokered communication services between

[4]Communication addressed to a specific host.
[5]Communication directed to many possible hosts.
[6]The *maximum_agents* parameter restricts the number of connected agents.

controllers and potentially other plugins.

By default, ActiveMQ supports automated (AUTO) [106] detection for protocols used in the Cresco framework communication. Natively, ActiveMQ supports channels using MQ telemetry transport (MQTT[7] [91]), advanced message queuing protocol (AMQP) [107], OpenWire [108], representational state transfer (REST) [109], RSS and Atom [110], streaming text oriented messaging protocol (STOMP) [111], web services invocation framework (WSIF) [112], WebSocket Notifications [113], and extensible message and presence protocol (XMPP) [114] natively. Regardless of the transport protocol, the text-based MsgEvent format is used. The MsgEvent protocol allows for the heterogeneous operation of the Cresco framework across a broad variety of communication channels.

The communication broker's modes of operation are listed below.

- *Agent-to-Region*: A controller functioning in an agent-only mode. The controller creates a communication channel that is maintained by a regional controller broker. The regional controller maintains a list of agent-only communication channels and routes messages between them.

- *Region-to-Region*: A controller that maintains its own broker for Agent-to-Region communications. In addition, methods to peer (bridge) its broker with another regional controller broker are provided. Broker peering provides the ability of one broker to see the clients on another broker, which allows for the routing of messages directly between connected regions. Once a broker bridge is established, agent-only communications channels maintained by regions are shared between brokers, allowing for region-to-region communications. Regional bridge health is monitored and if a bridge failure is detected, the bridge and related communication channels are removed.

- *Region-to-Global*: A special case of Region-to-Region communication where a

[7]MQTT is a widely adopted transport protocol in IoT applications.

broker bridge is used to connect a regional controller to a global controller. The global controller is identified by its regional controller communication channel on the connecting regional controller.

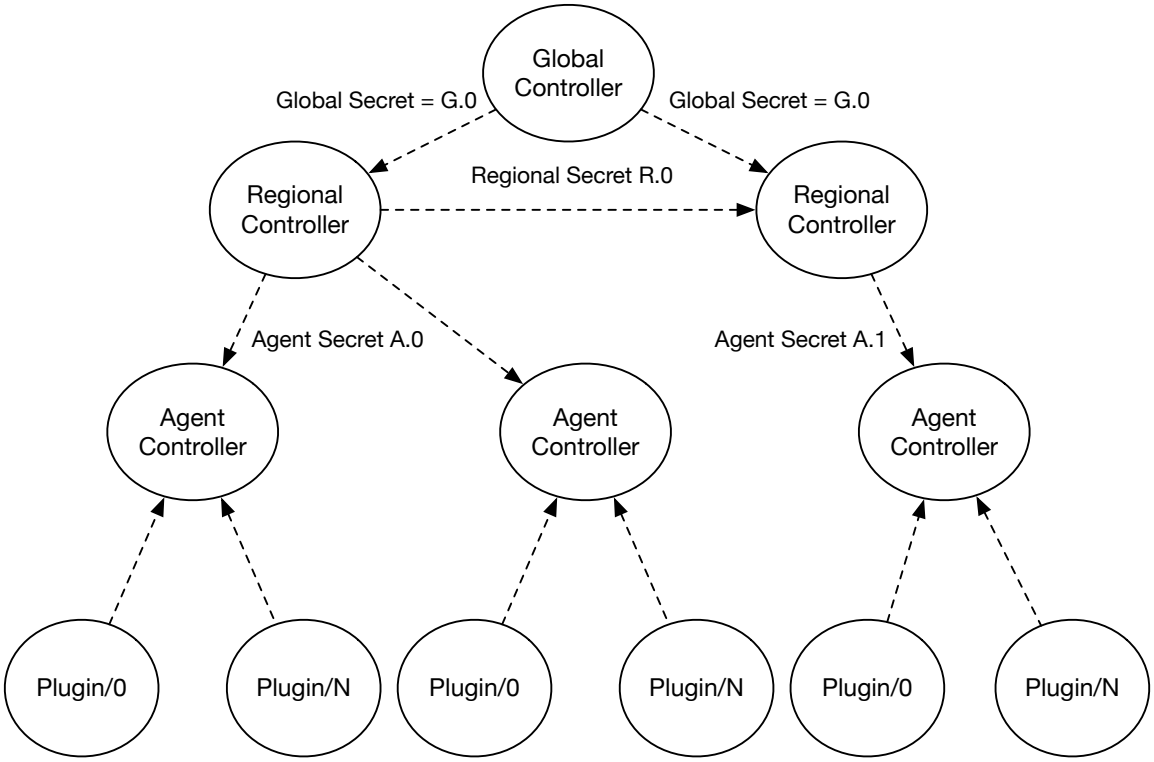

Figure 3.9: Secret-Key Managed Dynamic Discovery

Complex structures of networks of agents can be constructed through dynamic discovery managed by shared discovery secrets. Figure 3.9 shows an agent structure where two regions have been created dynamically through the use of differing agent discovery secrets in the same discovery domain. During the agent discovery process the existing regional controller is unable to decrypt and respond to the discovery message, so the agent promotes itself to a regional controller. The new regional controller is able to communicate with the old regional controller since the shared regional discovery secrets are the same. Likewise, both regions communicate to the same global controller since the global discovery secret is the same. In the example if we wanted to prevent Region-to-Region or Region-to-Global communications with

existing resources the shared secret could be changed. Using the shared secret method of dynamic discovery we are able to add large numbers of agents to existing networks with maintaining a desired network structure.

The discovery process described above allows us to both prescribe static agent relationships and create scalable dynamic networks of agents. In the next section we describe how brokered messages are routed between communication channels.

3.6.4 Controller Message Routing

From a procedural prospective, Controller Message Routing is the same as agent message routing, described in Section 3.2.3, *Agent Message Routing*. The same variables used in the msgRoute thread in Agent Message Routing are used in Controller Message Routing to determine message destination. However, the controller has additional message paths, which result in a different routing table from the agents. The following route destinations and related calling function are shown below:

- F_0: externalSend: Send to external location through the message broker.
- F_1: plugin.sendMsgEvent: Send message to the host agent
- F_2: getCommandExec: Submit message to the plugin executor.

MsgEvents are submitted to the routing engine through the *msgIn()* plugin function described in Section 3.4.1, *Plugin Interface*. The msgIn function passes messages to the *msgInProcessQueue* executor service. The executor service spawns a msgRoute thread for each incoming message. For each new message arrival a *MsgRoute* thread is created to route the incoming message.

In addition to messages arriving from the host agent, the controller routing engine must also route message related to agent, regional, and global destinations. The Cresco Controller Plugin provides inter-node agent communications.

Table 3.2: Controller Route Table

Route Code	RX_r	RX_a	RX_p	TX_r	TX_a	TX_p	F_0	F_1	F_2
0	0	0	0	0	0	0	0	1	0
21	0	1	0	0	1	0	1	0	0
48	1	1	0	0	0	0	1	0	0
52	1	1	0	1	0	0	1	0	0
53	1	1	0	1	0	1	1	0	0
56	1	1	1	0	0	0	0	1	0
58	1	1	1	0	1	0	0	0	1
62	1	1	1	1	1	0	0	0	1

The route path truth table is shown in Table 3.2. The default route action is to drop messages. For the sake of simplification we have omitted the dropped message case from the table.

The simplified boolean expressions for our three actionable route cases are shown below:

- *external*: $RX_a \wedge \overline{RX_p} \wedge TX_r \wedge \overline{TX_a} \wedge TX_p \vee RX_r \wedge RX_a \wedge RX_p \wedge \overline{TX_a} \wedge \overline{TX_p}$
- *sendMsgEvent*: $\overline{RX_r} \wedge \overline{RX_a} \wedge \overline{RX_p} \wedge \overline{TX_r} \wedge \overline{TX_a} \wedge \overline{TX_p} \vee$
  $RX_r \wedge RX_a \wedge RX_p \wedge \overline{TX_r} \wedge \overline{TX_a} \wedge \overline{TX_p}$
- *getCommandExec*: $RX_r \wedge RX_a \wedge RX_p \wedge TX_r \wedge TX_a \wedge \overline{TX_p}$

The boolean expressions for Agent Message Routing differed by a single boolean value and were simple enough to be implemented with conditional expressions. However, the boolean expressions representing Controller Message Routing are more complex. For Controller Message Routing we maintain a static route table, which is sufficient to route message to and from host agent, local plugin, and external sources and destinations.

3.6.5 Controller API

The Cresco Controller API provides functions used in the configuration and execution of controller management functions. API functions are executed through CONFIG

and EXEC MsgEvent type messages destined for the the agent. Currently implemented API functions are described below.

CONFIG

- *addplugin*: Function used to provision a Cresco plugin configuration for assignment on a specific agent.
- *removeplugin*: Function used to decommission a Cresco plugin configuration on a specific agent.
- *gpipelinesubmit*: Function used to submit a collection of Cresco plugin descriptions for assignment on a number of agents.
- *getgpipelinestatus*: Function used to get the status of a specific pipeline.
- *getgpipeline*: Function used to get the current configuration details of a specific pipeline.
- *getgpipelinelist*: Function used to get the list of current pipelines.
- *gpipelineremove*: Function used to decommission pipeline resources and remove a specific pipeline from the controller database.

EXEC

- *pluginupload*: Function used to upload plugins to the controller.
- *plugininventory*: Function used to list all plugins available on a controller.
- *plugininfo*: Function used to retrieve a description of a specific plugin available on a controller.
- *getpluginstatus*: Function used to retrieve the current status of a Cresco plugin.

- *getenvstatus*: Function used to retrieve the Cresco Topology managed by a controller.

- *resourceinventory*: Function used to retrieve the description of resources managed by a controller.

- *regionalimport*: Function used to import regional databases on global controllers.

The Controller API is accessible through Representational State Transfer (RESTful) methods. The Glashfish [115] embedded HTTPS web server implementation is used to provide RESTful services.

In the next section, we will describe the implementation of Cresco plugins.

3.7 Plugin Implementations

A number of Cresco Plugins have been implemented to provide resources or manage workloads. We describe a few notable plugins in this section.

3.7.1 Infrastructure as a Service (IaaS) Plugin

Infrastructure as a Service (IaaS) is a form of so-called cloud computing used to manage collections of compute, storage, and network components to provide virtual machine, and other low-level infrastructure resources. Typically, we designate IaaS offered publicly by a third-parties as public IaaS providers. IaaS services that are not available to the public, we refer to as private IaaS. IaaS platforms, such as OpenStack, allow users to dynamic provision and de-provision resources as needed programatically. For example, as the number of concurrent users increases for a specific application, an application management system can increase the number of application servers to distribute workloads. Likewise, as the number of application users decreases so can the number of application servers.

In edge applications, a number of agents will be statically configured on standalone servers located at the logical edges of networks. For example, a standalone server acting as a data collector for an array of distributed sensors might be configured with static plugins and controller configurations. However, the upstream data processing system might be configured for dynamic operation where plugin assignment are migrated between pools of regional resources. In order for regional resources to expand and contract as needed, we need automated methods to manage regional resources.

The IaaS Plugin is used to manage resources provided by IaaS cloud computing frameworks. We use the open-source Apache jClouds [116], software library to manage IaaS providers. Apache jClouds supports a number of public and private IaaS providers including Amazon Web Services, Microsoft Azure, Google Cloud, and OpenStack-based services.

Functions provided by the IaaS Plugin are used to provision and de-provision resources based on request from the Cresco schedulers.

3.7.2 System Information Plugin

The System Information Plugin is used to update regional and global controllers with the current operating configuration and resource utilization of systems hosting Cresco agents. We use the cross-platform Operating System and Hardware Information [117] library to access system information. Information such as operating system versions, CPU, memory, disk, and other resource configurations and related utilization are collected. Listing 3.10 shows an example report generated by the System Information Plugin.

Listing 3.10: SysInfo Data

```
1  --System Information
2  sys-uptime=12 days, 15:14:26
3  sys-os=GNU/Linux Debian GNU/Linux 8 (jessie)
4
5  --CPU Information
6  cpu-sn-ident=Intel64 Family 6 Model 69 Stepping 1
7  cpu-sn=unknown
8  cpu-summary=Intel(R) Core(TM) i7-4650U CPU @ 1.70GHz
9  cpu-core-count=2
10 cpu-ident=Intel64 Family 6 Model 69 Stepping 1
11
12 -CPU Utilization
13 cpu-per-cpu-load=CPU Load per processor: 8.1% 5.1%
14 cpu-user-load=6.09
15 cpu-nice-load=0.0
16 cpu-sys-load=0.5
17 cpu-idle-load=93
18
19 -Memory Utilization
20 memory-available=1249308672
21 memory-total=2094940160
22
23 -Storage Utilization
24 fs-map=0:/,1:shm
25 fs-0-available=57786044416
26 fs-0-total=63370678272
27 fs-1-available=67108864
28 fs-1-total=67108864
29
30 -Network Information
31 nic-map=0:eth0
32 nic-0-ip=fe80:0:0:0:42:acff:fe11:2%eth0,172.17.0.2
```

System information data is reported to regional and global controllers, where capacity and utilization information is employed for scheduling purposes. In addition, reported utilization information compared to KPI reports are used in resource value analysis.

3.7.3 Process Manager Plugin

The process manager plugin is used to launch, observe, and manipulate processes and schedulers on Unix-like operating systems, such as Linux. Processes are launched

using common shell parameters and observed through resulting process IDs. One way in which the scheduler is manipulated is through a processes *nice* [118] value. The nice value is set through a kernel-level call and corresponding user-level application. By default, processes have a *nice* level of 0, by changing this value the process is given more or less scheduling priority. The scheduler will satisfy request based on the lowest numerical priority p. So, setting the nice value to -20 would result in the highest scheduling priority, while 19 would represent the lowest priority. The share of resources allocated to a process based on nice value varies based on scheduler implementation. Along with the process *nice* settings, there are also *ionice* settings that specifically related to storage IO priority (I.e. file system devices).

This plugin can be used to run processes once, a designated number of iterations, or maintain a process indefinitely through repeated execution.

3.7.4 Container Manager Plugin

The container manager plugin is used to launch, observe, and manipulate application containers. We make use of Docker [119], a popular container management platform. Docker, or simply *containers*, are an operating-system-level virtualization technology. Using containers, specific application dependencies are bundled within the container so no central management of library and runtime dependencies is required. Containers are arranged in layers of dependencies, which are registered through a central container service. This service can be thought of as an "App Store," where specific dependency layers are uniquely registered. The layered registration of containers provides the ability of a layer, regardless of the number of containers that use the layer on a host, to only be stored once. For example, if 100 containers using the same Java dependency are run on a single host, only disk storage related to a single Java dependency layer will be consumed. When layers of dependencies are arranged to provide a new combined layer of dependency or application, the resulting container is called an *image*. Container images provide all system-level and application-level

dependencies required to run one or more applications. Each time a container layer is changed, a new registration is required for that image, so registered container images provide a unique reference related to specific application versions.

Containers can have specific resources restrictions, such as processor time, memory, and storage capacity, applied at the container-level. For example, suppose there are a number of containers on a host. One processor core can be exclusively assigned to a single container, group A, while the remaining processor cores can be assigned to the rest of the containers, group B. Within the group B containers a subgroup B.1, can be assigned a higher priority of processor time than the rest of the potential containers in group B. Likewise, specific memory and storage constraints can be applied per container or groups of containers. Figure 3.10 shows container resource assignments as described in the previous example.

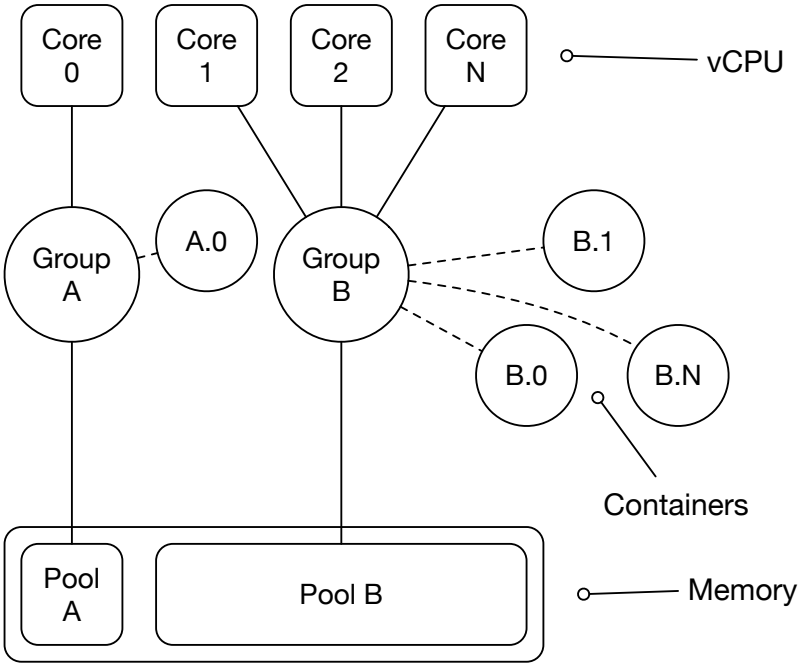

Figure 3.10: Container Resource Assignments

This Container Manager Plugin is used to retrieve a container from a registry and manage its operation, including resource settings.

3.7.5 AMPQ Plugin

The AMPQ Plugin utilizes the Advanced Message Queueing Protocol (AMPQ) [120] to provide communication. The RabbitMQ [121] AMPQ protocol implementation is being used. This plugin establishes authorizations, queues, and exchanges for Cresco-managed resources. For instance, the AMPQ Channel Plugin might be used to manage an intermediate communications channel between an otherwise inaccessable data source and destination.

In the next Chapter *Framework Technologies*, the Cresco approach to resource management, application description, and advanced operations modules are discussed.

4

Framework Technologies

The purpose of the work described in this dissertation is to establish an edge computing framework for the management of distributed applications and related resources. We have described the motivations, architecture model, and implementation details of the presented framework. In the previous chapter, we described a foundational platform to be used broadly in the configuration of edge components and the reporting of component-level metrics. Component-level information provided by the Cresco framework allows for higher-order functions, such as scheduling optimizations, application fault-tolerance, anomaly detection, predictions of future resource needs, and resource trading to be implemented for distributed applications management.

In Section 4.1, *Resource Management*, we define management techniques for underlying resources commonly used by the Cresco framework. In addition to understanding the operating state of underlying resources, in order for high-level functions to operate dynamically, they must also understand the context of how resources are related in distributed applications. In Section 4.2, *Cresco Resource Model*, we discuss how resources are described and managed within the Cresco framework. In Section 4.2.1, *Cresco Application Description Language*, we propose a language to describe resource relationships between components managed by the Cresco framework. In Section 4.3, *High-level Operations*, we describe high-level component-application

functions implemented by the Cresco framework.

4.1 Resource Management

There are potentially many layers of resource scheduling in edge computing environments. We define the following resource layers in the context of resource scheduling or provisioning. Resource layers are listed from the lowest (closest to the hardware) level of hardware abstraction, to the highest (closest to the application):

1. *Hardware*: Hardware-level resource scheduling pertains to the execution of processor instructions, as defined by a specific Instruction Set Architectures (ISA), such as the the common $x86\_64$ [122] ISA. Operating system kernel-level components interact with the hardware level using specific instruction sets through interface registers, such as instruction registers. Scheduling required to execute instructions submitted by the Kernel-level is executed at the hardware-level.

2. *Kernel*: Kernel-level scheduling pertains to the prioritized access of shared resources on the hardware-level. Kernels work as an interface layer between hardware and application processes.

3. *Application "Agent"*: Agent layer scheduling pertains to the intra-instance management of application resources. In this context, an instance is an instantiation of processes (plugins) isolated within same kernel instance. Isolated resources are discussed in detail in Section 4.1.1, *Isolated Resources*. Examples of agent scheduling include the management of resource competition between two Cresco plugins running on the same devices.

4. *Application "Regional"*: Regional scheduling pertains to the management of Agent resources across potential resource providers. In this context, a provider is a manager of resources for many kernel-level schedulers (Isolated Resources), such as a *Infrastructure as a Service* (IaaS) provider discussed in Section 4.1.2.

Examples of regional scheduling include the provisioning of virtual machines to support Cresco agents and the deployment of interoperating plugins between agents managed by the same provider.

5. *Application "Global"*: Global scheduling pertains to the management of resources between regions and their related resource providers, including isolated resources and edge devices. Examples of global scheduling include provisioning workload operations across multiple regional providers. For example, data collection services might be scheduled on sensor devices across a city, the aggregation of a cities sensor data might be assigned to *region X*, a city data center resource, and analysis resulting data might be assigned to *region Y*, a designated public cloud provider.

Edge computing framework must address a number of computational environments and resource types. Centralized (isolated) computing resources typically include central processing (CPU), volatile (RAM), and non-volitile (Disk)[1] storage components that work together as independent computational resources. Communication between computation and storage components within individual central nodes is consistent across components using the same communication channels. For example, the process of moving data from one processor to central memory is typically uniform across processors. Perhaps more importantly, a central scheduler has full view of the current state of computational components and can manage resources accordingly. Distributed computing, as the name suggest, distributes computation to otherwise independent computational resources through communication networks. Depending on the distributed computing architecture, communication and resource uniformity varies greatly. For example, locally-distributed high-performance computing clusters are typically composed of nodes and communication resources that are uniform and located in the same data center. For instance, node architectures, sizing, and com-

[1]Recent advancements [123] in non-volatile memory latency are blurring the lines between RAM and disk storage.

munication can vary greatly for the highly geographically-distributed *Folding@Home* [124] project, where individuals around the world volunteer personal computational resources for molecular dynamics simulations.

Edge computing applications are made up of isolated, locally-distributed, and geographically distributed components. In the following sections we describe how the Cresco framework manages specific resource environments.

4.1.1 Isolated Resources

In the context of isolated resource management, we assume devices are capable of running an Operating System (OS), such as Linux, (which allows for kernel-level resource management). Alternatively, resource management is maintained by a high-level API, such as those provided by virtual machine and container [125] management systems such as Docker.

Resource request by application processes are managed by kernel schedulers, which determine access priorities based on scheduler implementations. Implementations of *Weighted Fair Queueing* [126] algorithm are commonly used in kernel-level scheduling. The Completely Fair Scheduler (CFS) [127] has been the default scheduler in the Linux kernel since the *2.6.23* release. Covering the complexities of kernel-level scheduling is beyond the scope of this dissertation. However, it is sufficient to know that kernel schedulers are manipulated through kernel-level function calls to control the process-level priority of resource access. As described in Section 3.7.3, *Process Manager Plugin*, we have implemented a process management plugin that provides kernel-level priority control through the use of *nice* and *ionice* settings.

We briefly discussed Linux kernel namespace isolation of network services in Section 1.3.2, *Identification*. Linux namespaces have been part of the Linux kernel since the *2.4.19* release in 2002. While the implementation of kernel namespaces is beyond the scope of this document, it is sufficient to know that namespaces operate under a

parent-child[2] hierarchy where sibling resources are isolated from each other. Kernel namespaces are used to isolate a number of system resources including:

- *Process IDs (PID)*: Using namespaces it is possible to have multiple process trees nested and independently managed within a parent-child namespace hierarchy. For example, process IDs P_1 and P_2 in a parent namespace might map to process ID P_0 in two independent child namespaces.

- *Networks (NET)*: Namespaces are used to separate network devices and services such as network interface controllers (NICs), routing tables, and firewall rules.

- *Hostnames (UTS)*: Namespaces that allow for hostname operating system identifiers to be assigned to child namespace.

- *Filesystem mounts (MNT)*: Namespaces that allow filesystem mounts to operate independently between child namespaces. For instance, the root (/) filesystem mount points are configured independently between child namespaces.

- *Inter-process communications (IPC)*: System V inter-process communication [128] is isolated between child namespaces.

- *Users IDs (UID)*: User accounting are isolated between child namespaces.

Namespaces provide the fundamental underpinnings necessary for containerization under the Linux operating system. Containers are an operating-system-level virtualization technology, where user environments, including operating system dependencies, are run under a parent kernel in isolated namespaces. Linux Kernel Control Groups (cgroup) [129] are used to impose resource imitations, prioritization, accounting, and control of Linux processes. The most common container-based management technology is Docker [119], which in addition to namespaces provides management

[2]Unless otherwise specified, process interactions with the Linux kernel are performed in the "default" namespace.

for a number of additional image, storage, and network resources. Docker implements granular container resource controls through the use of cgroups. As described in Section 3.7.4, *Container Manager Plugin*, we have implemented a container management plugin that provides kernel-level priority control through container-based resource assignments.

Operating-system virtualization should not be confused with lower-level hardware virtualization. In hardware virtualization, hypervisors [130] running on physical hardware present emulated representations of physical components to operating systems. From an operating system prospective, virtual hardware is functionally equivalent to physical hardware. Similarly, from a Linux process prospective, kernel interactions within a isolated child namespaces are functionally equivalent to those in the default namespace. However, it is worth noting that it is generally accepted that operating-system-level isolation is computationally less expensive [131] than hardware virtualization, where workload dependencies are meet equivalently using the two technologies.

Virtual machine and related infrastructure management frameworks will be discussed in the next section.

4.1.2 Infrastructure as a Service (IaaS)

Infrastructure as a Service (IaaS) is a form of cloud computing that is focused on the management of computational, network, and storage infrastructure components. IaaS is most commonly used to automate the provisioning of virtual machines. IaaS resource management provides identity management, provisioning, monitoring, and resource utilization accounting. There exist a number of public IaaS providers including but not limited to Amazon Web Services [16], Google Cloud Platform [132], and Microsoft Azure [17], where underlying resources are owned and operated by the public cloud provider. In addition, there exist software such as OpenStack, that can be used to deploy private clouds, where the underlying cloud infrastructure is owned

and operated by the institution consuming the resources.

IaaS frameworks provide programatic interfaces to manage collections of resources across individual servers, server clusters, and collections of geographically distributed clusters. Collections of underlying infrastructure such as CPU, RAM, and Disk storage are assigned to virtual machine instances. As described in the previous section, virtual machines provisioned from IaaS providers function as isolated resources. Clusters of locally-distributed servers managed by IaaS frameworks are typically referred to as *zones*. Parameters such as resources sizing and zone identification are specified at time of instance creation. A kernel-level scheduler assigns time to system-level (CPU, RAM, IO, etc) resources on a single node to processes, in order to satisfy workload request. Similar to a kernel scheduling assigning time to infrastructure components, an IaaS scheduler provisions infrastructure resources from a number of potential nodes to form virtual machines. The IaaS scheduler attempts to balance resource assignments evenly between infrastructure components within availability zones. For example, if a IaaS cluster manages N infrastructure nodes with equal available resources and N resources are requested from those nodes, then a single resource will be provisioned from each node. Likewise, consider the case where initially M resources were provisioned on $node_0$ and $M * 2$ resources were provisioned on $node_1$. If M additional resources are requested, the IaaS scheduler will assign resources from $node_0$. After this assignment $M * 2$ resources are provisioned from both $node_0$ and $node_1$, thus balancing allocations across resource nodes.

Many IaaS frameworks, such as OpenStack, lack the concept of inter-node resource scheduling. If resources for high-value workloads are provisioned on the same node with low-value allocations, the two will compete for resources without preference from the kernel-level scheduler. Similarly, instances that compete for specific resources, such as highly computational workloads are intentionally scheduled on separate nodes. The lack of inter-node scheduling in IaaS frameworks leads to workload performance variations across nodes. As we discussed in the previous section, it would be possible

to manage allocation preference on this level, but this is not typically the practice within IaaS frameworks.

Vertical resource scaling is the process of increasing the access to, or capacity of isolated resources, such as increasing a processes priority or increasing the quality of resources assigned to an individual virtual machine. In IaaS, vertical scaling can be accomplished for individual resources, to the extent of the maximum infrastructure resource available. Specifically, resource allocations can not exceed the physical size of individual components. Resources such as CPU and RAM are typically[3] deployed from resources in the same power domain (same node), while storage is often provided from a separate domain. For example, IaaS can not be used to provision a virtual machine in which the size of the virtual machine exceeds the capacity of the largest available physical machine[4].

Horizontal resource scaling is the process of increasing the quality of process or isolated resources, such as the distribution of workload across multiple threads and processor cores or the provisioning on additional Cresco Agents and Plugins. Horizontal scaling in IaaS is accomplished by provisioning additional IaaS instances. For example, IaaS frameworks provide the ability to provision (clone) new virtual machines from existing machines. Likewise, existing virtual machines can be suspended or deleted, resulting in a release of active resources. The process of dynamically scaling applications using this method is referred to as *elastic computing*[135]. A single node OpenStack deployment might only scale a few instances across a few CPU cores, where as large deployments like those maintained by CERN, might span thousands of instances over one hundred fifty thousand cores [136].

As described in Section 3.7.1, *Infrastructure as a Service (IaaS) Plugin*, we have implemented a plugin that provides the ability to interface with IaaS management frameworks for the propose of infrastructure provisioning. While IaaS does not allow

[3]This may change in the future with the advancement of so-called memory-centric computing [133].

[4]Software exist [134] to overcome this problem, but it is not considered a typical feature of an IaaS framework.

us to observe the underlying physical resources of instances, we can observe resource utilization and workload performance of Cresco-enabled instances. As described in Section 3.7.2, *System Information Plugin*, we have developed a plugin to report the operating state, inventory, resource utilization, and performance of isolated resources (physical devices, virtual machines, containers, etc).

In the next section we discuss the language used to describe and abstract Cresco applications from underlying resources, allowing for automated and semi-automated resource management.

4.2 Cresco Resource Model

In the previous sections we discussed various types of resources across several operating environments. We discussed how specific Cresco plugins manage resources used by edge applications. We have claimed that Cresco plugin configurations can be used to represent workloads and that collections of plugins are used to construct Cresco applications. In this section we will discuss how Cresco Plugin configurations are developed and related.

Cresco Plugin configurations and related applications relations can be assigned manually (statically). In fact, the entire Cresco architecture supports static assignment of agent names, roles, plugins, and inter-agent relationships. In Section 5.3, *Genomic Processing Framework*, we cover a Cresco-based application that is composed primarily of statically assigned components. In this use case, the Cresco framework is considered part of an In Vitro Diagnostic (IVD) device and the enforcement of validated (static) resources is required. Alternatively, Cresco Plugin configurations can be generated and assigned to applications dynamically. By contrast in Section 5.4, *GLobal Edge Application Network*, we discuss how the Cresco framework is used to dynamically manage a globally distributed edge application network.

While we don't typically think of genomic processing and global edge applica-

tion networks as being related, there are similarities from a computational model standpoint. In both cases application components (data collectors, privacy modules, encryption, event processing, reporting, etc.) can be represented as nodes and the flow of application data between components represented as edges. The resulting collection of components can be modeled as an acyclic graph, commonly referred to as an application pipeline. If we are able to abstractly describe graphs that model applications without assigning unnecessary operating environment or location constraints we can bring to bear a host of computational techniques to manage distributed applications.

In the following sections we cover how applications are described, components represented, and resources assign within the Cresco framework.

4.2.1 Cresco Application Description Language

We propose the Cresco Application Description Language (CADL) to model distributed applications managed by the Cresco framework. CADL is a graph language, where nodes represent Cresco Plugin configurations, edges represent relationships between Cresco Plugins, and graphs composed of nodes and edges represent applications. The format of CADL node fields and requirements are shown below:

- [**node_id**] *(required, unique)*: Node IDs are unique identifiers for nodes within specific pipelines.
- [**node_name**] *(required)*: Node names are used as short descriptions for nodes within specific pipelines.
- [**type**] *Required*: Node types represent specific Cresco Plugin implementations.
- [**description**] *(optional)*: Node descriptions are used for descriptions of node operations within specific pipelines.

- [**params**] *(optional)*: Params are collection of key-value pairs that specify plugin-specific configurations. Manifest descriptors within Cresco Plugin implementations determine parameter requirements.

- [**isStateless**] *(optional)*: The isStateless parameter is a boolean value representing the ability of the configuration instantiation to be migrated between Cresco Agents without maintaining plugin memory state. For example, the configuration of a stateless plugin processing (filtering, format conversion, etc.) data from a source with delivery guarantees, such as a durable queue, can be migrated without memory migration or data loss.

- [**isSource**] *(optional)*: The isSource parameter is a boolean value designating the node as a data source for a potential external pipeline. For example, a destination node that removes sensitive data in one pipeline might serve as the data source for another pipeline.

- [**location**] *(optional)*: The location parameter is used to relate nodes to specific agents or locations. For example, to sample the network traffic at a specific location, the location parameter would need to match the Cresco Agent location parameter at the desired location.

Listing 4.1 shows the node description for a plugin that serves as an AMQP data exchange at location X.

Listing 4.1: CADL Node

```
1  "node_id":"0"
2  "node_name":"pStart"
3  "type":"amqp"
4  "params":
5    "amqp_server":"localhost"
6    "outExchange":"eQuery"
7    "amqp_login":"login"
8    "amqp_password":"password"
9  "isStateless":true
10 "isSource":true
11 "location":"X"
```

As previously mentioned, edges represent the relationship between nodes. The format of CADL edge fields and requirements is shown below:

- [**edge_id**] *(required, unique)*: Edge IDs are unique identifiers for nodes within specific pipelines.
- [**node_from**] *(required)*: The node_from parameter is used to designate source node_id for the edge within a specific pipeline.
- [**node_to**] *(required)*: The node_to parameter is used to designate destination node_id for the edge within a specific pipeline.

Listing 4.2 shows an example of a CADL edge description relating two node_ids.

Listing 4.2: CADL Edge

```
1 "edge_id":0,
2 "node_to":"1",
3 "node_from":"0"
```

CADL node and edge descriptions are combined to form a pipeline description, which represents a Cresco application. The format of CADL pipeline fields and requirements are shown below:

- [**pipeline_name**] *(required)*: Pipeline names are used as short descriptions for pipelines maintained by a specific Cresco Global Controller.
- [**nodes**] *(required)*: Nodes are collections of CADL node descriptions. At least one node description must exist for a pipeline to be considered valid.
- [**edges**] *(optional)*: Edges are collections of CADL edge descriptions.
- [**description**] *(optional)*: Pipeline descriptions are used to describe the operation of pipelines.
- [**isFaultTolerant**] *(optional)*: The isFaultTolerant parameter is a boolean value designating that pipeline components should be rescheduled if failures are detected.

CADL pipelines can be used to describe a number of applications. Suppose we want to construct the following application pipeline:

1. Read JSON-formated Netflow records from an AMQP data source and emit data to a downstream node.

2. Read data from an upstream node, marshal JSON data into a strongly typed Netflow class, calculate the top ten network flows in a one minute sliding window, and emit JSON-formatted data to a downstream node.

3. Read data from upstream node and place results in a FIFO (first-in-first-out) memory buffer, which is externally accessible through a RESTful interface provided by the plugin.

Listing 4.3 shows a three state CADL pipeline for the previously described application.

Listing 4.3: CADL pipeline

```
1  {"nodes":[
2  {"node_name":"pStart","type":"amqp","node_id":"0","params":{"
       amqp_server":"localhost","outExchange":"someexchange","
       amqp_password":"somepassword","amqp_login":"somelogin"}},
3
4  {"node_name":"netFlow Query","type":"esper_query","node_id":"1","
       params":{"query_class":"netFlow","query_string":"select ip_src
       , ip_dst, bytes from netFlow.win:time(1 min).ext:sort(10,
       bytes desc)"}},
5
6  {"node_name":"","type":"membuffer","node_id":"2","params":{"
       data_url":"http://localhost/API/buff0"}}],
7
8  "edges":[
9  {"edge_id":0,"node_to":"1","node_from":"0"},
10 {"edge_id":1,"node_to":"2","node_from":"1"}],
11
12 "pipeline_name":"Top 10 Netflows"}
```

CADL descriptions are submitted to Cresco Global Controllers for interpretation and resource scheduling. Figure 4.1 shows the steps taken in the deployment of a Cresco application.

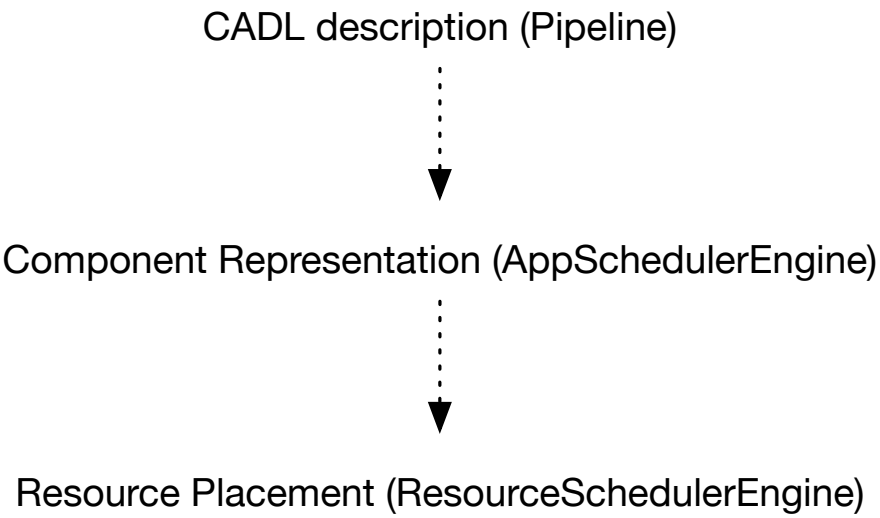

Figure 4.1: Cresco Application Process

In the next section we will cover how CADL components are represented within the Cresco framework.

4.2.2 Component Representations

The *AppSchedulerEngine* process is responsible for identifying incoming pipelines, translating pipeline request into plugin resource requests, and submitting plugin request for resource scheduling. CADL pipeline descriptions are submitted to the Cresco Controller API, as described in Section 3.6.5. On submission, the pipelines are recorded to the Cresco Global Controller database and submitted to the *AppScheduleQueue* queue. The *AppSchedulerEngine* process reads incoming pipeline request messages from the *AppScheduleQueue*, parses the CADL pipeline description, and creates nodes and edge configurations representing the pipeline in the controller graph database. The following node and edge class instances are used to describe pipelines in the database.

- **Node** *pipelineNode* : pipelineNodes are roots nodes for pipelines described by the CADL language.

- **Node** *vNode* : vNodes maintain a record of pipeline nodes as described by the CADL language.
- **Node** *iNode* : iNodes represent Cresco Plugin resource assignments in relation to vNodes. There is a one-to-many relationship between iNodes and vNodes.
- **Node** *eNode* : eNodes represent data exchange mechanism between iNodes. eNodes maintain configuration information such as data exchange locations and authentication information generated by Cresco during pipeline interpretation.
- **Node** *rNode* : rNodes maintain a record of a specific Region in the Cresco database.
- **Node** *aNode* : aNodes maintain a record of a specific Agent in the Cresco database.
- **Node** *pNode* : pNodes maintain a record of a specific Plugin in the Cresco database.
- **Edge** *isVnode* : isVnode associates pipelines to vNodes.
- **Edge** *isVconnected* : isVConnected associates vNodes that are connected based on CADL description.
- **Edge** *isInode* : isInode associates vNodes and iNodes.
- **Edge** *eOut* : eOut associates the flow of data from a iNode to a eNode.
- **Edge** *eIn* : eIn associates the flow of data to a iNode from a eNode.
- **Edge** *isEconnected* : isEconnected edge indicates the directed flow of data from one eNode to another.
- **Edge** *isAssigned* : isAssigned associates Cresco Plugins to iNode configurations.

The relationship of Cresco database classes is shown in Figure 4.2.

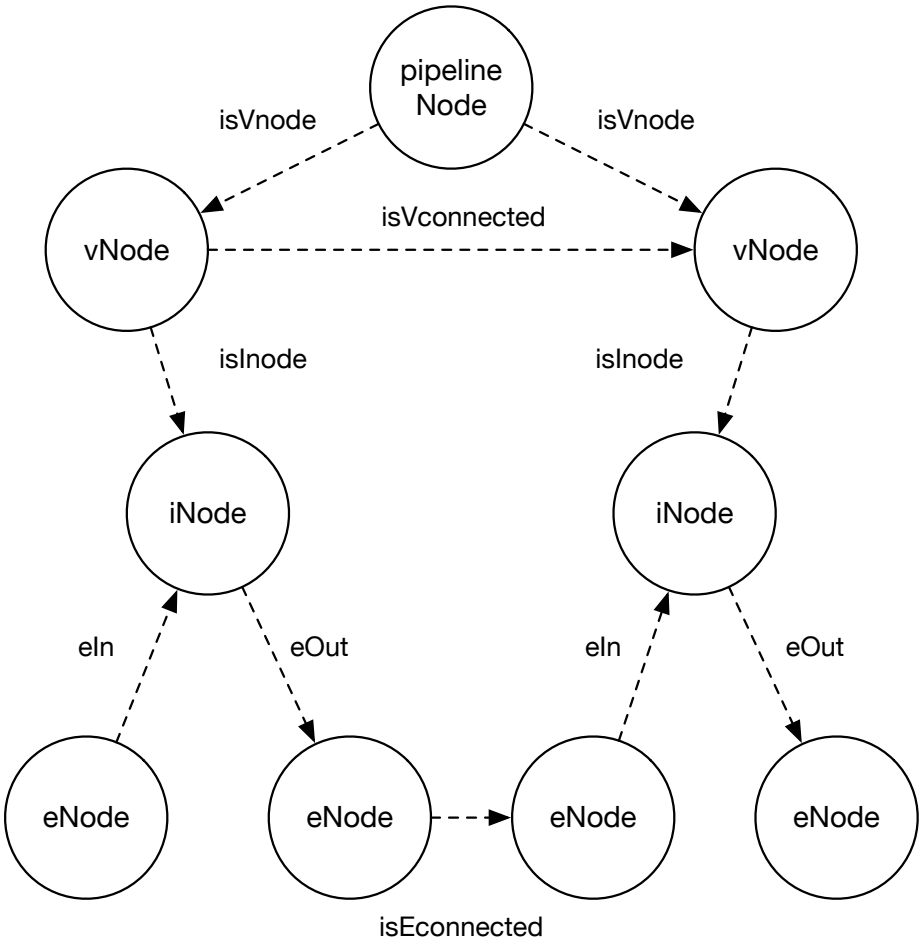

Figure 4.2: Cresco Application Graph

On submission to the Cresco Global Controller, a pipelineNode is created to record the initial CADL description. The incoming CADL expression is then forwarded to the *AppSchedulerEngine* process for scheduling, where nodes described in the pipeline are represented as vNodes in the database. If needed, iNodes are created for each vNode to represent plugin implementations of request vNodes. However, iNodes can be shared between pipelines. On iNode, and related eNode generation, we use graph database query functions to search for existing candidate iNodes with exact the same "params" attributes and data paths as the iNode representation of the pipeline-specific vNode. If an iNode replacement candidate is found, we traverse "isECon-

nected" and "eIn" relationships to determine if iNode and eNode implementations for the source data path are exactly the same as the sub-graph to be implemented. Listing 4.4 shows a query used to traverse candidate iNode data paths.

Listing 4.4: Cresco Database Node Traversal

```
1 select from (traverse in() from [node_id]) where @class <> 'vNode
    ' and @class <> 'Pipeline'
```

If a valid iNode sub-graph is found, the vNode "isINode" relationship is assigned to the existing iNode. All remaining iNodes without an existing "isINode" relationship are submitted for resource scheduling.

In the next section we describe resource placement.

4.2.3 Resource Placement

The *ResourceSchedulerEngine* is responsible for identifying incoming request, determining the best-fit location for resource assignment, and is responsible for resource assignment to specific agents. Incoming requests are identified by MsgEvent messages sent to the resourceScheduleQueue queue. Resource placement messages can be generated from a number of sources including the *AppSchedulerEngine*, Cresco API, or from *High-level Operations* modules, described in Section 4.3. Resource placement message contain resource identifier information in the form of inode_id and resource_id assignments, predicate restrictions such as location_id, and all parameter information required to provision a specific plugin. Resource scheduling takes place in the following order:

1. iNodes can represent plugin implementations used by a number of pipelines. The status of a pipeline that makes use of a specific iNode might be inactive, whereas the iNode itself is active in another pipeline. The status of the existing iNode representing the requested vNode is confirmed to be inactive to prevent duplicate resource assignments.

2. The controller plugin inventory is checked to determine if the requested plugin exists on or is accessible by the controller.

3. The plugin information specified in the request is checked against the local plugin manifest to determine if the required plugin parameters have been provided.

4. aNodes (Agent representations) information is queried to determine if predicate (location, cpu-core-count, memory-available, etc.) requirements can be satisfied. Agents satisfying predicate restrictions are placed on the agent candidate list.

If no agent candidates exist for resource scheduling, the unscheduled resource request is recorded in the controller database and if enabled, the *Guilder* module, described in Section 4.3.1, is notified of a pending resources request. If appropriate resources are identified in the future the pending resource will be scheduled.

The candidate agent list could include a number of agents, all technically capable of satisfying the resource request. If enabled, *Optima*, described in Section 4.3.3 is used to determine the most appropriate candidates for resource assignment. Otherwise, the least loaded (based on plugin count) agent is selected from the candidate agent list for resource assignment. Once an agent is selected for resource assignment the global controller sends a message to the agent to download the latest plugin. Once plugin download has been verified, a message is sent to the agent to start the plugin based on the parameters provided in the resource request. The *ResourceSchedulerEngine* process will spawn a new *PollAddPlugin* process, which watches the controller database for notification that the plugin related to the initial iNode has been started and is producing KPI updates. If the *PollAddPlugin* timeout (default of 30 seconds) is exceeded, the iNode is marked as failed, an agent health check is started, and the resource scheduling process is restarted.

4.3 High-level Operations

A number of high-level operations modules have been developed to demonstrate the benefits of the Cresco approach to distributed resource and application management. The modules and related services described in this section serve as proof of concepts and are not required for Cresco operations. A list of high-level projects and their related modules is shown below:

- *Guilder*: provides analysis and coordination for resource acquisition, relinquishment, trading, and service cost.
- *Futura*: manages analysis related to workload classification, clustering, resource utilization, and future needs prediction.
- *Optima*: manages analysis related to scheduling optimizations of individual workloads, pipelines, and entire networks of Cresco-managed components.

4.3.1 Guilder

The Guilder project demonstrates the ability of Cresco to acquire, relinquish, and trade resources with both fixed and dynamic utilization costs. In this context, fixed resources are computational devices typically acquired through capital expense (CAPEX). CAPEX cost are often amortized over the life of the device providing a fixed cost regardless of resource utilization. Dynamic utilization cost are resources provided by public and private cloud providers in the form of dynamically provisioned services, such as virtual infrastructure. Dynamic resources are considered operating expenses (OPEX), where cost occurs only while the resource is in use. Fixed or dynamic resources are often determined by specific workload requirements. For example, a workload designated for a specific capture device or location might force the assignment of a fixed cost resources in the form of a edge device. Alternatively, a request of 1000 workload resources might exceed fixed resource capacity and require

the provisioning of resources from a public cloud environment. In many cases we have the option to deploy workloads on a number of fixed and dynamic resources.

Resource Provider Evaluations Guilder provides analysis services used to determine the value of potential resources. Agents and related information-gathering plugins are deployed to infrastructure *Resource Providers* capable of hosting Cresco workloads, which include both physical devices and virtual machines. In the context of scheduling, Resource Providers are resources that are already under the management of the Cresco framework and should not be confused with potential resources provided by IaaS resource providers. Agents can be deployed directly on standalone devices and resources that compose private computational clouds. The underlying infrastructure supporting public clouds is not accessible for direct monitoring. However, agents can be deployed on virtual machines provided by public and private clouds to provide assessment of the performance of the underlying system. If fact, due to over-subscription and service tier (preference to one node over another) variations of resources in virtual environments node-specific evaluation is the only way to determine realized resource capacities. As described in Section 3.7.2, *System Information Plugin*, Cresco Plugins can be used to to determine the logical processor core count C and idle load I of a specific host. On plugin initialization the System Information Plugin performs the NIST SciMark 2.0 [137] benchmark on the host system. The benchmark B is performed on a single logical core of the system, which allows us to estimate overall computational potential P as: $(C) \cdot (B) = P$. While the benchmark value is obtained only on plugin initialization, system KPI message are sent periodically (default of 15 seconds) to Cresco. Using system KPI messages we estimate available computational capacity A as: $(P) \cdot (I/100) = A$. Currently, Guilder demonstrates resource evaluations based on computational performance. Similar assessments can be made using memory, disk, or network information provided by system KPI information. Resource analysis services provide by Guilder are used by

the *Optima* project described in Section 4.3.3.

Resource Provider Pricing and Provisioning As described in Section 3.7.1, *Infrastructure as a Service (IaaS) Plugin* Cresco Plugins provide methods to manage resource provided by a number of cloud providers. Guilder provides services to provision and de-provision resources as needed from cloud providers supported by the IaaS plugin. In the context of Guilder, IaaS resources are managed in the form of virtual machine instances designated by a specific VM types[5]. VM types are predefined combinations of virtual cpu(s), memory, disk, and network capacities. For instance a small VM type might provide a single cpu, 512MB of memory, and a 1GB disk space, while a large type might provide 8 cpus, 32GB of memory, 1TB of disk space, and a 10GB network adapter. Typically, both public and private cloud providers conform to Amazon EC2 VM types [138].

The use of common VM types provides a method to compare public and provide cloud offering cost. Standalone hardware and private cloud costs based on VM types are statically set through Guilder configuration values. In addition, dynamic cost for public cloud resources are determined through provider specific interfaces, such as the Amazon EC2 price API [139]. Guilder maintains the cost for all provisioned and potential resources.

Guilder demonstrates how Cresco participates in a resource market with a number of potential "buyers" and "sellers". In such systems, the use cost of similar services may vary widely and agents participating in said sources can have various policies for utilizing resources depending on pricing and availability. Viewing distributed resources as a market provides one way of approaching resource allocation problems.

[5]Depending on IaaS provider, the terms size, flavor, or instance are used.

4.3.2 Futura

The Futura project demonstrates the ability of Cresco to profile workloads and future-needs prediction. Resource utilization and other KPI metrics are observed and recorded by the Cresco framework. As previously mentioned in the *Guilder* section, metrics are gathered from *Resource Providers* to determine potential and available resource capacities. However, in most cases more than one workload will run per Resource Provider, so provider metrics alone are not enough to determine workload-specific resource utilizations.

As described in Section 3.7.4, *Container Manager Plugin*, Cresco Plugins report workload-specific *Resource Metrics*, including average CPU utilization U. In the context of containers, resource metrics are gathered per Linux namespace, which provides an accurate kernel-level account of resources utilized by a container for the duration of its operation. As with the *System Information Plugin*, resource utilization KPIs are reported to the Cresco framework periodically. A summary of resource utilization, such as the total number of container cpu cycles, system system-wide cpu cycles, filesystem bytes read, filesystem bytes written, network bytes received and network bytes transmitted is provided. On report generation, point-in-time resource utilization values such as cpu, memory, filesystem, and network per second utilizations are calculated for use in time-series analysis. Historic records of configuration-specific KPI metrics are maintained by Cresco.

Resource utilization metrics, such as the number of cpu cycles observed to satisfy a specific workload are only useful if we understand the performance characteristics of the observed metric. As previously mentioned in the *Guilder* section, we take stock of resource provider performance characteristics including logical core count C, and single core benchmark values B. Relating Resource Provider and Resource Metric values Futura determines workload utilization W as: $(C \cdot B) \cdot ({}^{100}/_{U}) = W$. The workload utilization value is a device-independent estimation of computational need for a specific workload. With the implementation of additional benchmark metrics in

the *System Information Plugin*, a number of workload utilization values can be generated. The grouping or separation of workload in the scheduling process is highly dependent on the type of resources workloads utilize. For example, computational and memory-intensive workloads compete for resources when provisioned on the same hosts. Alternatively, workloads that heavily utilize network communications might predominately communicate with another workload in the same application pipeline. For cases of intra-pipeline communication, the best approach might be to group these workloads together on a single host or a cluster of hosts in the same geographic location to take advantage of higher communication speeds, lower latency, and lower cost. However, due to effects of so-called data-gravity, a phenomena where data tends to reside in proximity to point of initial processing, high network utilization is often correlated to high disk utilization. As with high cpu and memory high disk utilization workloads should be separated in most cases to avoid resource contention issues. Regardless of the logic used to group or separate workloads, programatic clustering methods are needed to identify groups of similar workloads. Using The Apache Commons Mathematics Library [140] Futura provides clustering functions for Resource Metrics. Using the k-means [141] clustering method the Futura clustering function takes as input the desired number of clusters, *clusterK* and the cluster method, *clusterType*, shown below:

- *All*: Cluster based on all cpu, memory, disk, and network statistics.
- *Disk-Network*: Cluster based on disk and network statistics.
- *CPU*: Cluster based on cpu statistics only.
- *Memory*: Cluster based on memory statistics only.
- *Disk*: Cluster based on disk statistics only.
- *Network*: Cluster based on network statistics only.

Futura clustering functions are used in both scheduling optimization functions and longer-term resource prediction analysis. Using the historical data maintained by Futura not only can we predict needs for specific workloads, but using time-series samples we can estimate future needs based on past resource utilization. In analysis of fixed cost resource planning we can use Futura clustering functions to estimate the most appropriate configurations (higher or lower cpu, memory, disk, network) for purchase. Likewise, during the resource scheduling process if there are not enough resources to satisfy request, new dynamic cost resources can be acquired using Guilder. However, workload resource needs must be translated into specific VM type(s) that can be acquired by Guilder. The *Optima* project, described in the next section, is used to determine optimal resource assignments.

4.3.3 Optima

The Optima project demonstrates the benefits of Cresco in resource scheduling optimization. In this context, scheduling is process of satisfying computational resource request with appropriate resource assignments. There are a number of ways scheduling optimization can take place. For example, the selection of the highest performing host for a specific workload, or the section of most geographically diverse hosts to deploy a highly-available pipeline. In addition to initial resource assignment, optimization techniques can be applied to existing resource assignments such as the global optimization of all resource assignments across pipelines and workloads based on overall costs. In the case of existing resource assignments we are concerned with the reassignment, expansion, or contraction of resources for groups of workloads. Ours is not a project in combinatorial optimization, but constraint programming tools are used as part of this project to solve optimization problems.

In Section 4.2.2, *Component Representations*, we described the AppSchedulerEngine. In Section 4.2.3, *ResourceSchedulerEngine*, we described the ResourceSchedulerEngine. The AppSchedulerEngine is responsible for the translation of CADL descriptions into

Cresco Plugin configurations that can be scheduled by the ResourceSchedulerEngine. If there is no predicate assignment (location, region, agent, etc.) in the CADL to directly relate a CADL node to a specific agent or groups of agents, the AppSchedulerEngine must determine the agent assignment. When enabled, Optima is used to determine a potentially optimal assignment of CADL nodes to agents. Currently, we are using a single estimated variable in our optimization calculations and make no guarantee of assignment optimality. However, we do demonstrate how potential projects might make use of Cresco to obtain distributed sources needed for optimization calculations.

Resource Provider Location Problem (RPLP) There are a number of possible approaches to determine the optimal assignment of workloads to resource providers. For the purposes of demonstration we adapted the Warehouse Location Problem (WLP) [142] to determine the best workload to resource provider assignment(s).

In the Resource Provider problem, a resource scheduler considers a number of potential providers to supply resources to specific workloads. Each possible resource provider has a potential capacity designating the maximum quantity of resources that is can supply. Each resource must be supplied by exactly one resource provider. The resource cost to satisfy a resource request depends on the resource provider in relation to initial purchase cost and observed workload resource utilization metrics. The objective is to determine which resource providers to use, and which resource providers should supply the various resource requests, such that the sum of total pipeline cost is minimized.

Optima makes use of the Choco [143] constraint programming library to implement a solver for RPLP. The implemented solver takes as input the number of the number of workloads (stores) S, the number of Resource Providers (warehouses) W, the cost for using a provider instance (opening a warehouse) C, an array of provider instance capacities K, and a matrix of cost P, relating the cost for a specific provider

to satisfy the resource request of a specific workload. The number of workloads is provided by the scheduler, Futura data is used to calculate W and K values, C is determined through Guilder data, and the cost matrix P is calculated using both Futura and Guilder data. The resulting solver solution provides workload assignments and the total cost for all assignments. If node assignment is not possible due to resource constraints the optimization process fails and the scheduler is notified that additional resources are needed.

Resource Provider Optimization (RPO) As previously mentioned, Resource Providers in the context of scheduling pertain to resources already under Cresco management. When additional resources are needed, Optima is used to translate workload resource needs into specific VM types to be managed by Guilder. Resource sizing specified by VM types results in new Resource Providers managed by Cresco.

Analysis performed by the University of Kentucky [144] compared 1500 real-world virtual machines spanning academic, research, healthcare service areas with over 50 public cloud offerings, based on VM type specifications. The virtual machines in the study were manually (IT professional selected resource quantities) provisioned without specific VM type constraints. Analysis showed that 94% of the existing virtual machine resource configurations could be matched exactly to a VM type offered by a public cloud provider, based on virtual cpu and memory metrics. Introducing storage into the comparison, where a match is made if the storage specified by the VM type is within 35% of the actual storage, reduces the match rate to 15%. These results suggest that the assignment of a single instance of a specific VM type will result in the underutilization of at least one resource in 85% of cases.

There are several techniques to address the underutilization of instance resources due to public cloud over-provisioning. If access to private cloud infrastructure is permitted, custom VM types can be created for specific workload needs. However, this creates complications when trying to compare the cost of public and private cloud of-

ferings. In most cases, better approach is to assign more than one workload to a single instance. In order to determine the appropriate VM type size we can make use of the optimization approach described for RPLP. If we add potential Resource Providers, based on available public and private VM types to the list candidate providers, we can determine a potentially optimal size and quantity of new resources for Guilder to acquire.

Global Provider Optimization (GPO) Through the scheduling process various sizes of Resource Providers are provisioned to support collections of workloads. While attempts are made to optimally size the creation of Resource Providers per pipeline request, over repeated scheduling iterations Resource Provider fragmentation, much like memory or disk fragmentation [145] can occur. For example, suppose we add a new Resource Provider RP_0 of size S_0 and cost C to satisfy the requirements of a new pipeline P_0. Now suppose we add another pipeline P_1 that requires the addition of another Resource Provider RP_1 of size S_1 and cost C. From the standpoint of the individual pipelines P_0 and P_1 resource assignments are optimal. While not optimal for individual pipeline assignment, suppose a potential Resource Provider RP_p of size $S_0 + S_1$ and cost $C \cdot 1.5$ is avalable. From a global prospective taking into account both pipelines the single Resource Provider RP_p would result in a lower overall cost. As we previously stated, it is unlikely that the provisioned Resource Providers fully utilize all available resources, so perhaps there exist a Resource Provider RP_{p-1} of $C \cdot 1.25$ that can satisfy the combined resource needs of pipelines P_0 and P_1.

Global Resource Provider optimization is accomplished in much the same way as RPLP and RPO. Our provider list is composed of potential Resource Providers as reported by Guilder. Our workload include all current configurations, which are not otherwise restricted by a agent assignment predicate. The resulting solver solution provides workload assignments for potential Resource Providers and the total cost for all assignments. If potential cost reductions exceed a threshold specified in

Optima configuration, the scheduler is notified and the rescheduling process it started.

In Chapter 5, *Case Studies of Edge Computing*, a number of project implementations are discussed that either directly contributed to Cresco components or make use of the Cresco framework.

5

Case Studies of Edge Computing

In Chapters 1, *Edge Computing Introduction*, and 2, *The Architectural Model*, we discussed a number of potentional use cases for the Cresco framework in relation to edge computing. In this chapter we cover specific case study projects that either directly led to the development of specific Cresco components and capabilities or are works based on the Cresco framework.

In Section 5.1, *Distributed Stream Analysis System*, we cover work originally published under the title *Scalable Hybrid Stream and Hadoop Network Analysis System* [146]. This work led to several key Cresco design principles and components including our stream-based (push) data model, NetFlow, Complex Event Processing (CEP), and Hadoop components.

In Section 5.2, *Workload Characterization*, we cover work originally published under the title *Collating time-series resource data for system-wide job profiling* [147]. This work led to the development of Cresco Global controller workload characterization module described in Section 4.3.2, *Futura*.

In Section 5.3, *Genomic Processing Framework*, we cover work originally published under the title *Constellation: A secure self-optimizing framework for genomic processing* [148]. The resulting genomic processing framework developed as a result of this work is an example of a statically deployed Cresco application.

In Section 5.4, *GLobal Edge Application Network*, we describe the use of Cresco in the development of an edge computing network.

5.1 Distributed Stream Analysis System

Collections of network traces have long been used in network traffic analysis. Flow [149] analysis can be used in network anomaly discovery, intrusion detection and more generally, discovery of actionable events on the network. The data collected during processing may be also used for prediction and avoidance of traffic congestion, network capacity planning, and the development of software-defined networking. Typically, network-capture devices are placed at points of network data aggregation, such as the point of exchange between organizational networks and the public Internet. However, as network flow rates increase beyond the capacity of single analysis devices, many organizations find themselves either technically or financially unable to generate, collect, and analyze network flow data.

In this section we describe a network analysis system that addresses problems of scale through the use of distributed computing methodologies. This system was developed and deployed at the University of Kentucky (UK). The UK campus network is composed of over 17,000 network segments and over 5000 wireless access points, which serve over 36,000 students, faculty and staff.

In this system both stream and batch distributed processing frameworks are leveraged. Stream processing methods are employed to collect and enrich data streams with ephemeral environment information, such as the current associated access point location for an observed campus network address. Enriched stream-data is used for event detection and near real-time flow analysis by an in-line complex event processor. Batch processing is performed by the Hadoop MapReduce framework [150], from data stored in HBase [151] BigTable storage by the stream processor. In benchmarks on our 10 node cluster, using actual network data, we were able to stream process

Table 5.1: UK Network Devices

Device	*Count*
Core	6
Distribution	44
Access	1176
Wireless Controllers	47
Wireless Access Points (AP)	5442
Virtual Switchs	42

over 315K flows/sec. In batch analysis were we able to process over 2.6M flows/sec with a storage compression ratio of 6.7:1.

5.1.1 NetFlow Generation

We estimate the average UK campus data rate between network segments to be 282GB/sec (on the order of 1PB/hour), a rate that far exceeds capacity of commodity servers to generate NetFlows from network observations (packet capture). Simply generating NetFlows from high traffic links is, in itself, a highly computational task [152]. Hardware devices equipped with ASICs, such as routers and switches, are capable of generating line-rate flow exports on aggregations of hundreds of high-speed (100G) links. However, with the introduction of new network protocols and service on existing hardware platforms, even ASIC-based devices can loose the ability to generate NetFlows. In fact, this is the case on the UK network where the use of Multi-protocol Label Switching (MPLS) [153] on the existing network hardware prevents NetFlow generation on the majority of devices.

Table 5.1, shows a list of devices found in the UK network.

We determined that a distributed network capture across the network core was the best location to observe traffic that is traversing both campus and external locations. This required distributed network probes consisting of a network capture devices (server hardware) that generate NetFlow records based on the traffic they observe. The probes ingest aggregates of distribution links from the core routers, effectively

monitoring all traffic passing from distribution to distribution (intra-campus), and core to edge (inter-campus). Each probe device runs an instance of Fprobe [154] for each monitored network interface. In Fprobe we are able to specify the Link layer header size, so MPLS header information is ignored and a NetFlow is generated from the correct IP diagram.

5.1.2 NetFlow Collection

While we are able to generate NetFlows using distributed probes we still have the problem of collection and processing. At the time of this writing no single appliance exist that can collect and process hundreds of thousands of flows per second from aggregated sources. To solve this problems we developed our own NetFlow collectors, which are implemented on the network probe devices. The NetFlow collectors then stream a pertinent subset of NetFlow information to an assigned Advanced Message Queuing Protocol (AMPQ) [155] queues. AMQP queues are provided by servers distributed in geographic proximity to probe devices. In this regard, probe devices act as self-contained edge endpoints capable of generating, collecting, and distributing NetFlows based on their location configuration.

5.1.3 NetFlow Processing

NetFlow records remain on distributed AMQP servers until enqueued by a queue subscriber. This method allows a central system to control the rate of data flow based on how quickly records are acknowledged from distributed sources. Aggregated flows of NetFlow records are subscribed to by an application topology we developed using Apache Storm [156]. Apache Storm is a cluster-based distributed real-time computation system, where functional components are arranged in Storm Topologies. The primary topology components of Storm are Spouts and Bolts. Spouts, as the name suggest, are used to ingest data streams and emit tuples consumable by the application topology. In this context, a tuple is a data diagrams in the form of a key-

value pair. Bolts read tuples from either Spouts or other Bolts, and also typically emit a tuple stream. Normally, tuple transformations, operations, and external data drains occur in Bolts. Our Storm topology is shown in Figure 5.1. Similar to MapReduce [157], Storm distributes and processes tuples of information on multiple nodes and processes. However, unlike MapReduce, Storm will process tuples until the job is manually terminated.

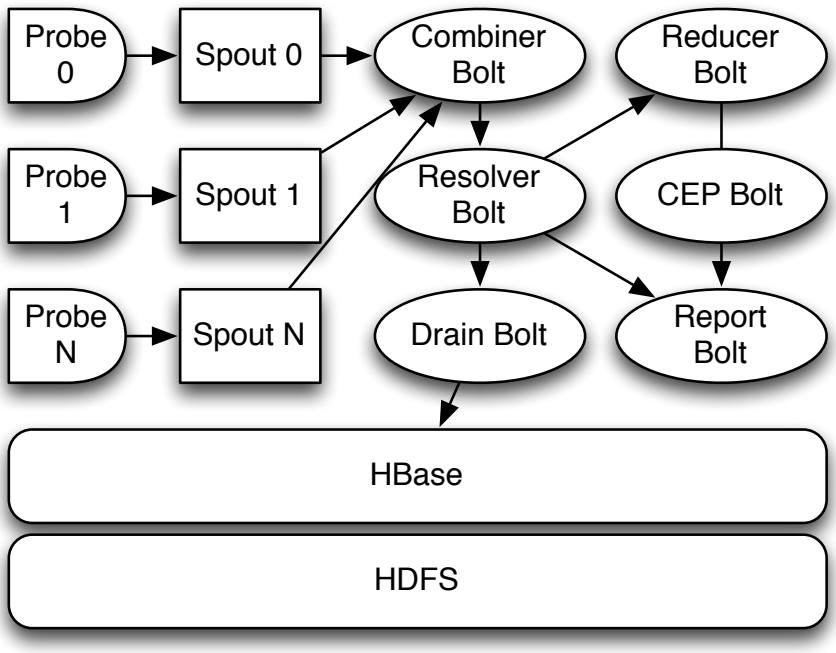

Figure 5.1: Storm Topology

In the context of Cresco, Storm spouts and bolts function in much the same way as Cresco Plugins do in Cresco Application pipelines. A number of Storm Spouts and Bolts were developed as part of this effort including a AMQP Spout based on RabbitMQ [121]. This Spout is used to retrieve bundles of NetFlows generated by the probes and emit exactly one tuple for each flow. Along with building the Storm tuple, the spout also injects a element identifying the originating probe and related network observation area. In effect, AMPQ Spout produces a stream of database records; the attribute names are in the Column 1 of Table 5.2, AMPQ Spout Tuple,

Table 5.2: AMPQ Spout Tuple

Element Name	*Description*
timestamp	Time of flow creation
srcIp	Source IP address
srcPort	Source Port
dstIp	Destination IP
dstPort	Destination Port
byteCount	Sum of bytes in flow
proto	IP protocol
first_t	Router uptime at flow start
last_t	Router update at flow end
collector	Probe Queue Name

while the meaning of these attributes is provided by Column 2, Description.

Additional bolts combine NetFlow fields to produce a tuple keys used to uniquely identify flow records, inject ephemeral data such as current network location, and compile data streams for Complex Event Processing (CEP) [158]. CEP is the term used to describe a collection of methods used in the analysis of unbounded streams of information. A CEP engine continuously processes information streams in an attempt to identify, and react to, meaningful events. An CEP bolt was implemented using ESPER [159], an event series analysis and event correlation engine (CEP). Using the ESPER EPL (SQL-like) query language we implemented several NetFlow analysis capabilities including network scan detection, identification of top talkers, top connections, highest transfer rates, lowest transfer rates, total flows per second, and total bandwidth per second.

Since individual components of Storm topologies, as with other cloud resources, can be instantiated in a number of locations, we must define endpoint destinations to receive data we can access. We developed a reporting bolt using a Simple Text-Orientated Messaging Protocol (STOMP) [160], which is directly consumable by web browsers using WebSockets [161]. WebSocket data pushed to web browsers allows users to observe CEP events as they occur in the application topology. An example of this type of reporting is found in the CEP "Top Talkers" bolt, shown in Figure

Table 5.3: Stream Process Rates

Source → Destination	*Processed Flows/sec*
AMPQ → Spout	318672
AMPQ → ResolverBolt	315208
AMPQ → DrainBolt	233864

5.2.

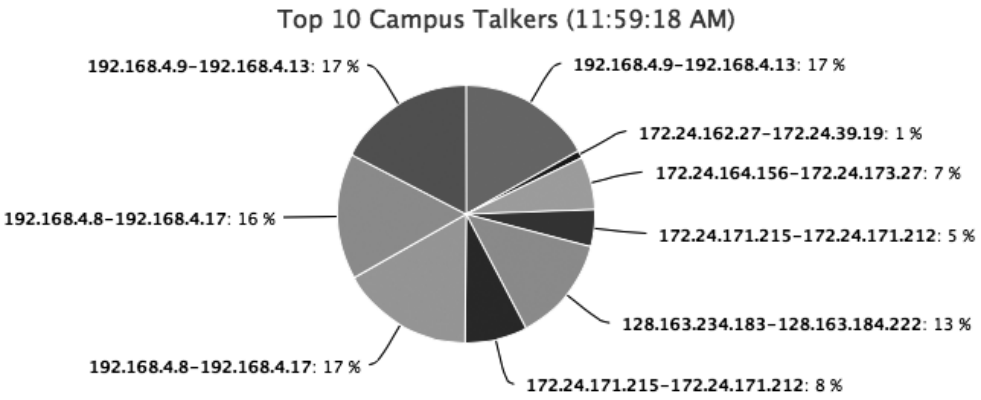

Figure 5.2: Live CEP Report

Both raw, enriched, and processed data streams are useful for offline analysis. Using HBase [151] we developed a bolt to drain steam processing data to an offline data repository. The performance of the described stream processing system is shown in Table 5.3. As shown in the table, the highest execution latency occurs in the Drain bolt, which is used to store enriched results in the HBase table. The next highest latency is the Resolve Bolt, which must calculate if the observed network address is located on campus and resolve the campus location from in-memory lookup tables.

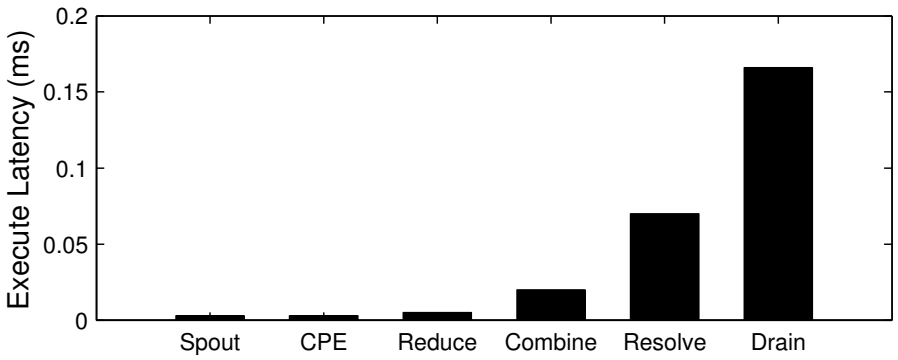

Figure 5.3: Topology Component Latency

5.1.4 Stream Processing in Cresco

Our work in distributed network and stream processing greatly influenced the development of Cresco. In the context of Cresco, probe devices act as independent Cresco Agents and Cresco Plugins, while AMQP servers function in a similar capacity to regional controllers. The resource and topology management capabilities of Apache Storm directly influenced Cresco Global controller operations and the need for the Cresco Application Description Language. Both Apache Storm and Cresco describe, implement, and maintain applications based on functional components arranged in topologies. However, Cresco functions at a much higher-level of abstraction than Storm. Storm components are represented as software code, where as Cresco components could be hardware or software represented through configuration and capable of a programatic interface. Storm topologies are typically implemented on clusters spanning a single geographic area, where Cresco is intended for geographic distribution. In fact, a Storm topology could be consider a Cresco component represented as a Cresco Plugin. Large-scale stream processing systems, like the one described can be rapidly developed using the Cresco framework.

5.2 Workload Characterization

The time-series utilization of computational resources such as CPU, memory, and data transfer can be used to characterize the behavior a wide-range of workloads. Workload characterization is in turn helpful in cyberinfrastructure planning, resource provisioning, and scheduling optimization.

We developed a workload characterization system to determine workload types and related resource needs for jobs running on the University of Kentucky "Lipscomb"[162] High Performance Computing (HPC) cluster. Using this system we characterized over 200 thousands jobs using 30 billion time-series metrics.

The HPC cluster is composed of commodity hardware and software, such as that found in both standalone servers and cloud computing clusters, and as a result characterization techniques developed in this work are broadly applicable across computational environments. The node-level and cluster-level attributes of the Lipscomb cluster are shown in Tables 5.4 and 5.5 respectively. Using the techniques described in this section we were able characterize workloads and develop a detailed view of resource utilization and scheduling patterns for our cluster.

Table 5.4: Node attributes

Node type	Count	CPU	Core	GPU	RAM
Basic	256	Intel E5-2670	16	-	64G
Hi-Mem	8	Intel E5-4640	32	-	512G
GPU	24	Intel E5-2670	16	2	64G

Table 5.5: Cluster attributes

Node type	Server	Core	GPU cores	RAM
Basic	Intel E5-2670	4096	-	16,384G
Hi-Mem	Intel E5-4640	32	-	4096G
GPU	Intel E5-2670	384	3584	1536G

5.2.1 Workload Collection

As with edge- and cloud computing systems, resource request are submitted to schedulers for provisioning. In the case of this work, job submission details were extracted from SLURM [163] and combined with resource scheduling information from MOAB [164], to form *job records* as shown in Figure 5.4.

- Key:
  - [*node_name*]-[*timestamp*]
- Value:
  - [*user_name*]-[*job_id*]-[*queue_name*]-[*job_status*]

Figure 5.4: Job Record Format

For resource utilization metric collection we used Ganglia [165], a popular metric collection system. Ganglia agents were deployed on HPC nodes which collected 41 metrics (cpu, memory, disk, network, etc.) every 15 seconds. The format of node *utilization records* is shown in Figure 5.5.

- Key:
  - [*node_name*]-[*metric_name*]-[*timestamp*]
- Value:
  - [*metric_value*]

Figure 5.5: Utilization Record Format

Due to the large number (billions) of utilization records, we used HBase [166], large key-value tabular database as our record repository.

5.2.2 Workload Collation

Job and utilization data were stored in separate unrelated data tables. In order to determine the resource utilization of a workload, we determine the active node(s) participating in a job and observe what resources are in use. Job records identify the active *job_id* for a specific node based on the record's *timestamp*. Utilization records provide resource usage data related to a specific resource *metric_name* for a specific *node_name*, based on the record's timestamp. During the collation process, job and utilization data are related by a timestamp range and node name key. Job records indicate the time a job was run on a specific node name, which is associated to performance information during the active job interval.

The relation of job and utilization data is established in three steps:

1. Job-node relationship: Job data is processed, resulting in a single file containing the starting and ending times of all jobs for all nodes.

2. Numeric-variable generation: Utilization data is processed, resulting in quartile and percentage variable for each metric, based on known fixed or minimum and maximum observed values.

3. Job-utilization relationship: Job and utilization data is processed providing raw, quartile, or percentage resource averages across all nodes for each job.

The format of job-to-node records is shown in Figure 5.6.

- Key:
  - [*node_name*]
- Value:
  - [*TS_start*],[*TS_end*],[*user_id*],[*job_id*],[*queue_name*], [*job_status*]

Figure 5.6: Job-to-Node Record Format

The output of job-node relation step is used in the job-performance relation step to group performance metrics for a specific job across nodes.

The job-utilization relation step uses data generated in the job-node relationship and numeric-variable steps to relate node utilization metrics to corresponding jobs. As with the node objects, the job record matching functions will be executed for each utilization metric, so an efficient method is needed. Within node objects, tree map structures are used to store job records. The tree map class is based on a special type of binary search tree called a red-black tree [167], which guarantees $O(\log n)$ lookup complexity, and more importantly allows us to quickly determine the floor (closest lower timestamp) and ceiling (closest higher timestamp) jobs for a specific metric timestamp. If a utilization metric is found to exist within a known job window, numberic varable data is used to recalculate the metric value, and the metric is emitted to the reducer with a key referencing its related job. The reducer phase calculates mean average of individual job metrics and merges results into columns of utilization metrics and rows of jobs.

The next subsection describes the method used to cluster workloads based on job-utilization data.

5.2.3 Workload Clustering

The general approach to data clustering in this section can be extended to a number of areas where distributed sources of data can be assembled centrally and used to develop systematic characterizations of workloads. The work in this section greatly influenced the Cresco Futura project described in Section 4.3.2.

In this section we will use job-utilization data to determine clusters of similar workloads based on job and resource utilization statistics. For calculations in this section, we discard any job with a runtime less than a minute or any job with fewer than five full (all attributes) metric samples. The resulting corrections reduced the number of jobs by 32%.

In batch-scheduler systems, jobs are submitted to specific queues. As a general rule, the longer the allowed wall clock time (WCT), the lower the resource assignment for the queue. Queue policy attributes for the Lipscomb cluster are shown in Table 5.6. Job-utilization data extracted from the job scheduler is used to calculate per-queue job statistics, as shown in Table 5.6.

Table 5.6: DLX Queue attributes

Queue Name	Node Type	WCT[1] Limit	Min-core	Max-core
debug	Basic	1 hr	1	16
PartNod	Basic	12 hr	1	15
Short	Basic	1 day	512	1024
GPU	GPU	3 days	1	265
Med	Basic	7 days	65	512
FatComp	Hi-Mem	14 days	1	32
gauss	Basic	30 days	16	16
Long	Basic	30 days	16	64

Table 5.7: DLX Queue statistics

Queue Name	Job Count	Ave WCT	Total WCT	Node Count[2]
PartNod	732	1.3 hr	0.03%	1
debug	6124	15 min	0.05%	1
FatComp	2817	14 hr	1.4%	1
GPU	4859	14.4 hr	4%	1.3
Short	41	7.5 hr	5.12%	1.17
gauss	11423	13 hr	5.4%	1
Med	1163	1.4 days	14%	10.4
Long	111519	12.4 hr	70%	1.1

On a fundamental level, cluster analysis algorithms are based on measurements of distances between metric values. We found that the selection of appropriate [168] clustering metrics greatly influenced the characterization of workloads. Initial clustering calculations included all 41 utilization metrics across all node types. However, strong correlations between metrics such as *pks_in* - *bytes_in* and *mem_free* - *swap_free*, along with variations of maximum and minimum metric values across differing node types, produced utilization clusters that were either non-useful or inaccurate. We

restricted our dataset to the 96% of jobs that execute on a *Basic* nodes. Limiting workload analysis to a specific node type allows for a uniform evaluation of resource utilization for a known window of job time as defined by the job-node relationship. In addition, we further restricted our dataset by eliminating metrics that from a utilization clustering standpoint would be considered duplications. Finally, we calculated new metrics to ensure each cluster metrics was related to resource utilization, not the lack of utilization (memory used vs. memory free) and estimated data generation metrics (Total bytes written) based on average metric rates and job runtime. The list of metrics used for utilization clustering is shown in Figure 5.7.

- *cpu_load*: Percent of computational load (utilization) [169]
- *mem_used*: Available memory - free memory.
- *job_write*: (Average disk write) * (runtime)
- *job_read*: (Average disk read) * (runtime)

Figure 5.7: Cluster utilization metrics

We selected the K-means [141] clustering method, which has been used ([170], [171]) successfully in workload characterization analysis. K-means cluster analysis requires the specification of the number of clusters to extract. Several cluster variable-selection heuristics have been developed [172] for k-means clustering, including the so-called "elbow" scree plot method. Using the scree plot method we selected between 4-8 clustering groups depending on the data set. However, in each analysis clusters of similar usage ratios were combined into three final clusters. Cluster profile values are based on maximum (1) and minimum (0) observed metrics across data sets. The job-utilization clusters for Basic queue nodes are shown in Figure 5.8.

Our cluster analysis identified *CPU*, *CPU + RAM*, and *IO* dependent workload profiles. Statistics related to our workloads are shown in Table 5.8.

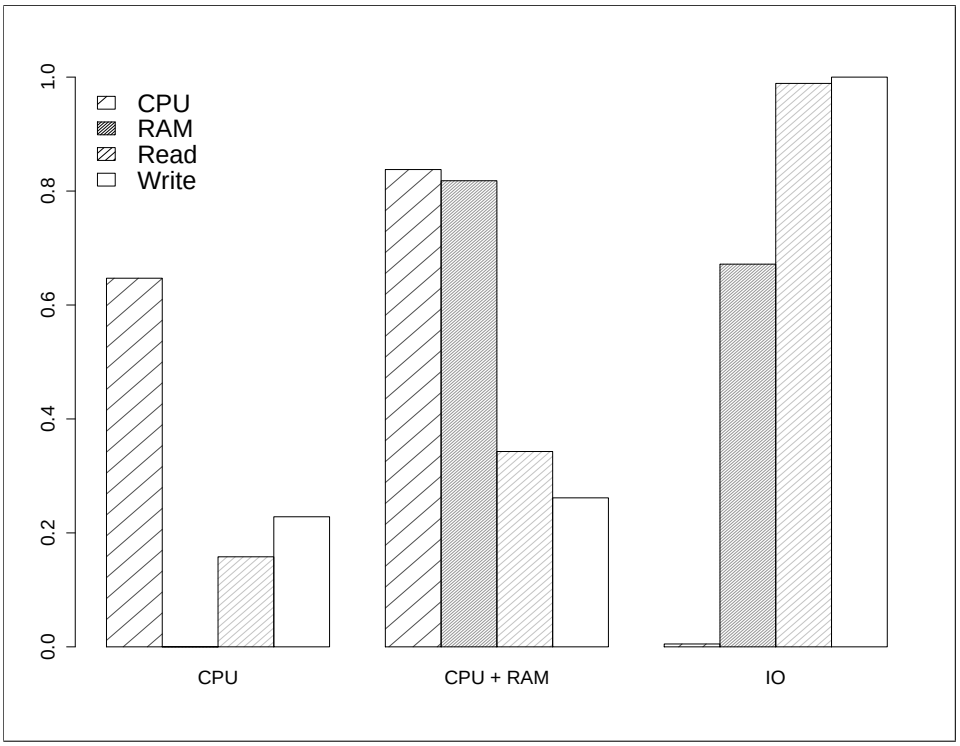

Figure 5.8: Basic queue node cluster profiles

Table 5.8: Basic node cluster statistics

Metric/Cluster	CPU	CPU + RAM	IO
job count	40%	44%	15%
cluster time	8%	79%	13%
cpu_load_ave	72%	88%	22%
cpu_load_max	250%[3]	1475%	60%
mem_used_ave	22%	88%	77%
mem_used_max	60%	100%	100%
total_read_ave	90MB	271MB	906MB
total_read_max	250GB	327GB	1.5TB
total_write_ave	2.3GB	2.6GB	16GB
total_write_max	3.7TB	1.1TB	22TB

In the following sub-sections we provide details of cluster analysis for each of the original three (CPU, CPU + RAM, and IO) workload profile groups.

CPU sub-clustering: The CPU sub-cluster profile is shown in Figure 5.9. Within the CPU profile we identified a *LOW CPU* resource sub-cluster, which accounts for 28% of the CPU cluster jobs and 35% of the runtime. This sub-cluster uses less than 48% of available CPU and 13% of RAM.

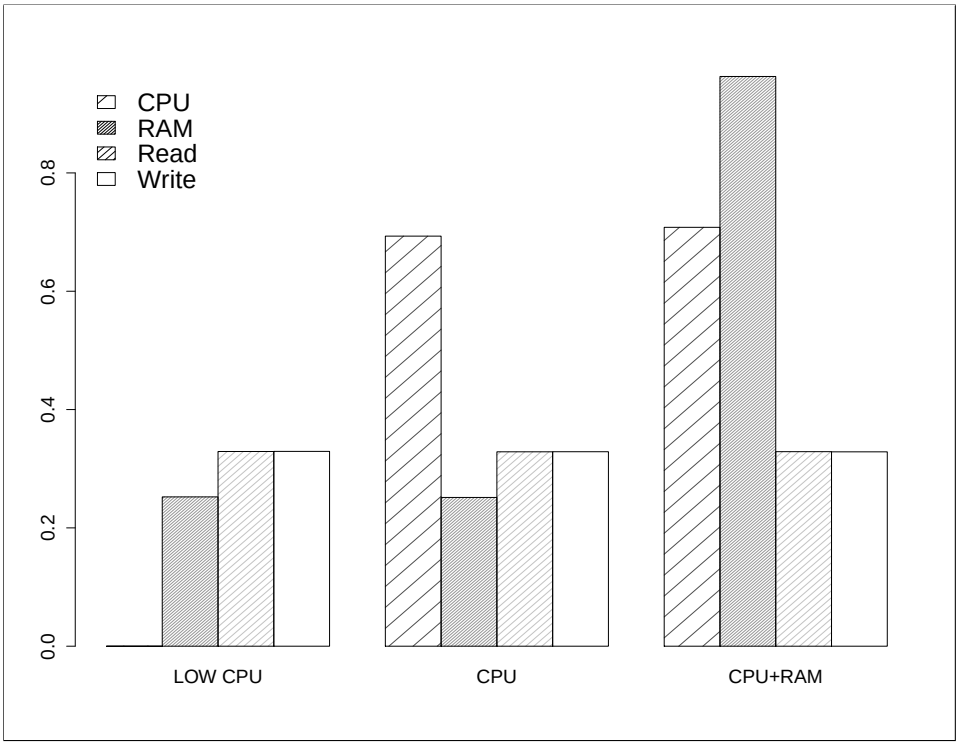

Figure 5.9: CPU sub-cluster profiles

CPU+RAM sub-clustering: The CPU+RAM sub-cluster profile is shown in Figure 5.10. Within the CPU+RAM profile we identified a *LOW RAM* resource sub-cluster, which accounts for 21% of the CPU cluster jobs and 8% of the runtime. This sub-cluster uses less than 67% of available RAM. In total, this cluster was shown to underutilize resources on 40% of jobs.

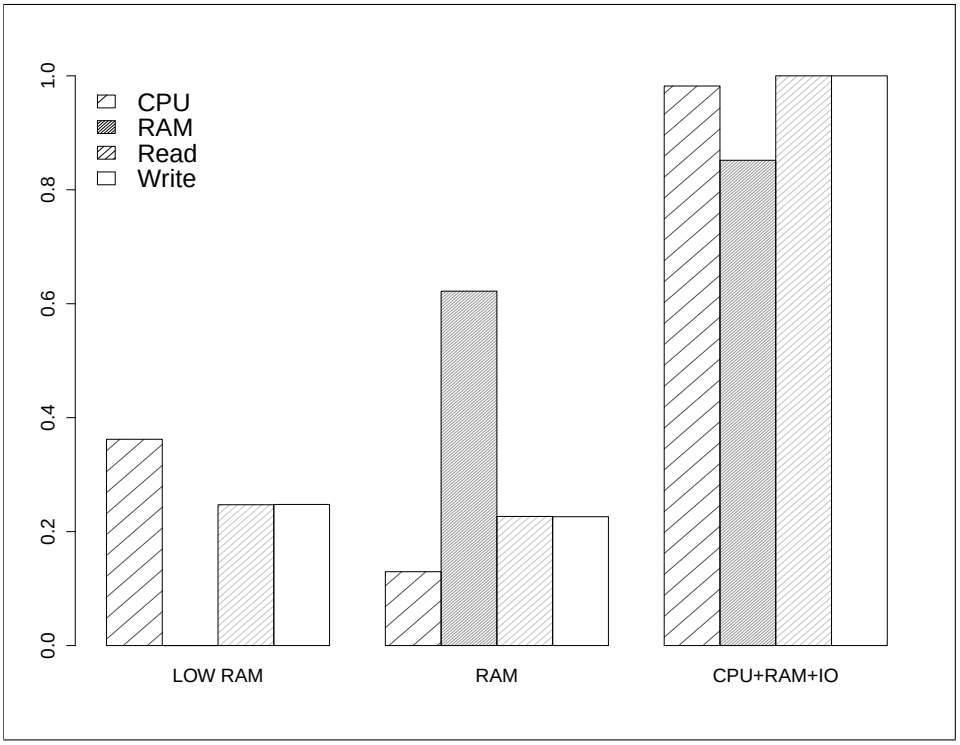

Figure 5.10: CPU + RAM sub-cluster profiles

IO sub-clustering: The IO sub-cluster profile is shown in Figure 5.11. Within the IO profile we identified a *LOW WRITE* resource sub-cluster, which accounts for 22% of the CPU cluster jobs and 16% of the runtime. This sub-cluster uses less than 6% of available CPU and 25% of RAM. In total this cluster was shown to underutilize resources on 32% of jobs.

5.2.4 Workload Characterization in Cresco

The analysis of HPC workloads suggest that all but a very few data-dependent and/or data-restricted jobs are candidates for alternative computational architectures. Highly computational multi-node and accelerator-based jobs make use of traditional local and national HPC resources, while single-node jobs are candidates for consolidation on virtual machines ([173], [174]), container-based environments [131], or unisolated process co-location [175] on physical nodes.

Based on continued increases in workload and computational diversity, future re-

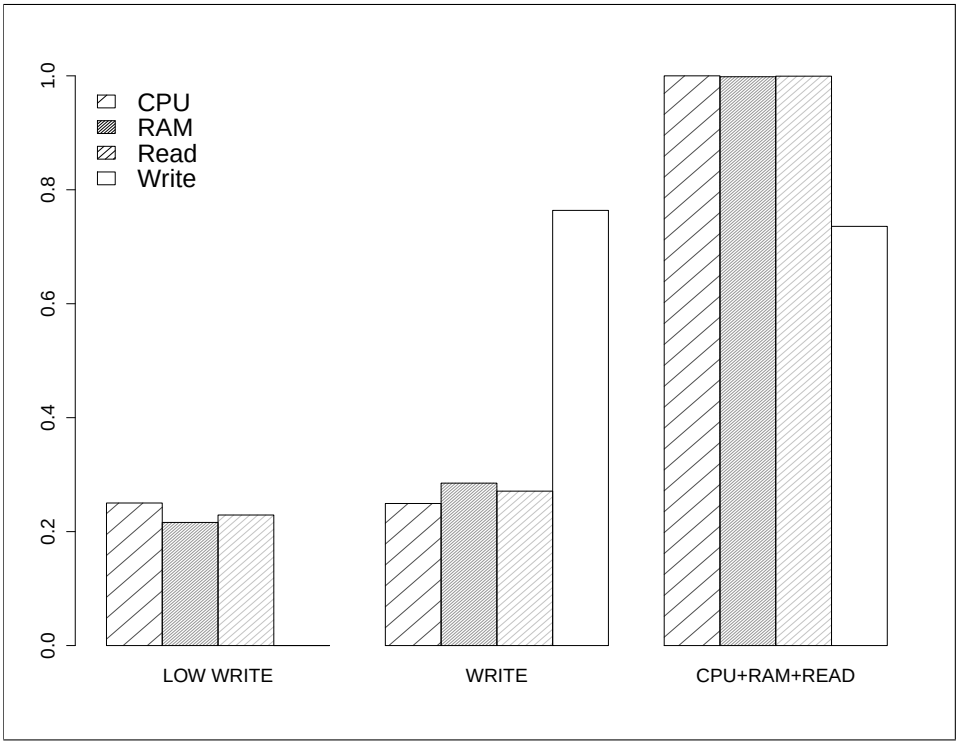

Figure 5.11: IO sub-cluster profiles

source scheduling systems, like those at use by Google [176], will need to support hybrid (batch and continuous) resource assignments across both general purpose and accelerated processing (GPU [177], MIC [178], FPGA [179], etc.) environments. In addition, the introduction of container-based [125] resource isolation provides the ability fine granularity manipulate resource allocations, even within a single computational node.

In the context of Cresco, we are also interested in workload prediction (Futura), resource utilization reporting, and resource scheduling optimization (Optima). While a number of existing techniques ([180], [181], [182]) have been developed for workload characterization, few make use of high-rate discrete metric collection and processing, or high-rate continuous workload assessment modeling, which in and of itself constitute a serious computational challenge. The lessons learned and techniques developed in this work directly contributed to workload characterization methods used by Cresco Global controller modules.

5.3 Genomic Processing Framework

Genomic processing is important to a wide range of areas, from fundamental research to applications in diagnostic medicine. Starting in late 2000s, advancements in second-generation DNA sequencing technology out-paced improvements in computational processing. The rapid acquisition of genomic data results in many terabytes of storage and tens of thousands of CPU hours of processing. As whole exome sequencing (WES) and whole genome sequencing (WGS) using next generation sequencing (NGS) technologies become common in diagnostic medicine and research, computational and ongoing storage costs increasingly become a larger portion of sequencing operational costs. Genomic processing for clinical purposes imposes even greater challenges related to operating environment validation, service level commitments (SLC) and data stewardship.

Genomic pipelines are often developed for specific processing environments and are typically unable to leverage resources available outside the designated environment. Genomic information generated by NGS is only part of the data involved in pipeline processing. Depending on the pipeline in use, hundreds of gigabytes, possibly terabytes, of additional data such as: reference, annotation, tools, and operating systems dependencies are needed. Due to effects of so-called data-gravity, a phenomena where data tends to reside in proximity to the point of initial processing, one typically finds pipelines either limited to local resources environments or restricted to platforms utilizing shared public cloud resources.

In collaboration with the University of Kentucky Medical Center we developed a flexible genomic processing framework for "write once, run anywhere" custom genomic pipelines, to be executed across a range of computational resources and environments. We believe that a wide variety of computational environments, determined by workload characteristics, should be used for genomic processing including HPC, cloud computing, and accelerated hardware (GPU, FPGA, custom ASIC, etc). The purpose of the work described in this section is to acquire data from genomic sequencer de-

vices, identify potential processing resources, and manage the end-to-end flow of data and processing encapsulating genomic pipeline processing within a Cresco-managed application.

5.3.1 Cresco-based Architecture

Genomic processing pipelines can be represented by a directed acyclic graph (DAG). In a pipeline DAG, nodes represent data sources or transformations, while edges represent the flow of the pipeline between nodes, as shown in Figure 5.12.

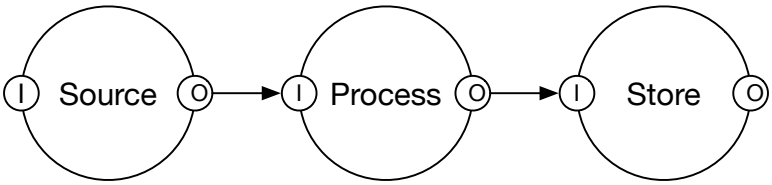

Figure 5.12: Pipeline DAG

The simple pipeline shown in the above figure is representative of a single phase pipeline, such as the conversion of raw sequence images to Fasta [183] records.

Nodes represent context-specific functional workloads, such as data transformation, enrichment, processing, etc., edges are points of data exchange. Specifically, nodes represent functional modules that provide sets of tools and data sources used in genomic processing. Nodes can act as both sources and destinations for flows of data, as defined by the corresponding directed edges. Input and output interfaces (areas where edges connect), shown in Figure 5.12 as I and O, for each node type are defined by their data schema. In order for the output interface of the *Source* node, $Source_O$, to be compatible with the input port of the *Process* node, $Process_I$, the data schema of the two interfaces must be compatible, including any security restrictions inherited from upstream data flows.

Cresco provides underlying resource management, component communication, job monitoring, and performance measurement for the genomic processing framework.

The Genomic Control System (GCS) makes use of the Cresco Application Description Language to generate user-defined pipeline graphs, which in turn are used by Cresco to implement and execute pipelines. Some genomic data, especially in the areas of diagnostic medicine, must be tightly controlled from a data access standpoint. As per the Cresco framework, all agent communication is encrypted, with explicit access controls for each participating agent. GCS receives state information from agents operating on resource nodes, which describe capacities (CPU count, available memory, etc.) and current operational status. When work arrives for a specific pipeline the GCS assigns workloads to resources agents. If no resource is available, the GCS can request additional resources by running an agent on a wide-range of dynamic HPC or cloud environments. Once initialized, the agent contacts the GCS to advertise its capabilities. On workload initialization, the agent adjusts tool parameters described in the pipeline manifest to match its discovered capabilities, thus providing tuning on an instance runtime level. During workload execution, the agent sends both resource utilization and process status data back to the GCS. Through the analysis of resource cost and utilization, in-conjunction with runtime information, best-fit resources can be determined.

In the next section, the computational, network, and storage environments managed by Cresco as part of the genomic processing framework are described.

5.3.2 Cresco-managed Environments

Cresco manages a number of environments and technologies at the University of Kentucky in relation to the described genomic framework including VM, Container, storage, and networking.

Containers Application container technology [119] is extremely attractive for use in genomics processing. From the prospective of the application execution environment, a container running on a laptop is the same as a container running in a HPC or cloud

environment. Phases (nodes) of pipelines are implemented as Docker containers in this framework. Genomic container images contain all system-level and application-level dependencies required to run one or more genomic applications. Each time a container layer is changed, a new registration is required for that image, so registered genomic container images provide a unique reference related to specific application versions. Registered genomic container images are used across all genomic processing environments in the framework. Container operations are managed as Cresco Plugins.

Data written to a container during application execution is non-persistent, so pre-emption of container operation results in the loss of any data generated during a specific execution of the container. However, if an attacker has access to a node running a container, the current state of the running container can be registered (saved) as a new image. To prevent data loss as a result of creating a new container, the filesystems (within the container storing restricted data) are encrypted with private keys that reside only in container memory. Keeping filesystem encryption keys in memory prevents access to restricted data if the container was restarted, such as the case if a container was registered as a new image. Encryption is enforced on any container-network communication, preventing data loss through traffic "sniffing" [184]. In addition, container communication is restricted to isolated nodes, which limits data loss through controlling communication end-points. Containers make use of namespace isolation, which means that application processes and memory running inside of a container are protected from other containers as well as the host, which runs in the default namespace.

Despite controls limiting the interaction of unrestricted and restricted workloads, the encryption of data-at-rest and data-in-transit on containers, and restriction of container communication to trusted sources, container compromise is still possible. To limit the impact of compromised containers, single-use tokens are issued by the pipeline controller to containers for every genomic data-set transaction. Additionally, a new container will be executed for every new genomic data-set transaction. This

policy ensures that a compromised container can, at most, expose its workload, but not the workloads of other containers, or a broader collection of restricted data.

While containers are ideal for providing reproducible application environments, current container storage architecture is not well suited for large (>1Tb) volume management. More importantly, we don't want to have to move hundreds of gigabytes of reference data for each container execution or be restricted to locations where reference data already exist. Along with container-based versioning, we also provide methods to package, retrieve, and validate additional tools and data required to run custom pipelines.

Object Storage Object-based storage is used for the management of raw and processed genomic data. Object-based systems allow for the control of data on the object-level (per sequence or result), they allow meta-data including additional auditing and security controls to be part of the data management environment. In addition, object-based storage allows us to more tightly control data protection policies by providing replication.

While object-based storage is ideal for storing raw genomic and processed result data, few applications make use of object-based storage directly. Cresco Plugins manage filesystem and object interactions. It is worth noting that just as with computation environments, both local and public cloud storage can be used by our framework.

With abstracted execution (Containers) and storage (Object) environments, the framework is capable of scheduling work in a number of environments, as discussed below.

HPC clusters The use of HPC clusters is common in genomic and bioinformatic processing. There are two common approaches taken to address the processing of restricted data on HPC clusters:

- *Physical-level isolation*: A physically isolated HPC cluster dedicates computational, storage, and network resources to restricted workloads. These, often physically-isolated, HPC clusters are purpose-built and used for the exclusive processing of specific types of workloads, such as genomic pipelines. In this model, tightly controlled security measures can be implemented around points of data ingress and egress, since users and workloads in the system are considered trusted. Typically, users of these systems must agree to usage policies where all users are responsible for their own data, as well of data of others if it is inadvertently encountered in the trusted system. The primary benefit of this model is the ability to restrict communications to known trusted sources. The primary drawback of physical-level isolation is that unused resources that might otherwise be shared with unrestricted workloads are unavailable for assignment.

- *Shared*: In shared HPC cluster deployments restricted and unrestricted workloads are run on the same physical infrastructure and operators rely on usage policies and security controls to mitigate risk of data contamination. As with physical-level isolated systems, users of these systems agree to usage policies that make the user responsible for managing their own data as well of any other restricted data they might encounter. While it is possible that compensating controls like queue-level separation of workloads are implemented, typically these systems don't distinguish between unrestricted and restricted workloads. With shared HPC systems control of points of data ingress and egress is much more difficult since HPC operators have no *a priori* knowledge of unrestricted data transfers. The primary benefit of this model is that all available resources can be used for both restricted and unrestricted workloads. The primary drawback of this system is that while well-defined usage policies and violation notifications limit liability risk, policy in and of itself provides no technical controls to prevent policy violation.

While this framework could be used for physically isolated HPC clusters, the privacy preserving aspects of the framework provide the most benefit in shared HPC environments. As mentioned above, shared HPC systems traditionally do little or nothing to separate restricted workloads from unrestricted workloads. The primary risk in mixing workload types, and by association workload data, is that global constraints can't be applied to a shared HPC that both simultaneously protect restricted workloads, while providing open access to unrestricted workloads. Based on this observation, we must conclude that from a shared HPC administrative-level (scheduling, accounting, storage, etc.), restricted and unrestricted workloads can not and will not be differentiated. In place of system-wide policies, we must focus on protecting restricted workloads on the workload level. This is accomplished by isolating restricted workloads through the use of container technology managed through Cresco Plugins.

Cloud computing HPC environments are typically comprised of highly-connected physical hardware nodes, with many processor cores. However, there are some genomic processing steps that are serial in nature and only make use of a single core. When a multi-core node is used for single-core (serial) operations, the rest of the cores sit idle, thus wasting potential processing power. One approach to dealing with serial operations is to execute single-core pipeline processes on virtual HPC nodes. Virtual HPC nodes share underlying physical resources and provide process isolation on the machine-level. For instance, two virtual HPC nodes running on a single physical server could process two separate serial processes simultaneously. In addition, since isolation is taking place on the machine-level, unrestricted and restricted workloads can run on the same physical machine in two logically separated virtual machines.

As with physical HPC environments, containers managed by Cresco are used within virtual environments to manage data flow and processing.

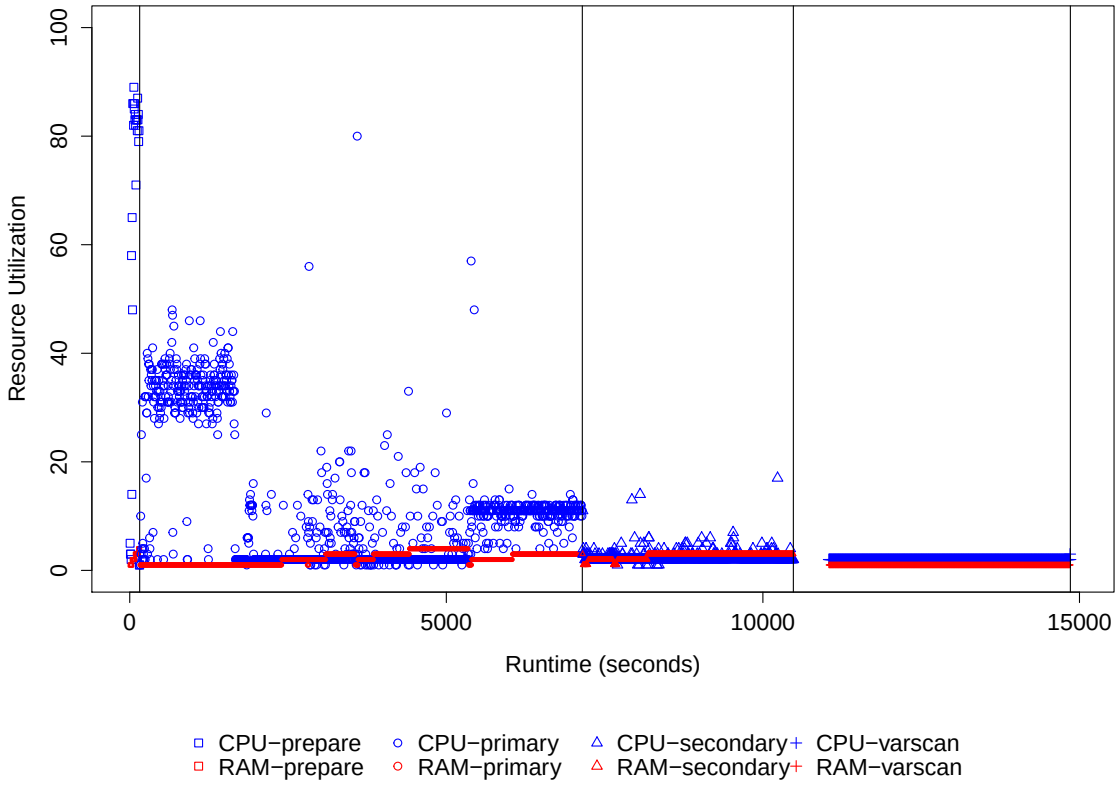

Figure 5.13: Pipeline processing environments

5.3.3 Cresco-managed Operations

In this section we cover genomic processing steps managed by the GCS and related Cresco components.

- *Genomic object generation*: Raw data from genomic sequencers is stored on a network-attached client workstation. A Cresco Agent is deployed on sequencer workstations and a genomic processor plugin is used to monitor the workstation filesystem for the start of sequence generation. Once a new sequence is observed the Cresco Plugin registeres the sequence with the GCS and awaits output completion. Once complete, the plugin generates an MD5 [185] hash of all files generated by the sequencer and transfers the data to the object storage system.

- *Pre-processing*: The GCS notifies a pre-processing node that a new sequence is

available and provides information (location, credentials, object id, etc.) needed for data transfer. The pre-process node transfers and MD5-verifies the sequence data, then notifies the GCS that the sequence is ready for pre-processing. Based on bioinformatician parameters, sequences are broken into samples and corresponding sample configurations are generated. The samples and configuration are then transferred to object storage and the GCS is notified that samples are ready for processing.

- *Processing*: The GCS notifies the processing agent that a sample is ready for processing and, similar to pre-processing, data is transferred to the processing node. The agent on the processing node reads the configuration manifest and begins environment validation. Resource bundles specified in the manifest are download from object storage, if needed, and verified. The specified base genomic container is pulled or cache is verified from the container registry. Once the container is active and the specified environments has been validated, the agent adjusts runtime parameters of genomic tools in the pipeline based on available resources, and starts processing. Using Cresco KPI, communication resource utilization paramaters, as shown in Figure 5.13, and process status are communicated to the GCS for the duration of execution. Using KPI indicators the efficiency and cost effectiveness of various environments can be determined through comparisons of resource utilization, job execution time, and environmental cost.

- *Post-processing*: Once all of the samples have been processed, the GCS directs resulting data to automated post-processing, an interactive genomic workspace, or to an external application or storage.

5.3.4 Cresco-based Genomic Framework

Our Cresco-based genomic processing framework provides many of the advanced features of commercial cloud-based frameworks, with the added flexibility of edge (local, cloud, or hybrid) operation based on workload needs. In fact, while our current implementation manages custom pipelines, projects that already provide containerized images can be used directly within our framework.

5.4 GLobal Edge Application Network

Testbeds provide environments for replicable testing of computational hypothesis and techniques. Testbed environments are widely used in the development and demonstration of distributed network and communication systems. The Global Environment for Networking Innovation (GENI) [186] and Future Internet Research and Experimentation (FIRE) [187] are two of the most prominent next generation network and innovation testbed projects. The GENI and FIRE projects allow for "at scale" network experimentation using a network of globally distributed edge resources. In efforts to address the specific IoT experimentation needs [188], a number of independent IoT testbeds [189] have been deployed in federation with FIRE across Europe. The majority of these IoT specific testbeds focus on lower-level device data collection and typically range in scope from single buildings to city-scale (Smart City) experimentation. GENI resources (GENI racks) are deployed at universities and cities around the world, which results in the potential for broad geographic experimentation. In the US, IoT and Smart Cities applications, like those developed as part of US Ignite [190] efforts, are often deployed on GENI resources. However, at the time of this writing there are no specific Cyber-Physical Systems (CPS), Machine to Machine (M2M) technologies, Industrial Internet, or Smart Cities project federations with the GENI network as there are with FIRE. Perhaps of greater concern, testbeds are generally considered research environments, where service is maintained

on a best-effort basis. While testbeds are suitable for experiments, they are subject to resource failure and are not appropriate for mission-critical applications, such as those used in the management of city, regional, or national operations. In the absence of resilient cyberinfrastructure, software frameworks can be used to mitigate failures and manage the resource needs of applications.

Below we discuss the GLobal Edge Application Network (GLEAN), a distributed network of managed resources found on the edges of networks and central data centers. GLEAN operates across a number of environments including standalone servers, distributed testbeds, and cloud computing resources. The network is specifically designed to address the challenges of IoT data collection, analysis, monitoring, and measurement across islands of edge and data center resources.

As previously described, the following items must be addressed in order to use existing testbeds as production edge application environments; a) Stability of computing and network resources; b) Management a large number of objects; c) Quality-of-Service (QoS) enforcement of resource reservations; d) End-to-end monitoring and measurement of resources; e) Machine-to-Machine (M2M)-focused operations; f) Simple deployment of durable applications. While many other requirements are specific to IoT operations, the previous list is limited to addressing suggested limitations of existing underlying deployments.

A gap exist between production cloud-based IoT frameworks, which are often focused on consumer devices, and testbed IoT edge (city, building, etc.) frameworks, typically focused on Cyber-Physical System (CPS), Industrial Internet, and Smart Cities. While GLEAN does not claim to close the gap between IoT frameworks, we do offer a host of changes to be made to in conjunction with existing edge environments and architectural principles for new classes of globally-connected infrastructures.

The remainder of this section will cover GLEAN: A GLobal Environment for IoT Edge Computing, which is experimentally deployed on a 5 region (25 node) testbed at the University of Kentucky.

5.4.1 GLEAN Architecture

While the service-level requirements of production environments differ from experimental testbeds, the GLEAN environment can be deployed on the existing GENI or similar environments. Using software-defined provisioning and networking [53] capabilities of the GENI network we can stitch together islands of low-level edge resources, which from the infrastructure standpoint are indistinguishable from standalone resources. The remainder of this section describes a system that can work from within, or adjacent to GENI, depending on the desired service-level requirements. The Cresco framework is used to manage GLEAN resources and operations across local, regional, and global domains. As is common with software defined systems (including GENI), GLEAN is divided into separate control and application planes. While the Cresco framework is used in the control plane, additional IoT frameworks and/or federations can be used on the application plane.

Stability of computing and network resources Testbed environments provide low-level access to physical and virtual infrastructure, which is needed for experimentations, such as protocol and device development. In testbeds such as GENI, provisioning of low-level infrastructure must be coordinated between heterogeneous hardware and software implementations, which could be geographically distributed around the globe. The testbed scheduling service has little or no information pertaining to the operational state of the underlying system. This type of high-level scheduling of low-level geographically distributed resources is very different from the way cloud providers, such as Amazon EC2, provide resources. For example, a cloud provider has complete control over their underlying infrastructure and software stack used in the provisioning of virtual resources. In comparison, GENI must manage the low-level stitching of communication paths through Internet2 (I2) in conjunction with local (campus) networks, and the provisioning of computational resources across

heterogeneous environments. These scheduling practices, while necessary for experimentation with testbed environments, lead to high-rates of provisioning failures. In GLEAN we focus on providing environments for edge applications, not underlying infrastructure, so once core underlying resources have been provisioned they remain online as long as the framework is active. Applications share underlying low-level resources managed and monitored by GLEAN. We are not suggesting that access to low-level hardware (network devices, sensors, etc.) is not needed, but rather that from an edge computing prospective access to these devices can be gained through external IoT gateways, lower-level framework federations, or directly over higher-level protocols.

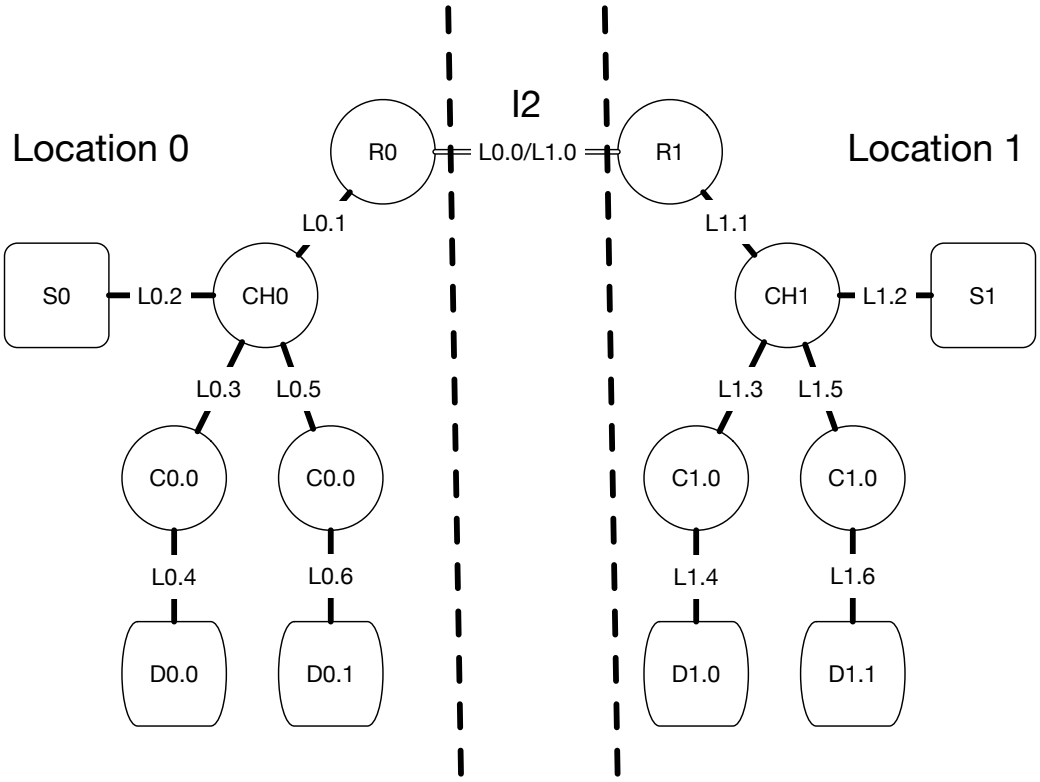

Figure 5.14: GLEAN sites connected over Internet2

Figure 5.14 shows two GLEAN sites connected over a low-level connection through I2. The link *L0.0/L1.0* represents a Data-Link layer (L2) connection directly between edge routers $R_i, i = 0, 1$. Each edge site provides one or more computational resource

providers *CH* with optional storage *S* resources. Links *LX.1* represent L2 communication between edge routers and computational/storage resources. Container-provided or -managed resources are represented by *C*, where links *LX.4 & LX.6* represent several possible communication methods including, but not limited to, native IPv6 container endpoints, IPv4 tunnels over IPv6 networks between containers, or other protocols and transport mechanisms implemented in conjunction with *CH* resources. Devices, represented as $D_{x.y}$, can be directly accessible globally or serve as data sources for edge gateways and/or higher-level processing functions. Figure 5.15 provides an example of multi-transport communication between two endpoint devices managed by GLEAN.

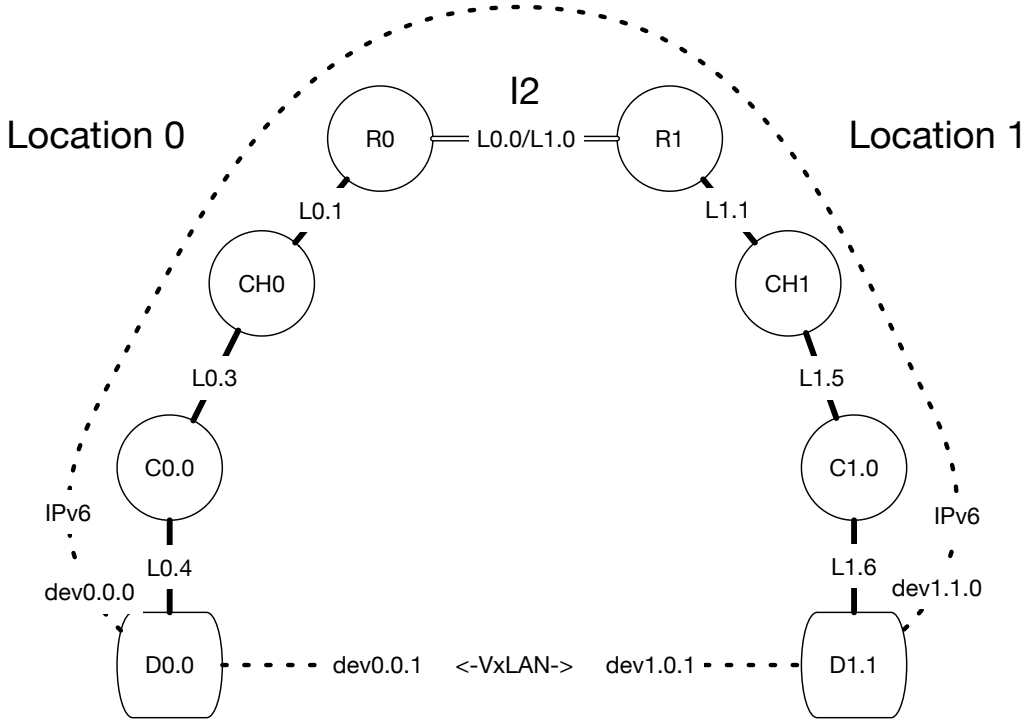

Figure 5.15: Tunnel between regions

Management of a large number of objects The first step in managing IoT objects is in network communications. However, there are already over twice as many devices in China alone (9 billion as of 2014) than there are total IPv4 network addresses [1]. Thus, GENI and many other testbeds that are based on IPv4 addressing,

will not be sufficient to address a large number of devices. In addition, IPv6 support for public cloud resources has been limited to specific regions and services or completely unavailable. GLEAN operates in a "dual stack" allowing communications on IPv4 and IPv6 networks. GLEAN can be used to translate messages and address resources between IPv4 and IPv6 accessible resources. While GENI might not support IPv6 addressing of its resources, it does provide lower-level network resources, which allows us to communicate using IPv6 between GLEAN instances. For instance, two GENI virtual machines in separate geographic locations might use publicly accessible IPv4 addresses to communicate with the outside world. However, these same virtual machines might also have a direct link-layer connection between each other or another endpoint, which can be used to communicate using IPv6. In GLEAN, the control plane and associated network overlays all operate over IPv6 taking advantage of the Quality of Service (QoS), security, and large address range features of the IPv6 protocol. Application layer services can operate over IPv4 or IPv6, depending on requirements and availiability. IPv6 allows us to efficiently assign and route billions of addresses to individual edge resources. In Figure 5.14, routers designated as *R0* and *R1* are directly connected via L2 link. These software[4] routers run Bird [191] Border Gateway Protocol (BGP) daemons and are capable of propagating IPv4/IPv6 routes between other GLEAN sites and external networks. Figure 5.16 shows an example GLEAN site network.

As shown in the previous figure, a */56* IPv6 range is advertised externally by the site. For each compute host *C* a */64* range is assigned, providing 2^{64} addresses for each resource-providing or managing host. For each application container *D* a */128* address is assigned, which allows the application container to natively communicate with IPv6 networks. Additionally, */64* and */128* addresses can be assigned to external IoT gateways and directly to devices.

Resources and devices functioning as part of GLEAN are managed by Cresco,

[4]The software routers can be replaced by hardware routers if needed.

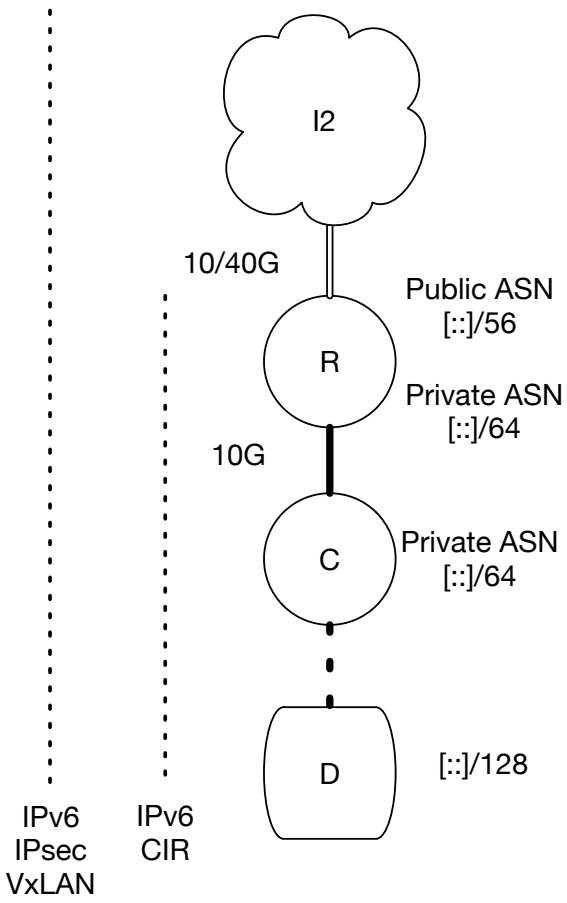

Figure 5.16: GLEAN site network

which is capable of addressing large numbers of devices.

QoS enforcement of resource reservations Testbeds are able to provide complex resource reservations, but lack the global ability to enforce resource-level Service Level Commitments (SLC). In some cases, such as with I2 AL2S links [192], the underlying infrastructure does not support QoS controls necessary to satisfy SLCs. However, from the edge computing prospective, we can control both compute and network resources to a high degree of detail. In addition, while we can't yet[5] guarantee bandwidth between I2 connected sites, we can control the priority of traffic between sites. The Linux kernel's network stack provides native network traffic control and shaping features. QoS policies can be implemented from routers to containers, as

[5]QoS features are on the I2 AL2S roadmap

shown in Figure 5.16, and noted by *IPv6 CIR* (Committed Information Rate) [193].

QoS policies can be implemented on software and hardware routers connecting sites, either through static classes or dynamic reservations. Through the use of Linux kernel namespace isolation and resource control groups (cgroup) we can impose resource imitations, prioritization, accounting, and control of application containers. While similar limits can be imposed by hypervisors managers for entire virtual machines, cgroups management allows for the process-level control of applications. For example, specific components of the Cresco control system can be given system-wide priority of compute and network resources, as needed for pseudo real-time system control. Likewise, specific virtual machine resource equivalencies can be assigned and enforced for application containers. With the assignment of explicit minimum resource allocations across edge resources, we can enforce SLCs of resource reservations. Dynamic QoS operations will be managed by Cresco agents.

End-to-end monitoring and measurement of resources End-to-end monitoring and measurement of federated resources used in experimentation and distributed applications is a challenge. While high-level objects like provisioned network and compute resources are available, low-level monitoring and measurement of underlying edge resources and related networks are either not available or specific to underlying federations or service offerings. For example, an application provisioned between two sites might use resources provided by different federated compute projects, with differing and possible unavailable low-level resource monitoring capabilities. In addition, data related to the state of the physical network(s) providing connectivity between sites might also be unavailable. In GLEAN we can deploy Cresco agents for resource-providing systems to verify operational status, including verification of SLCs. In addition, lower-level infrastructure performance information can be made available in conjunction with application-level performance information, allowing for the correlation of edge reservations to global application performance.

M2M-focused operations In testbed and cloud computing infrastructure environments resource topologies are either requested as independent resource items or as collections of interconnected systems. For instance, one might use Amazon EC2 to provision one or more independent virtual machines. Likewise, a researcher might use GENI network to provision a multi-site topology connecting computational resources running specific software by means of the *Rspec* [194] description language. In both of these cases resources are described and provisioned statically through central control services. In an edge-focused environment, infrastructure management must not only respond to application-level changes, but it must anticipate, coordinate, and implement SLC-driven changes dynamically, based on direct application interactions. For instance, a overloaded edge at site A must be able to intelligently interact with edge site B and cloud site C to determine appropriate workload offloading, based on observed workload characteristics.

The GLEAN distributed control plane is based on the Cresco hierarchy of distributed agents. Agents operate autonomously and are capable of dynamically developing operational topologies through M2M discovery processes. Every agent can communicate with all other agents through a protocol-independent communication hierarchy. Agent communication is restricted independently at each level of the hierarchy, based on group security policies. Agents are responsible for application components and resources, while Regional controllers are responsible for operations in their region, and global controllers manage regional controllers. Provisioning is accomplished through M2M-focused resource requests distributed throughout the global environment. Global requests are filtered by input predicates and best-fit matching of resource to workload is pushed down to regional and agent-levels.

Simple deployment of durable applications Application platform services, like those provided by public clouds, abstract the underlying details of infrastructure from application developers. If an underlying infrastructure component fails on the plat-

form, workload and data is reassigned to healthy resources. For location independent applications, such as websites, a platform abstraction where the underlying service requirements determine workload placement is attractive, since we can move the data to the locations where resources are available. However, in the realm of edge computing we often want to selectively determine where workloads are processed, thus moving computational resources to sources of data.

In contrast to cloud platform services, testbed resources (GENI) are explicitly assigned by users using a topology description language "Rspec". While users can provide custom images and describe complex resource topologies, there is otherwise no abstraction from the infrastructure level. Unlike cloud platforms, testbed provisioning systems typically do not detect and reassign resources on infrastructure failures.

Applications deployed on cloud platforms lack edge computing control and testbeds lack infrastructure abstractions and resiliency that simplifies the deployment of durable applications. GLEAN aims to bring platform-like abstractions of infrastructure to edge computing environments, including but not limited to, testbeds. While the details of the provisioning process are outside the scope of this section, as with the GENI Rspec, GLEAN uses the Cresco Application Description Language (CADL) to implement and maintain application topologies. However, while Rspecs resource assignments are typically prescribed, with GLEAN, resources are assigned through both predicate filtering and best-fit scheduling. For example, data collection services are pushed to specific locations as predicated by description, while higher-level processing can be assigned, and in the future reassigned, to an adjacent edge or cloud service. In addition, the current implementation of GLEAN makes use of both public and private container registries. Public registries are typically used to provide source containers for applications. Private registries are used as both application sources and container snapshot targets. The description of the application along with the ability to snapshot[6] existing deployed applications, allows GLEAN to redeploy application

[6]Snapshots provide disk and configuration data only.

components or entire topologies in the event of infrastructure failure. In addition, where permitted by predicate assignment, workloads can be reassigned as environmental variables change. For instance, location independent workloads on a specific edge can be migrated to cloud resources as additional local resources are needed.

5.4.2 Related CPS Environments

As previously mentioned, there are examples of CPS and IoT related edge computing on existing GENI and FIRE international testbeds. While these IoT efforts might operate on international testbeds, most projects typically focus on smaller scale testing.

City-scale IoT-focused testbeds such as SmartSantander [188] and others [189] also exist. A number of IoT efforts focus on the deployment of IoT hubs to support diverse communication protocols and large numbers of devices on a building-scale, such as FIT IoT-LAB [195] and The IoT Hub [196].

Components of both existing testbeds and IoT-specific efforts complement GLEAN. Existing federation services and low-level resources are used by GLEAN to deploy globally distributed edge services, which can include existing city- or building-centric IoT hubs.

5.4.3 GLEAN Conclusions

Global testbeds lack the production quality service aspects of public cloud computing offerings. Conversely, public cloud computing offerings lack the edge computing resources offered by testbed resources. Both cloud and testbeds typically lack the ability to directly address, manage, and access very large numbers of devices. In addition, neither testbeds or cloud offerings provide end-to-end monitoring, measurement, provisioning, and migration of services between edge and cloud resources. Existing IoT efforts (Hubs, IoT-testbeds, etc) typically focus on low-level device communication and are limited to city- or building-centric deployments.

We have described how GLEAN, and thus Cresco, can be used to provide a global environment for IoT edge computing. GLEAN bridges the gap between existing global infrastructures and existing IoT efforts by addressing issues related to a) Stability of computing and network resources, b) Manage a large number of objects, c) QoS enforcement of resource reservations, d) End-to-end monitoring and measurement of resources, e) M2M-focused operations, and f) Simple deployment of durable applications.

6

Building a Smart City Application

In this chapter we describe a process of developing a Cresco application to solve a hypothetical problem. While there are a number of existing real-world Cresco applications, including those described in Chapter 5, *Case Studies*, we want to describe the potential use of Cresco in large geographically distributed edge applications. Cresco serves as a potential framework to solve a number of challenges required for the advancement of Smart Cities. We will describe the process of using the Cresco framework in the context of a hypothetical Smart City application used to manage city-wide distributions of sensor arrays and vehicle data. Sections 6.1, *Application Requirements* and 6.2, *Application Design*, are written as if the hypothetical application was to be fully developed and implemented. The remaining sections describe how we implement and operate the a subset of the described hypothetical application to demonstrate Cresco in this context.

While the source data and resulting analysis data will be simulated, the application will make use of Cresco as if it was fully developed and implemented in a production environment. The described application makes use of many of the advanced features of the Cresco framework and aims to highlight the use of Cresco in an edge computing environment.

6.1 Application Requirements

As mentioned in Chapter 1, *Edge Computing Introduction*, Smart Cities applications such as traffic and environmental sensor management require data processing on street-intersection, neighborhood, and city-wide levels. Potential data sources include distributed sensor arrays, vehicles, and personal devices. Edge resources might be used for data interoperation, processing (analysis) services, and the coordination of information, such as autonomous Vehicle-to-Vehicle (V2V) interactions. As previously mentioned, a single autonomous car is capable of generating four terabytes of data daily, which places serious demands on infrastructure and software systems supporting Smart City efforts. Likewise, the coordination of millions of potential sensors and devices in a large metropolitan area is a serious computational challange.

While there are no accepted standards for infrastructure supporting Smart City efforts several trends have emerged, which will guide our application requirements. Cities like Chicago are deploying general purpose sensor arrays to "track the city's vitals" [59]. These sensor arrays are often deployed in conjunction with existing light poles, which provide power and often network connectivity for street-level cameras. Assuming this trend continues we should expect to see bidirectional wireless communication from distributed "processing poles" (PP), which can be used to communicate with end-devices such as vehicles or personal devices. PPs will likely provide sensor and device gateway functions (communication, data exchange, filtering, etc.) for street-level services areas. As previously mentioned, an attractive location for neighborhood-level data aggregation and intermediate processing of PP data is within telecommunication central offices (CO). Projects like CORD [36] provide Central Office Processing (COP) services from computational clusters distributed around city COs. COP might include analysis services to route traffic, notifications of higher-level city services of street-level problems, and additional analysis services only possible through the coordination of data from multiple PPs. While COPs might maintain a subset of city-wide data, such as current street-level traffic data, they need not

maintain time-series logs of data from their or other COP regions. Aggregations of neighborhood-level COP data and related processing will likely be maintained in large computational clouds found within cities or other geographic areas. As data is propagated from PPs (street-level) to COPs (neighborhood-level) and finally to CP(s) (city-wide), computational capacity increases as does communication latency. This inverse relationship between potential analytic capacity and communication results in the generation of computational models on higher-levels of process hierarchy and the execution of models on lower-levels. For example, aggregations of street-level data summaries provided by COPs can be used by CPs to generate weighted graphs pertaining to city-wide traffic status. City-wide traffic status might then be propagated to all COPs for use in neighbor-level traffic route calculation. Likewise, images and related sensor data generated by PPs might be propagated to CPs to generate image models to be implemented on PPs. Computer vision models on PPs might be used to determine events such as if an intersection is clear of snow or to detect if an accident has occurred. A potential city-wide data path topology between devices and processor is shown in Figure 6.1. In the figure, SAs represent sensor arrays and Vs represent vehicles. Sensor arrays are uniquely identifiable for each PP, while vehicles and other transient devices might be observed by one or more PPs. Data exchange between end-devices and PP is defined as *PP-Device Data*, data between PP and COP is defined as *COP-PP Data*, data between two COPs is defined as *Inter-CO Data*, and data exchanged between COP and CP is defined as *CP-COP Data*.

An application used to support the described Smart City will need to provision workloads on a number of devices over a distributed geography, manage the flow of data between workloads, and exapnd or contract provisioned resources based on sources of data, results of analysis, and resource requirement demands.

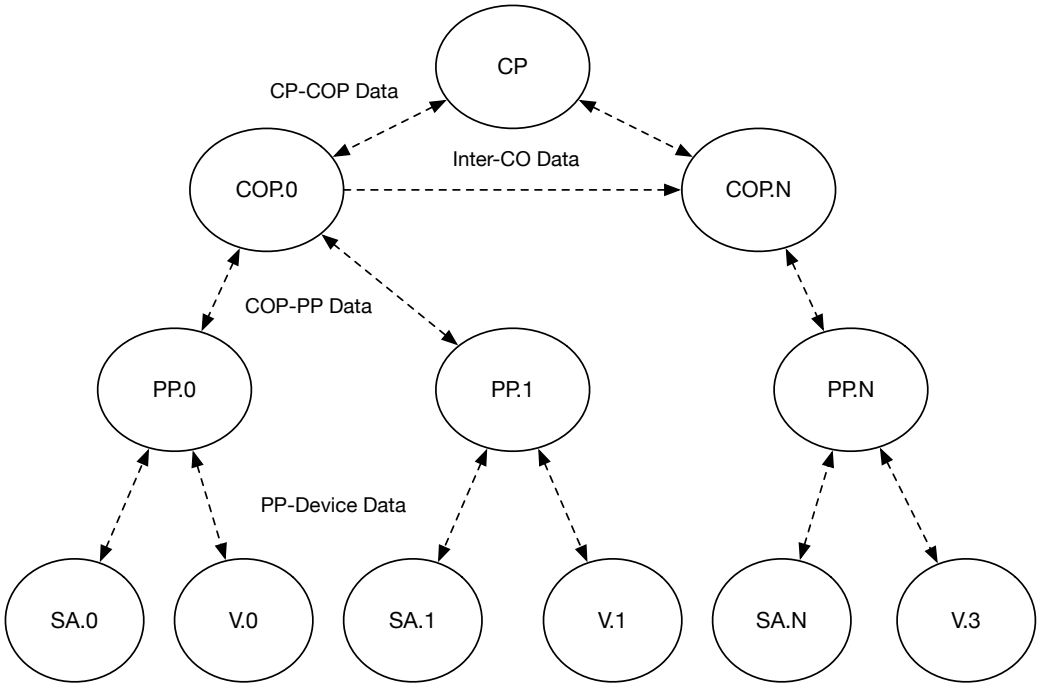

Figure 6.1: City-Wide Data Topology

6.2 Application Design

In this section, we describe the design aspects of a Cresco application used in the management of Smart Cities. As previously discussed, the Cresco framework operates in a hierarchy, which is configured based on application requirements and avalable resources. Figure 6.2, shows the hierarchical mapping of Cresco components to Smart City workload, resources, and designated data processing locations (PP, COP, CP). In the figure the data designated by dashed lines represents Cresco Plugin-to-Plugin workload communications, which is commonly referred to as the *Data Plane*. Solid lines represent data exchanged between Cresco components in the operation of the framework, which is commonly referred to as the *Control Plane*. Data and control planes can both operate on the same physical and logical networks. However, when planes are combined on the same network, priority is typically given to the control plane allowing control instructions to be issued even if the data plane is saturating the network.

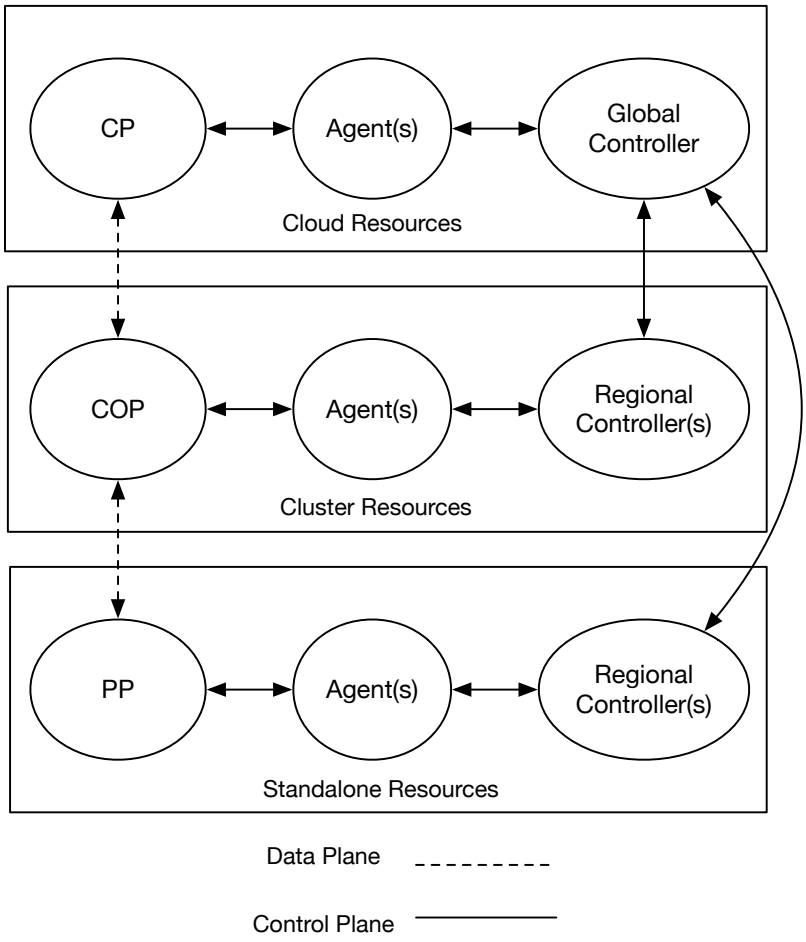

Figure 6.2: City Data and Cresco Topology

PPs are deployed on standalone devices with little or no general computing capacity. COPs are deployed on cluster resources with the ability to increase and decrease capacity as needed. CPs are deployed on large computational clouds, which could be provided by the city or a public cloud provider. If a city event is taking place in a specific neighborhood the processing capacity of the assigned COP should be increased as needed. Likewise, during events impacting the entire city (for instance rush hour), COP and CP capacities should be increased. The Cresco framework will be used to increase and decrease process capacities as needed.

PP: PPs serve as the interface between end-devices and the application. PPs will function as gateways for sensors and transient devices such as vehicles and personal

mobile devices. Sensor arrays maintained within the same proximity as a PP will be directly wired to and potentially powered by the PP gateway. Transient devices will communicate with PPs using wireless technologies. Since PPs function as wireless radio transmitters, the coordination of wireless transmit power will be coordinated through wireless management services provided by COPs. Data obtained by PP gateways will be transmitted to or between end-devices and upstream COP services. For application communications, PPs will connect to a message queues provided by their upstream COPs. The Cresco components that make up PPs are shown in Figure 6.3.

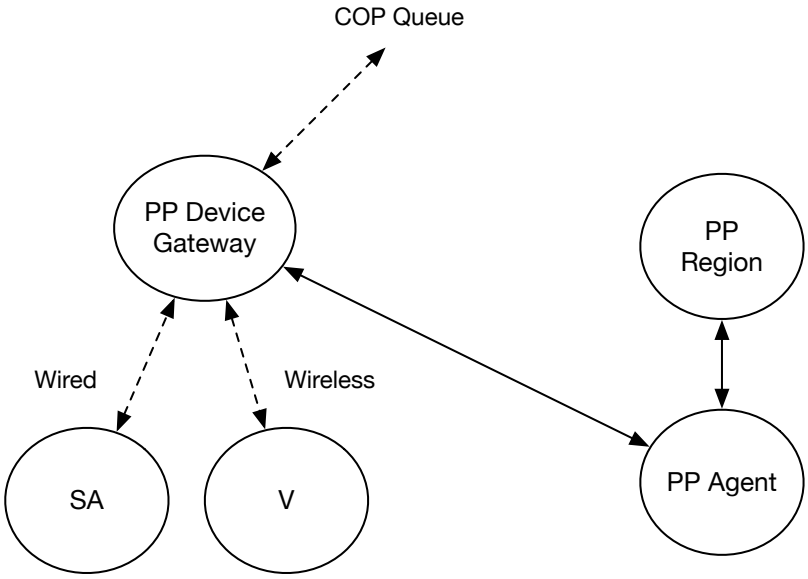

Figure 6.3: PP Data and Cresco Topology

When PP sensors readings exceed alarm thresholds, a message alert is propagated to upstream COP processors. Device data such as numbers of observed vehicles, speed, and traffic routing request that are propagated to COP processors. In addition, data reduction and privacy-preserving features are provided by data filtering and obfuscation measures applied to device data. Likewise, information pertaining to specific vehicles is assigned an internal unique identifier by the PP Plugin before data is propagated to COP processing, which obfuscates any possible identifiers provided by the remote device as part of the PP-device gateway message exchange. Data from

the PP Plugin is propagated to an assigned queue provided by a COP.

COP: Given COP resources should be able to expand and contract as needed the number of Cresco components provisioned for each COP will vary. At a minimum a COP will be composed of a queue and workload processor. The queue and workload processor are implemented as Cresco Plugins. The workload processor provide analysis services for connecting PP systems. Workload processors will be expanded as needed through the provisioning of additional Cresco Plugins. The queue will be used to exchange data plane messages between PP, COP, and CP systems. A single queuing plugin will be implemented per COP system. The Cresco components that make up COPs are shown in Figure 6.4.

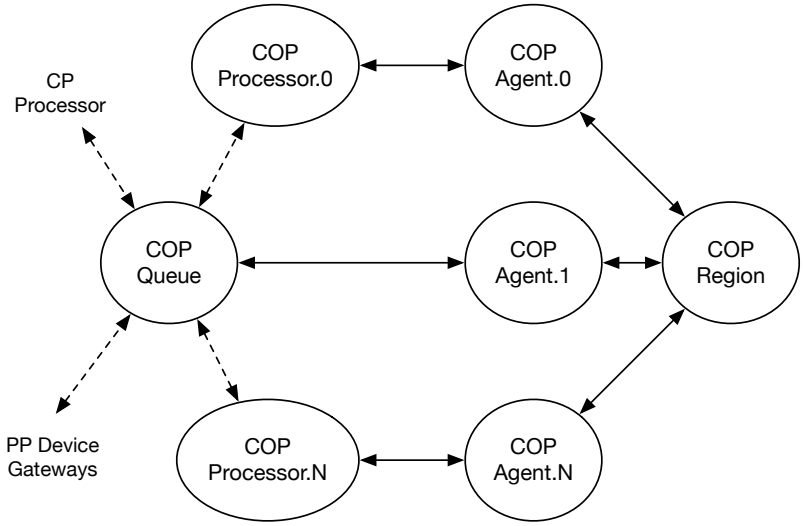

Figure 6.4: COP Data and Cresco Topology

COP Queue: The COP Queue is provided by a Cresco Plugin that provisions a single instance of the RabbitMQ queue management system for each COP location. Incoming and outgoing queues are provided for each PP location, COP Processor, and CP Processor. All data plane communication takes place over the COP Queue.

COP Processor: The COP Processor is provided by at least one Cresco Plugin for each COP location. Since there is a single COP Queue per location, there are

single sources of data for each incoming PP data stream. When multiple COP Processors exist, they retrieve their data from the same queue in a round-robin order, where each queue subscriber (COP Processor) receives an equal share of incoming data messages. Incoming PP messages are retrieved from queues, where exactly one message is delivered to a queue subscriber. In the described application, the primary purpose of the COP Processor is to maintain a summarized state of neighbor data from information obtained from PPs. In addition, COP Processors answer request from end-devices that are communicated through PPs based on data that is maintained and processed on the COP. Also, a COP Processor might provide a vehicle directions based on known city-wide traffic status. Finally, COP Processors communicate system-wide directives to PPs, which in turn communicate these directives to end-devices. For example, broadcast notifications pertaining to inclement weather city-wide, or directed notifications of public safety events (shooting, riot, gas leak, etc.) pertaining to a neighborhood location. COP Processors also relay coordinated data streams to CP Processors for further analysis.

CP: CP analysis could require significant computational, network, and storage resources, perhaps beyond what is available within existing infrastructures. As with COP, CP resources will vary based on load. However, unlike COP Cresco will provision additional infrastructure capacity as needed for CP operations. Infrastructure capacity will be increased through the provisioning of virtual machines running Cresco agents, which will in turn allow for the provisioning of additional CP workload processors. CP workload processors will subscribe to message queues provided by distributed COPs. The Cresco components that make up CPs are shown in Figure 6.5.

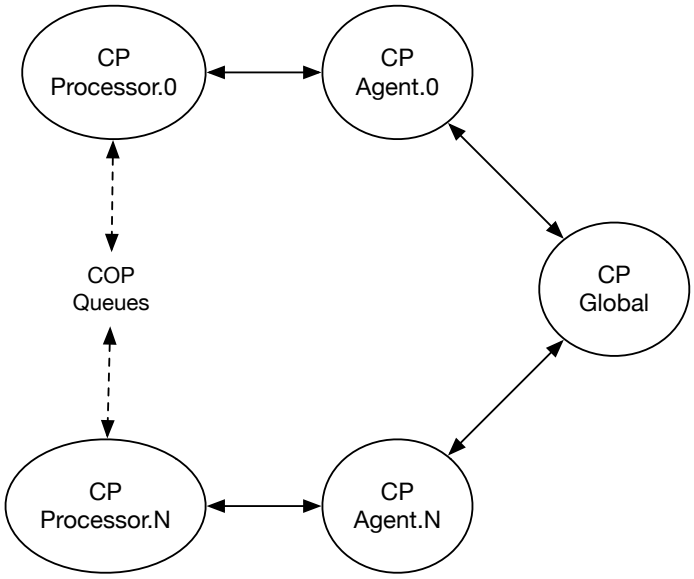

Figure 6.5: CP Data and Cresco Topology

CP Processor: The CP Processor is provided by at least one Cresco Plugin for each CP instance. Exactly one CP instance is responsible for recording time-series and event information in a central database. One or more CP instances use the central database to develop models that can be used by CP and COP Processors to make decisions. For example, traffic data propagated from PPs through COPs makes its way into the central database, which reflect the current status of city-wide traffic.

6.3 Application Implementation

In this section we describe the implementation details of components used to demonstrate the Cresco framework in the described application. Specifically, we will describe plugin implementations representing components described in the previous section, *Application Design*.

The Cresco Application Description Language (CADL) is used to describe the configuration and relationship between Cresco Plugins, plugin locations, and related plugin configurations used in the described application pipeline. At the end of this section we provide a CADL fragment used in initial application deployment.

6.3.1 Plugin Implementations

The following Cresco Plugins are used in the described application implementation.

PP Plugin: Since we don't have access to the physical devices that might be used in the PP environment we developed a Cresco Plugin to simulate the PP gateway environment, including data generation for sensor array and vehicle data. This plugin will connect to a message queue provided by the COP. From an application topology prospective the inclusion of data generation services within the representative gateway plugin constituters the only logically topology change between the proposed design and demonstrated implementation. The Cresco components that make up the PPs implementation is shown in Figure 6.6.

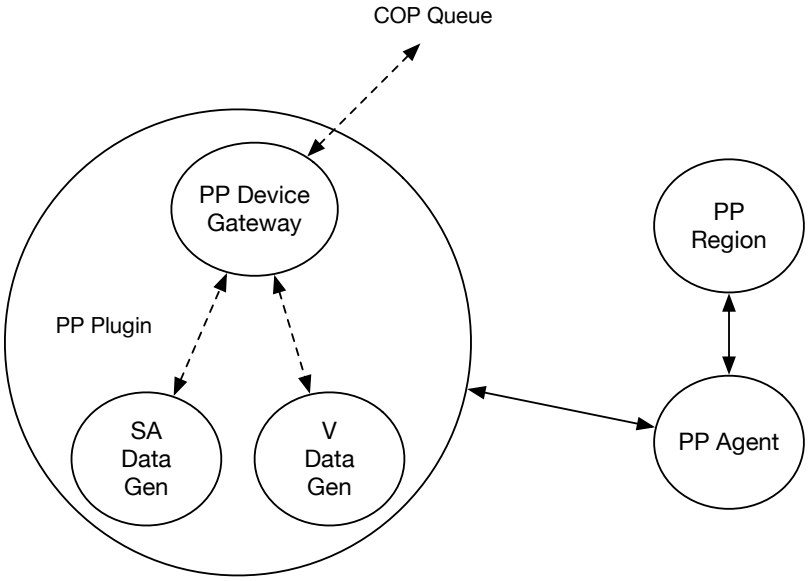

Figure 6.6: PP Data and Cresco Implementation

This plugin simulates the output of an array of sensors, which will be configured to pseudo-randomly generate readings that exceed alarm thresholds. Device data will be generated that simulates periods of high and low traffic, including numbers of observed vehicles and their related speed.

COP Queue Plugin: There will be no changes between design and implementation of COP Queue services.

COP Processor Plugin: The COP Processor as designed above is responsible for a number of complex tasks including coordination between PP and CP services. Demonstrating the use of Cresco in a COP implementation requires that we demonstrate the ability to scale, configure, provide communication, and maintain operation of COP Processes. The COP Processor Plugin implementation simulates analytic functions, communicates using a COP Queue Plugin, and based on self-reported load, request the addition or reduction of COP Processor Plugins within its COP location. The initially provisioned (COP Master) COP Plugin will be maintained for the duration of application operation. A COP Master communicates with its host Cresco Regional Controller to control COP Processor Plugin count. Through the use of the Cresco Optima project, described in Section 4.3.3, new COP Processor Plugins are provisioned on the least loaded COP instance.

CP Processor Plugin: As with COP Processors, the CP Processor is responsible for a large number of tasks. We will simulate the operation of CP Processors including downward propagation of city-wide data, end-device alerting, and on-demand expansion of CP resources. While COP Processors are expanded through the process of adding addition Cresco Plugins to existing infrastructure, CP resources are expanded through the addition of cloud-based infrastructure. The initially deployed (CP Master) CP Processor is responsible for maintaining a central database and for the expansion and contraction of infrastructure resources. Through the use of the Cresco Guilder project, described in Section 4.3.1, additional CP Processor Plugin requests issued to the Cresco Global Controller will result in the provisioning of additional virtual machines to host CP Instances. Through the use of the Cresco Optima project new CP Processor Plugins are provisioned on the least loaded CP instances.

6.3.2 Implementation CADL

The CADL for the application implementation describes the initial state of the application including configurations, relationships, and plugin location specifications. Example CADL node descriptions for CP Processor Plugins, COP Queue Plugins, COP Processor Plugins, and PP Plugins are shown below in Listings 6.1, 6.2, 6.3, and 6.4 respectively.

Listing 6.1: CADL CP Processor Node

```
"node_id":"0"
"node_name":"CP_PROCESSOR.0"
"type":"cp-processor-plugin"
"params":
 "pluginname":"cp-processor-plugin
 "jarfile":"cp-processor-plugin-0.1.0.jar"
 "location":"CP.0"
"isStateless":false
"isSource":false
```

Listing 6.2: CADL COP Queue Node

```
"node_id":"1"
"node_name":"COP_QUEUE.0"
"type":"cresco-container-plugin"
"params":
 "pluginname":"cresco-container-plugin
 "jarfile":"cresco-container-plugin-0.1.0.jar"
 "container_image":"rabbitmq:3-management"
 "e_params":"CRESCO_LOCATION=COP_QUEUE.0,RABBITMQ_USER=
     crescorquser,RABBITMQ_PASS=rqpassword"
 "p_parms":"5672,15672"
 "location":"COP_QUEUE.0"
"isStateless":false
"isSource":true
```

Listing 6.3: CADL COP Processor Node

```
"node_id":"2"
"node_name":"COP_PROCESSOR.0"
"type":"cop-processor-plugin"
"params":
 "pluginname":"cop-processor-plugin
 "jarfile":"cop-processor-plugin-0.1.0.jar"
 "location":"COP.0"
"isStateless":false
"isSource":false
```

Listing 6.4: CADL PP Node

```
1  "node_id":"3"
2  "node_name":"PP.0"
3  "type":"cresco-container-plugin"
4  "params":
5   "pluginname":"cresco-container-plugin
6   "jarfile":"cresco-container-plugin-0.1.0.jar"
7   "container_image":"gitlab.rc.uky.edu:4567/cresco/cresco-pp-
        container"
8   "e_params":"CRESCO_LOCATION=PP.0"
9   "location":"PP.0"
10 "isStateless":false
11 "isSource":true
```

The most basic CADL for the described implementation contains: a single CP, *CP.0* with CP Processor *CP.0 Processor.0 Plugin*, a single COP, *COP.0*, with COP Queue Plugin *COP.0 Queue Plugin* and COP Processor Plugin *COP.0 Processor.0 Plugin*, and a single PP *PP.0*, with PP Plugin *PP.0*. This minimum design results in the provisioning of at least four agents and seven plugins, of which three are controllers. The minimum CADL Cresco topology is shown in Figure 6.7.

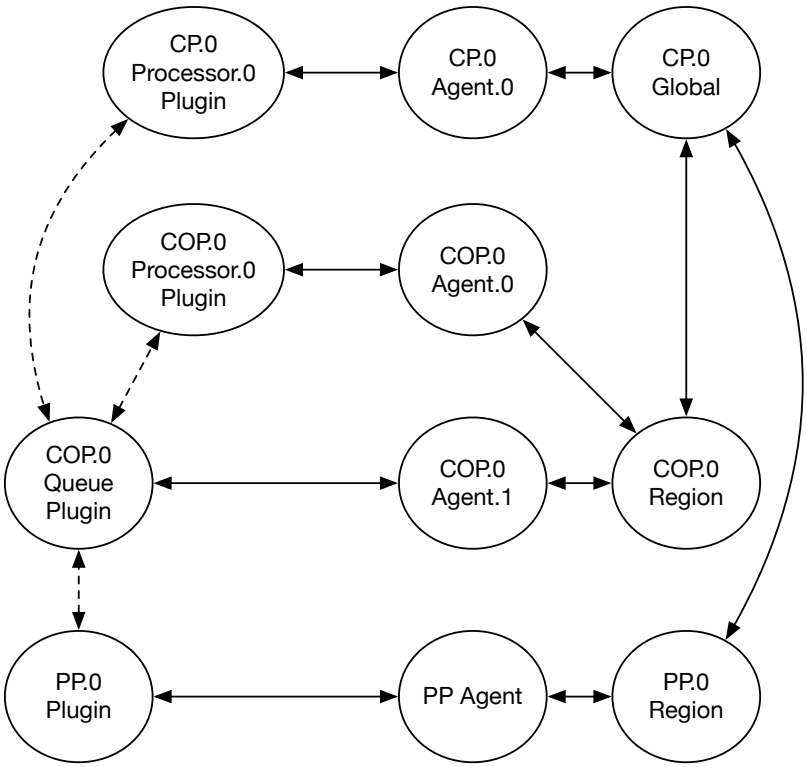

Figure 6.7: Minimum CADL Implementation Topology

The CADL implementation shown in the previous figure describes a single PP node. However, this single COP resource might be capable of supporting many PPs, with Cresco allocating resources as needed. Likewise, the single CP might be capable of supporting a number of COPs, and it to is dynamically scalable through the Cresco framework. The assignment of PPs to COPs in a real world deployment would depend on the geographic service of the COP itself. For example, PPs might be physical connected to specific COPs based on the distribution of existing fiber optics in a neighborhood.

As the size and complexity of applications grow, so does the CADL. For example, suppose we initially assign 100 PPs to every COP, and assign 100 COPs to a single initial CP. The initial CADL pipeline will contain a graph with 30,000 PP nodes and 40,000 edges, 500 COP nodes and 700 edges, and three CP nodes and two edges. The described graph results in a CADL size of 27.4 Megabytes. As previously mentioned, CADLs are highly compressible and the Global Controller supports the submission of Base64-encoded compressed CADL representations. In this case the compressed CADL is 1.3 Megabytes, which represents a 21:1 compression ratio. One method of reducing CADL pipeline size and complexity is to use application containers. The Cresco framework allows us to place agents and plugins within containers, while still maintaining agent, regional, and global communications. From a Cresco topology prospective a agent running within a container, on a VM, or on physical hardware is treated the same. For this implementation we combined a single agent and ten PP plugins within a container *cPP*. The cPP container is published to a central registry location: *gitlab.rc.uky.edu:4567/cresco/pp*. Each time the cPP container is executed the container is updated, if needed, from the central registry. Given Cresco is capable of running containers, even those with other Cresco agents, we can treat the cPP container as a single plugin, thus reducing the CADL graph.

Perhaps more important than CADL size is the ability to manipulate functional layers of applications independent of a single CADL pipeline representation. For

example, we might want to update the cPP container image without impacting COP or CP operations. In this implementation we will create three independent CADLs as shown below:

1. *queuePipeline*: The queuePipeline provides queuing services for PP, COP, and CP data-plane communications.

2. *PPPipeline*: The PPPipeline generates simulated data (vehicle, sensor, etc.) and provides communication services between simulated devices and higher-level control systems.

3. *COPPipeline*: The COPPipeline provides COP (edge) analysis services, such as filtering, alerting, and complex event processing services. This pipeline consumes data from the PPPipeline and communicates analysis results to and from higher-level control systems.

While isolating application components into separate CADLs provides greater flexibility, we must now keep track of multiple pipelines, including any inter-pipeline dependencies. In the next sub-section we describe the Application Controller, which is used to manage the pipelines described in this section.

6.3.3 Application Controller

A natural place to implement pipeline operations function is within CP operations. On initialization, the CP plugin contacts the Cresco Global controller specified in its configuration and deploys each CADL pipeline sequentially. The Cresco Global controller reports the status (status_code) of the pipeline. Once pipeline operational status is verified (status_code=10), the CP starts its analytic services.

In the next section we describe the operation of the application implementation described in this section.

6.4 Application Operation

We will simulate a distributed Smart City environment using 20 servers, each with the following resource capacity:

- *Cores*: 8
- *RAM*: 8G
- *Disk*: 128G

The Application Controller is deployed on a single virtual machine. The queuePipeline, PPPipeline, and COPPipeline, are deployed on 19 physical nodes, representing areas of a city identified by location identifiers $0-18$. On initial provisioning each location contains one COP, 15 cPPs (150 PPs), and a single queue. The Cresco Application Scheduler determined that no more than 15 cPPs could be assigned location based on observed cPP performance and available capacity. In the described configuration we can simulate one city, as described in Sub-section 6.4.4, *City-Level Operations*, 19 neighborhoods, described in Sub-section 6.4.3, *Neighborhood-Level Operations*, and 2850 sensor arrays, described in Sub-section 6.4.2, *Street-Level Operations*.

In the next section we discuss how application pipelines are deployed.

6.4.1 Pipeline Deployment

The first step in Application Controller operation is the deployment of the queuePipelines, PPPipeline, and COPPipeline CADLs. Listing 6.5 shows output of an Application Controller during startup.

Listing 6.5: CADL Deployment : Application

```
1 --Deploying queuePipeline
2 Waiting on queuePipeline 91e58c7e status_code: 3
3 Waiting on queuePipeline 91e58c7e status_code: 4
4 ...
5 Waiting on queuePipeline 91e58c7e status_code: 10
6
```

```
--Deploying COPPipeline
Waiting COPPipeline 6d59cc33 status_code: 3
Waiting COPPipeline 6d59cc33 status_code: 4
...
Waiting COPPipeline 6d59cc33 status_code: 10
--Deploying PPPipeline
Waiting on PPPipeline 69f60227 status_code: 3
Waiting on PPPipeline 69f60227 status_code: 4
...
Waiting on PPPipeline 69f60227 status_code: 10

--Application Start
Application Controller State: 10
queuePipeline 91e58c7e status_code: 10
copPipeline 6d59cc33 status_code: 10
ppPipeline 69f60227 status_code: 10
```

On submission of CADLs to the Cresco Global Controller the AppScheduler service starts the high-level scheduling process, which includes resource assignment. Listing 6.6 shows the output of the Global Controller while scheduling a node representing a cPP container.

Listing 6.6: CADL Node Scheduling

```
[AppSchedulerEngine] Location: 28
[AppSchedulerEngine] Assigned Nodes : 1
[AppSchedulerEngine] Unassigned Nodes : 0
[AppSchedulerEngine] Noresource Nodes : 0
[AppSchedulerEngine] Error Nodes : 0
[FuturaEngine] ResourceMetric for Container:
          gitlab.rc.uky.edu:4567/cresco/pp
[OptimaEngine] totalresourceWorkloadUtil: 289.0
               totalresourceAvalable: 5994.88
[ProviderOptimization] Starting Solver.
[ProviderOptimization] i=0 K[i] = 30
[ProviderOptimization] Solution #1
[AppSchedulerEngine] Submitted to ResourceScheduler
```

Based on the location constraints and previously observed resource utilization a candidate agent is selected by the AppSchedulerEngine for assignment. Once a Cresco agent has been assigned by the AppSchedulerEngine the resource request is submitted to the ResourceSchedulerEngine. The ResourceSchedulerEngine translates the CADL node description into a plugin configuration and then submits the plugin

configuration to the desired agent for initialization. Listing 6.7 shows the output of the Global Controller while scheduling a plugin representing a cPP container.

Listing 6.7: CADL Node to Plugin Scheduling

```
1 [ResourceSchedulerEngine] Incoming resource
2 [ResourceSchedulerEngine] starting precheck...
3 [ResourceSchedulerEngine] verifyPlugin params = OK
4 [ResourceSchedulerEngine] plugin precheck = OK
5 [ResourceSchedulerEngine] Payload: {msg=add plugin, dst_region=
    region-a4e4cdcf-8575-4612-aa7e-b8d0d31d276d, dst_agent=agent-
    ddbc73e6-8f09-4b53-a9f9-bde29884c90a, configtype=pluginadd,
    src_agent=agent-3ddd420c-1807-40ef-9b1c-adc4b58c3da6,
    src_region=region-a74b4d66-6ede-4c70-a0b5-7794a6b99fb8,
    src_plugin=plugin/0, http_host=http://172.17.0.1:32000/PLUGINS
    /,http://10.33.18.2:32000/PLUGINS/, jarmd5=1
    d918b8a374a3b32d8bde212f3307f6c, configparams=e_params=
    CRESCO_path_stage:CRESCO_cop_id:CRESCO_pp_amqp_host:
    CRESCO_discovery_secret_agent:
    CRESCO_discovery_ipv4_agent_timeout,pluginname=cresco-
    container-plugin,jarfile=cresco-container plugin-0.1.0.jar,
    CRESCO_path_stage=1,CRESCO_cop_id=cop-28,
    CRESCO_discovery_secret_agent=cresco_discovery_secret32,
    location=32,CRESCO_discovery_ipv4_agent_timeout=20000,
    container_image=gitlab.rc.uky.edu:4567/cresco/pp,resource_id=
    c4836c2f-b46a-4c33-be90-1556bf0f64bd,inode_id=f30c56e4-ad33-44
    c8-a4ab-2d28c360e87f}
6 [ResourceSchedulerEngine] Scheduling plugin on
7   region=region-ecaca483-56d8-43c8-9584-75feedbcddaf
8   agent=agent-04710df2-360f-44f3-8ebc-c3bf480dace5
```

Once pipelines have been started, distributed operations begin as described in the following sub-sections.

6.4.2 Street-Level Operations

We assume street-Level operation takes place in a location with limited computational capacity, such as a intelligent street light. On a street-level we want to simulate the collection of data from sensors, which should be physically connected or in proximity to collection devices managed by PPs. This includes specific sensors and devices, such as vehicles. We generate 1000 sensors values for each PP gateway node per second. These sensor values include a fix sensor identifier for each PP gateway. Peak traffic

periods may vary from city to city, from region to region, and seasonally. We assume a higher volume of traffic between 6-10 am (06:00-10:00) and 4-8 pm (16:00-20:00). Based on time of day maintained by each PP gateway, we generate data representing between 10 and 70 vehicles per second. Vehicle data includes a unique identifier and vehicle speed. Based on the simulated time of the day each PP gateway is capable of generating between 1010 and 1070 data points per second.

Simulated data on the street-level (PP) is transmitted to the neighborhood-level (COP) for processing. Neighborhood-level processing is described in the next subsection.

6.4.3 Neighborhood-Level Operations

We assume neighborhood-level operations take place in small-to-medium sized telecommunication facilities distributed throughout a city, with enough computational capacity to provide analytic services (COP) for a network of associated PP gateways. In the described application there are 150 PP gateways assigned per COP, which results in the processing of between 151,500 and 160,500 data points per second. The following COP analytic services have been implemented for neighborhood-level processing:

- *sensor_alert*: Sensor data filtering services, such as those that detect and provide alert on individual sensor anomalies, have been developed.
- *sensor_data*: Sensor data aggregation services, such as those that report average sensor readings based on a number of sensor locations, have been developed.
- *car_speed*: Vehicle data services, such as those that report average vehicle speed on a street-level, have been developed.
- *car_count*: Vehicle data service, such as those used to determine the number of vehicles in an intersection have been developed.

Listing 6.8 shows the complex event queries relating to the previously described analytic services. Queries can be added and removed from COPs dynamically through Cresco control-channel operations that interact directly with plugins, even when deployed within a container.

Listing 6.8: COP CEP

```
1 addQuery("sensor_alert", "select ppId, sensorId, sensorValue from
     sensorMap where sensorValue = 1000");
2 addQuery("sensor_data", "select ppId, sensorId, avg(sensorValue,
    group_by:sensorId) as avgValue from sensorMap.win:time_batch
    (15 sec) group by ppId output snapshot every 1 seconds");
3 addQuery("car_speed", "select irstream distinct ppId, avg(
    carValue) as sps from carMap.win:time(15 sec) group by ppId
    output snapshot every 1 seconds");
4 addQuery("car_count", "select ppId, count(*) as avgValue from
    carMap.win:time_batch(15 sec) group by ppId output snapshot
    every 1 seconds");
```

As previously mentioned, each COP instance receives between 151,500 and 160,500 raw data points per second, which results in the processing of 2,878,500 and 3,049,500 data points across 19 locations. The four described analytic operations generate approximately 150 data points per second for each COP. Data generated by COPs on the neighborhood-level is communicated to the CP. If COP operations were not distributed to the neighborhood-level, millions of data points per second would have to be transmitted to a central location for processing.

City-level operations are described in the next section.

6.4.4 City-Level Operations

In a real-world deployment, the majority of data modeling and storage would take place centrally, where computational resources are more likely to be available compared to neighborhood and street-level operations. However, the purpose of this chapter is to describe the use of Cresco Applications, not the implementation of a Smart City framework. We limit the implementation of city-level services to those required for the Application Controller.

As previously described, the Application Controller is responsible for pipeline operations. Once the initial pipelines have been deployed, the Application controller maintains an operational status for each COP based on data provided by Cresco and load information provided by each COP instance. Using methods described in the previous sub-section, he Application Controller determines the number of reported vehicles per COP location. If the number of vehicles per COP location exceeds the rush-hour threshold of 30 vehicles per PP gateway, an additional COP instance is added to the location. The Application Controller maintains a list of COP-specific pipelines and if the per COP vehicle threshold drops below 10 vehicles per PP gateway COP instance pipelines are removed until there is a single COP instance remaining in a specific location. Listing 6.9 shows the addition of COP pipelines by the Application Controller.

Listing 6.9: Add COP Processor Alert

```
1 Add COP: ce95b4eb-5954-42e3-a943-31d5ac11535c high cop-33:381.6
2 ...
3 Add COP: 4413c788-5fb6-448e-8387-7cd65c795e7d high cop-28:380.7
```

In addition to the described pipeline management functions, methods have been implemented in the Application Controller (CP) allowing communication to specific COPs and PPs for the purposes of alert acknowledgement.

6.5 Application Conclusions

While the application described in this chapter is hypothetical in nature, it serves to demonstrate the types of applications that might benefit from edge computing and by association the Cresco framework. While the data described in this chapter was simulated, the pipeline components related to data generation, communication, and associated analytics are real implementations. We demonstrated the distributed processing of over over 6 million simulated sensor data points per second. The foundational capabilities provided by Cresco might provide benefit to entire class of applications,

where component abstraction and dynamic resource scheduling are beneficial.

7

Conclusions

There is no generally accepted theory for edge computing. In this dissertation the characteristics, challenges, and motivations for edge computing were presented. A number of real-world use cases were described to support claims that in some cases moving computation resource assignments to sources of data is more effective than moving data to computational resources. As the number of connected devices increase globally, so will the need for intelligent end-to-end management of computational resources, workloads, and data.

An edge computing framework named Cresco was developed. While a number of open source packages were used to develop the framework all core software development, with the exception of the Cresco libraries discussed in Sections 3.4, *Cresco Plugin Library* and 3.5, *Cresco Library*, were developed by the author. Cresco libraries were developed by Caylin Hickey using common code that was originally written by the author and repeated between components. A number of custom plugins that where not covered in this dissertation have been developed by the author and others.

In the next section we describe work related to the Cresco framework described in this dissertation.

7.1 Related Work

Early designs and implementations of the work presented in this dissertation predate the seminal work of Bonomi et al., 2012 [21] defining the characteristics of Edge Computing[1]. The use of agents in our framework was greatly influenced by the work of V.S. Subrahmanian, et .al. [75]. The idea proposed by Named Data Networking (NDN) [197], that networks should be data-centric not host-centric greatly influenced design aspects of message routing, agent identification, and node hierarchy. Actor-model concurrent programing, especially as implemented in ERLANG [71], influenced agent implementation. Long-distance live (uninterrupted) workload migration [198], through the management of application and infrastructure layers, influenced design aspects of continuous scheduling and optimization. Work with Apache Storm Topologies [156] influenced the use of graphs to model the relationships between workloads and data flow.

Making use of previous efforts and expanding on the work of Bonomi et al., 2014 [90], Lopez et al., 2015 [22], and Varghese et al., 2016 [23], we developed an architectural model for an edge-focused distributed resource and application framework, that we named Cresco. Over the course of six years a Cresco implementation was developed, which now consist of nearly forty-thousand lines of source code. Currently, there are a number of mature Cresco-based applications used in production, with others under development.

In the next section we describe the accomplishments that were realized during the development of this dissertation.

[1] Initially referred to as Fog Computing.

7.2 Achievements

We believe the work described in this dissertation has demonstrated a number of achievements, namely:

- An architectural model was developed to address aspects of edge computing, as described in Chapter 2, *The Architectural Model*.
- Cresco, an edge-focued distributed application and resource management framework was implemented, as described in Chapter 3, *Cresco Implementation*.
- Cresco provides data collection services used to take stock of a geographically distributed network of resources.
- The Cresco Application Description Language (CADL) provides a way to describe applications, resource needs, and workload placements, as described in Chapter 4, *Framework Technologies*.
- Cresco provides services to schedule, provision, and maintain CADL pipelines.
- Cresco provides the ability to configure workloads and resources on the edge of networks, as described in Chapter 5, *Case Studies of Edge Computing*
- Cresco provides services to acquire new resources as needed from public and private computational clouds, as demonstrated in Chapter 6, *Building a Smart City Application*.
- Cresco provides services to attempt workload scheduling and resource acquisition optimization.
- Cresco provides services to predict workload and pipeline resource needs, based on observed resource utilization.
- Cresco provides services to predict overall system resource needs, based on historical utilization.

- Cresco provides services to group or separate workloads based on observed resource alignment (two nodes communicate with each other) or competition.

In the next section we describe potentially future areas of development related to the work described in this dissertation.

7.3 Future Work

There are a number of aspects of the work presented in this dissertation that can benefit from additional work, including but not limited to:

- Currently, the Cresco is implemented using the cross-platform language Java. While the computational requirements for basic agent services are low, Java interpreters are not supported on all devices. A simple communication library developed in C/C++ would allow Cresco to interact with more devices.

- The Futura project tracks a number of resource utilization metrics for workloads. However, for resource utilization assessment the current implementation is limited to CPU metrics. In the future, all tracked resource metrics should be included in workload utilization, clustering, and profile functions.

- The Optima project tracks a number of resource utilization metrics for resource providers. However, for resource provider capacity assessment the current implementation is limited to CPU metrics. In the future, all tracked metrics should be included in resource optimization scheduling functions.

- Guilder is capable of acquiring resources from a number of locations, including so-called "spot" instances, which are priced based on a point-in-time cost with time-limit restrictions. Futura is capable of predicting resource utilization, but functions have not been developed to predict workload duration. In the future, Futura should be expanded to allow for the acquisition of spot instances by Guilder.

- As with Futura and Guilder, Optima constrain programming models are based on CPU metrics only. In the future, Optima models should include all resource utilization aspects maintained by Futura.

The Cresco framework will continue to evolve as new applications are developed, challenges are realized, and the Cresco community expands.

Bibliography

[1] S. Chen, H. Xu, D. Liu, B. Hu, and H. Wang. A Vision of IoT: Applications, Challenges, and Opportunities With China Perspective. *IEEE Internet of Things Journal*, 1(4):349–359, Aug 2014.

[2] LM Ericsson. More than 50 billion connected devices. `http://www.akos-rs.si/files/Telekomunikacije/Digitalna_agenda/Internetni_protokol_Ipv6/More-than-50-billion-connected-devices.pdf`, 2011.

[3] Dave Evans. The Internet of Things: How the Next Evolution of the Internet Is Changing Everything. `http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf`, 2011.

[4] IBM. The Four V's of Big Data. `http://www.ibmbigdatahub.com/infographic/four-vs-big-data`, 2016. [Online; accessed 14-October-2016].

[5] Mark Weiser. The Computer for the 21st Century. *Scientific american*, 265(3):94–104, 1991.

[6] Radhakisan Baheti and Helen Gill. Cyber-physical systems. *The Impact of Control Technology*, 12:161–166, 2011.

[7] David Boswarthick, Omar Elloumi, and Olivier Hersent. *M2M Communications: A Systems Approach*. John Wiley & Sons, 2012.

[8] Peter C Evans and Marco Annunziata. Industrial internet: Pushing the boundaries of minds and machines. *General Electric. November*, 26, 2012.

[9] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of Things for Smart Cities. *Internet of Things Journal, IEEE*, 1(1):22–32, 2014.

[10] Inc. Amazon Web Services. AWS IoT. `https://aws.amazon.com/iot/`, 2016. [Online; accessed 27-May-2016].

[11] PTC. ThingWorx IoT. `http://www.thingworx.com/`, 2016. [Online; accessed 27-May-2016].

[12] LogMeIn. Xively. `https://www.xively.com/`, 2016. [Online; accessed 27-May-2016].

[13] Bosch IoT Suite. Bosch IoT Suite. https://www.bosch-si.com/products/bosch-iot-suite/platform-as-service/paas.html, 2016. [Online; accessed 27-May-2016].

[14] SensorCloud. SensorCloud. http://sensorcloud.com, 2016. [Online; accessed 27-May-2016].

[15] IoTivity. IoTivity. https://www.iotivity.org/, 2016. [Online; accessed 27-May-2016].

[16] Amazon Web Services Inc. Amazon elastic compute cloud (Amazon EC2). https://aws.amazon.com/ec2/.

[17] Microsoft Inc. Microsoft Azure. http://www.azure.com, 2014. [Online; accessed 8-December-2014].

[18] Cyber Physical Systems Public Working Group et al. Framework for Cyber-Physical Systems. https://pages.nist.gov/cpspwg/, 2016.

[19] Kenneth L Calvert, Matthew B Doar, and Ellen W Zegura. Modeling Internet Topology. *IEEE Communications magazine*, 35(6):160–163, 1997.

[20] K Raza and M Turner. *Cisco Network Topology and Design*. Cisco Press, 2002.

[21] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog Computing and its role in the Internet of Things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[22] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric Computing: Vision and Challenges. *SIGCOMM Comput. Commun. Rev.*, 45(5):37–42, September 2015.

[23] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S Nikolopoulos. Challenges and Opportunities in Edge Computing. *arXiv preprint arXiv:1609.01967*, 2016.

[24] Stuart A Boyer. *SCADA: Supervisory Control and Data Acquisition*. International Society of Automation, 2009.

[25] ORBCOMM. ORBCOMM. http://www.orbcomm.com/en/industries/oil-and-gas-utilities/pipeline-monitoring, 2016. [Online; accessed 11-September-2016].

[26] PennWell Corporation. The Role of satellites in oil and gas pipeline monitoring for leak and theft detection. http://www.pennenergy.com/articles/pennenergy/2014/05/the-role-of-satellites-in-oil-and-gas-pipeline-monitoring-for-leak-theft-detection.html, 2014. [Online; accessed 11-September-2016].

[27] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for VM-based Cloudlets in Mobile Computing. *IEEE pervasive Computing*, 8(4):14–23, 2009.

[28] Stephen R Ellis, Katerina Mania, Bernard D Adelstein, and Michael I Hill. Generalizeability of latency detection in a variety of virtual environments. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 48, pages 2632–2636. SAGE Publications, 2004.

[29] Rob Kitchin. The real-time city? Big data and smart urbanism. *GeoJournal*, 79(1):1–14, 2014.

[30] Md Rokebul Islam, Nafis Ibn Shahid, Dewan Tanzim ul Karim, Abdullah Al Mamun, and Md Khalilur Rhaman. An efficient algorithm for detecting traffic congestion and a framework for smart traffic control system. In *2016 18th International Conference on Advanced Communication Technology (ICACT)*, pages 802–807. IEEE, 2016.

[31] M Mitchell Waldrop. No Drivers Required. *Nature*, 518(7537):20, 2015.

[32] Eun-Kyu Lee, Mario Gerla, Giovanni Pau, Uichin Lee, and Jae-Han Lim. Internet of Vehicles: From intelligent grid to autonomous cars and vehicular fogs. *International Journal of Distributed Sensor Networks*, 12(9):1550147716665500, 2016.

[33] Sebastian Feese, Michael Joseph Burscher, Klaus Jonas, and Gerhard Tröster. Sensing spatial and temporal coordination in teams using the smartphone. *Human-centric Computing and Information Sciences*, 4(1):1, 2014.

[34] Debanjan Das Deb, Sagar Bose, and Somprakash Bandyopadhyay. Coordinating disaster relief operations using smart phone/pda based peer-to-peer communication. *International Journal of Wireless & Mobile Networks*, 4(6):27, 2012.

[35] Poonam Sinai Kenkre, Anusha Pai, and Louella Colaco. Real time intrusion detection and prevention system. In *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*, pages 405–411. Springer, 2015.

[36] Larry Peterson, Ali Al-Shabibi, Tom Anshutz, Scott Baker, Andy Bavier, Saurav Das, Jonathan Hart, Guru Palukar, and William Snow. Central office re-architected as a data center. *IEEE Communications Magazine*, 54(10):96–101, 2016.

[37] Gang Peng. CDN: Content distribution network. *arXiv preprint cs/0411069*, 2004.

[38] Hazim Almuhimedi, Florian Schaub, Norman Sadeh, Idris Adjerid, Alessandro Acquisti, Joshua Gluck, Lorrie Faith Cranor, and Yuvraj Agarwal. Your location has been shared 5,398 times!: A field study on mobile app privacy nudging.

In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 787–796. ACM, 2015.

[39] Serge Egelman, Adrienne Porter Felt, and David Wagner. Choice architecture and smartphone privacy: There's a price for that. In *The economics of information security and privacy*, pages 211–236. Springer, 2013.

[40] Adrienne Porter Felt, Serge Egelman, and David Wagner. I've got 99 problems, but vibration ain't one: a survey of smartphone users' concerns. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 33–44. ACM, 2012.

[41] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, page 3. ACM, 2012.

[42] Jialiu Lin, Shahriyar Amini, Jason I Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pages 501–510. ACM, 2012.

[43] Irina Shklovski, Scott D Mainwaring, Halla Hrund Skúladóttir, and Höskuldur Borgthorsson. Leakiness and creepiness in app space: Perceptions of privacy and mobile app use. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 2347–2356. ACM, 2014.

[44] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.

[45] Hassan Farhangi. The path of the Smart Grid. *IEEE power and energy magazine*, 8(1):18–28, 2010.

[46] Zhi Yang, Ben Y Zhao, Yuanjian Xing, Song Ding, Feng Xiao, and Yafei Dai. AmazingStore: Available, Low-cost Online Storage Service Using Cloudlets. In *IPTPS*, volume 10, pages 2–2, 2010.

[47] Yang Zhang, Dusit Niyato, Ping Wang, and Chen-Khong Tham. Dynamic offloading algorithm in intermittently connected mobile cloudlet systems. In *2014 IEEE international conference on communications (ICC)*, pages 4190–4195. IEEE, 2014.

[48] D Shires, B Henz, S Park, and J Clarke. Cloudlet seeding: Spatial deployment for high performance tactical clouds. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012.

[49] Keke Gai, Meikang Qiu, Hui Zhao, Lixin Tao, and Ziliang Zong. Dynamic energy-aware cloudlet-based mobile cloud computing model for green computing. *Journal of Network and Computer Applications*, 59:46–54, 2016.

[50] Amazon Web Services Inc. Netflix on AWS. `https://aws.amazon.com/solutions/case-studies/netflix/`, 2016. [Online; accessed 15-November-2016].

[51] Netflix. Netflix Company Profile. `https://ir.netflix.com/`, 2016. [Online; accessed 15-November-2016].

[52] Turnkey Linux. AWS Data Centers. `http://turnkeylinux.github.io/aws-datacenters/`, 2016. [Online; accessed 15-November-2016].

[53] Nick McKeown. Software-efined networking. *INFOCOM keynote talk*, 2009.

[54] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[55] Geoff Mulligan. The 6lowpan architecture. In *Proceedings of the 4th workshop on Embedded networked sensors*, pages 78–82. ACM, 2007.

[56] ZigBee Alliance. Zigbee 2007 specification. *Online: http://www. zigbee. org/Specifications/ZigBee/Overview. aspx*, 45:120, 2007.

[57] Tech Crunch. Autonomous cars generate 4tb of data daily. `https://techcrunch.com/2016/11/15/intel-announces-250-million-for-autonomous-driving-tech/?utm_content=buffer93975&utm_medium=social&utm_source=twitter.com&utm_campaign=buffer`, 2016. [Online; accessed 19-November-2016].

[58] Research and Innovative Technology Administration. Bureau of Transportation Statistics. `http://www.rita.dot.gov/bts/sites/rita.dot.gov.bts/files/publications/national_transportation_statistics/html/table_01_11.html`, 2015.

[59] Miles Bryan. Chicago puts sensors to work taking its vitals. `https://cdn.ampproject.org/c/www.marketplace.org/amp/2016/11/16/world/chicago-puts-sensors-work-creating-new-network`, 2016. [Online; accessed 19-November-2016].

[60] The OpenStack project. OpenStack. `http://www.openstack.org`, 2013. [Online; accessed 13-November-2013].

[61] V.K. Bumgardner. *Openstack in Action*. Manning Publications Company, 2015.

[62] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[63] Junaid Shuja, Abdullah Gani, Kashif Bilal, Atta Ur Rehman Khan, Sajjad A Madani, Samee U Khan, and Albert Y Zomaya. A survey of mobile device virtualization: Taxonomy and state of the art. *ACM Computing Surveys (CSUR)*, 49(1):1, 2016.

[64] ISI USC. Internet Protocol. RFC 791, RFC Editor, September 1981.

[65] Yakov Rekhter, D Karrenberg, G de Groot, and B Moskowitz. Address allocation for private internets. *IETF*, 1994.

[66] William Stallings. *Handbook of computer-communications standards; Vol. 1: the open systems interconnection (OSI) model and OSI-related standards*. Macmillan Publishing Co., Inc., 1987.

[67] Yvan Royon, Stéphane Frénot, and Frédéric Le Mouël. Virtualization of service gateways in multi-provider environments. In *Component-Based Software Engineering*, pages 385–392. Springer, 2006.

[68] George Tsirtsis. Network address translation-protocol translation (NAT-PT) RFC 2766. Technical Report 2766, The Internet Society, 2000.

[69] The Cresco Project. Cresco. `https://github.com/ResearchWorx/Cresco/wiki`, 2014. [Online; accessed 11-September-2016].

[70] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[71] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in ERLANG*. Citeseer, 1993.

[72] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The Scala language specification, 2004.

[73] Lightbend Inc. Akka framework. `http://akka.io/`, [Online; accessed 18-January-2016].

[74] Scott De Marchi and Scott E Page. Agent-based models. *Annual Review of Political Science*, 17:1–20, 2014.

[75] Ventatramanan S Subrahmanian. *Heterogeneous Agent Systems*. MIT press, 2000.

[76] Leigh Tesfatsion. Agent-based computational economics: Growing economies from the bottom up. *Artificial life*, 8(1):55–82, 2002.

[77] Peter Albin and Duncan K Foley. Decentralized, dispersed exchange without an auctioneer: A simulation study. *Journal of Economic Behavior & Organization*, 18(1):27–51, 1992.

[78] Wikipedia. Comparison of agent-based modeling software, 2016. [Online; accessed 26-November-2016].

[79] Kalliopi Kravari and Nick Bassiliades. A survey of agent platforms. *Journal of Artificial Societies and Social Simulation*, 18(1):11, 2015.

[80] Gurpreet Singh Bhamra, AK Verma, and RB Patel. Intelligent software agent technology: an overview. *International Journal of Computer Applications*, 89(2), 2014.

[81] Jacques Ferber. *Multi-agent systems: An introduction to distributed artificial intelligence*, volume 1. Addison-Wesley Reading, 1999.

[82] Douglas N Walton. *Dialog theory for critical argumentation*, volume 5. John Benjamins Publishing, 2007.

[83] Francois Yergeau. UTF-8, a transformation format of ISO 10646. Technical report, RFC Editor, 2003.

[84] Stefan Poslad. Specifying protocols for multi-agent systems interaction. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2(4):15, 2007.

[85] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. KQML as an agent communication language. In *Proceedings of the third international conference on Information and knowledge management*, pages 456–463. ACM, 1994.

[86] John R Searle. *Speech acts: An essay in the philosophy of language*, volume 626. Cambridge university press, 1969.

[87] Carlos Varela and Gul Agha. A hierarchical model for coordination of concurrent activities. In *International Conference on Coordination Languages and Models*, pages 166–182. Springer, 1999.

[88] Shang-Wen Luan, Jen-Hao Teng, Shun-Yu Chan, and Lain-Chyr Hwang. Development of a smart power meter for AMI based on ZigBee communication. In *2009 International Conference on Power Electronics and Drive Systems (PEDS)*, pages 661–665. IEEE, 2009.

[89] Dimitar Valtchev and Ivailo Frankov. Service gateway architecture for a smart home. *IEEE Communications Magazine*, 40(4):126–132, 2002.

[90] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog Computing: A platform for Internet of Things and analytics. In *Big Data and Internet of Things: A Roadmap for Smart Environments*, pages 169–186. Springer, 2014.

[91] The Apache Software Foundation. Apache ActiveMQ - MQTT. `http://activemq.apache.org/mqtt.html`, 2016. [Online; accessed 31-May-2016].

[92] The Apache Software Foundation. Apache license, version 2.0. `http://www.apache.org/licenses/LICENSE-2.0.html`, 2014. [Online; accessed 8-November-2014].

[93] FIPA Inform Communicative Act Specification. Foundation for intelligent physical agents, 2000, 2004.

[94] François Yergeau, Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0. *W3C Recommendation*, 4, 2004.

[95] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, RFC Editor, July 2006.

[96] John Ferguson Smart et al. An introduction to Maven 2. `http://www.javaworld.com/javaworld/jw-12-2005/jw-1205-maven.html`, 2005.

[97] Research Worx. Cresco Plugin Library Maven. `https://mvnrepository.com/artifact/com.researchworx.cresco/cresco-plugin-library`, 2016. [Online; accessed 4-December-2016].

[98] Research Worx. Cresco Plugin Library. `https://github.com/ResearchWorx/Cresco-Plugin-Library`, 2016. [Online; accessed 4-December-2016].

[99] Research Worx. Cresco Library Maven. `https://mvnrepository.com/artifact/com.researchworx.cresco/cresco-plugin-library`, 2016. [Online; accessed 4-December-2016].

[100] Research Worx. Cresco library. `https://github.com/ResearchWorx/Cresco-Library`, 2016. [Online; accessed 4-December-2016].

[101] OrientDB LTD. OrientDB. `http://orientdb.com/`, 2016. [Online; accessed 27-May-2016].

[102] ISI USC. Broadcasting Internet Datagrams. RFC 919, Network Working Group, October 1984.

[103] S. Derring R. Hinden. Broadcasting Internet Datagrams. RFC 2375, Network Working Group, July 1998.

[104] Frederic P Miller, Agnes F Vandome, and John McBrewster. *Advanced Encryption Standard*. Alpha Press, 2009.

[105] The Apache Software Foundation. Apache ActiveMQ. http://activemq.apache.org, 2013. [Online; accessed 13-November-2013].

[106] The Apache Software Foundation. Apache ActiveMQ - AUTO. http://activemq.apache.org/auto.html, 2016. [Online; accessed 31-May-2016].

[107] The Apache Software Foundation. Apache ActiveMQ - AUTO. http://activemq.apache.org/amqp.html, 2016. [Online; accessed 31-May-2016].

[108] The Apache Software Foundation. Apache ActiveMQ - OpenWire. http://activemq.apache.org/openwire.html, 2016. [Online; accessed 31-May-2016].

[109] The Apache Software Foundation. Apache ActiveMQ - REST. http://activemq.apache.org/rest.html, 2016. [Online; accessed 31-May-2016].

[110] The Apache Software Foundation. Apache ActiveMQ - RSS and Atom. http://activemq.apache.org/rss-and-atom.html, 2016. [Online; accessed 31-May-2016].

[111] The Apache Software Foundation. Apache ActiveMQ - STOMP. http://activemq.apache.org/stomp.html, 2016. [Online; accessed 31-May-2016].

[112] The Apache Software Foundation. Apache ActiveMQ - WSIF. http://activemq.apache.org/wsif.html, 2016. [Online; accessed 31-May-2016].

[113] The Apache Software Foundation. Apache ActiveMQ - WebSocket. http://activemq.apache.org/websockets.html, 2016. [Online; accessed 31-May-2016].

[114] The Apache Software Foundation. Apache ActiveMQ - XMPP. http://activemq.apache.org/xmpp.html, 2016. [Online; accessed 31-May-2016].

[115] Oracle Corporation. Glashfish. https://glassfish.java.net, 2014. [Online; accessed 7-November-2014].

[116] The Apache Software Foundation. Apache jClouds. http://jclouds.apache.org/, 2017. [Online; accessed 16-January-2017].

[117] OSHI. Operating System and Hardware Information. http://oshi.github.io/oshi/, 2017. [Online; accessed 16-January-2017].

[118] Tim O'Reilly Jerry Peek, Shelley Powers and Mike Loukides. *Unix Power Tools*. O'Reilly, 2007.

[119] Docker Inc. Docker. https://www.docker.com/, 2013. [Online; accessed 31-December-2016].

[120] Organization for the Advancement of Structured Information Standards. OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0. *OASIS Standard*, 2012.

[121] RabbitMQ. RabbitMQ. `http://www.rabbitmq.com`, 9 2013.

[122] Intel Inc. Intel® 64 and IA-32 Architectures Software Developer's Manual. *Intel Inc.*, 2010.

[123] Annie Foong and Frank Hady. Storage as fast as rest of the system. In *Memory Workshop (IMW), 2016 IEEE 8th International*, pages 1–4. IEEE, 2016.

[124] Stefan M Larson, Christopher D Snow, Michael Shirts, and Vijay S Pande. Folding@ home and genome@ home: Using distributed computing to tackle previously intractable problems in computational biology. *arXiv preprint arXiv:0901.0866*, 2009.

[125] Claus Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, pages 24–31, 2015.

[126] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM Computer Communication Review*, volume 19, pages 1–12. ACM, 1989.

[127] Chandandeep Singh Pabla. Completely fair scheduler. *Linux J.*, 2009(184), August 2009.

[128] Stephen A Rago. *UNIX System V Network Programming*. Addison-Wesley Professional, 1993.

[129] Rami Rosen. Resource management: Linux kernel namespaces and cgroups. *Haifux, May*, 186, 2013.

[130] David Chisnall. *The definitive guide to the Xen hypervisor*. Pearson Education, 2008.

[131] Miguel G Xavier, Marcelo Veiga Neves, Fabio D Rossi, Tiago C Ferreto, Tobias Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240. IEEE, 2013.

[132] Alphabet Inc. Google cloud computing. `https://cloud.google.com`, 2016. [Online; accessed 31-December-2016].

[133] Kimberly Keeton. The Machine: An Architecture for Memory-centric Computing. In *Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2015.

[134] Dirk Schmidl, Christian Terboven, Andreas Wolf, Dieter an Mey, and Christian Bischof. How to scale nested OpenMP applications on the ScaleMP vSMP architecture. In *2010 IEEE International Conference on Cluster Computing*, pages 29–37. IEEE, 2010.

[135] John Robert Wernsing and Greg Stitt. Elastic Computing: A framework for transparent, portable, and adaptive multi-core heterogeneous computing. In *ACM SIGPLAN Notices*, volume 45, pages 115–124. ACM, 2010.

[136] HPC Wire. CERN Details OpenStack Journey. `http://www.hpcwire.com/2014/11/04/cern-details-openstack-journey/`, 2014. [Online; accessed 8-November-2014].

[137] Roldan Pozo and Bruce Miller. Scimark 2.0. *URL: http://math. nist. gov/scimark2*, 2000.

[138] Amazon Web Services Inc. AWS EC2 Instance Types. `https://aws.amazon.com/ec2/instance-types/`, 2017. [Online; accessed 25-January-2017].

[139] Amazon Web Services Inc. AWS EC2 Price API. `http://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/price-changes.html`, 2017. [Online; accessed 25-January-2017].

[140] The Apache Software Foundation. Apache Commons Math. `http://commons.apache.org/proper/commons-math/`, 2017. [Online; accessed 27-January-2017].

[141] John A Hartigan and Manchek A Wong. A k-means clustering algorithm. *Applied statistics*, pages 100–108, 1979.

[142] William J Baumol and Philip Wolfe. A warehouse-location problem. *Operations Research*, 6(2):252–263, 1958.

[143] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016.

[144] VK Cody Bumgardner. Public IaaS Economics. Internal Report presented to University of Kentucky IT Executive Team, Lexington, Ky, 2011.

[145] Mark S Johnstone and Paul R Wilson. The memory fragmentation problem: solved? In *ACM SIGPLAN Notices*, volume 34, pages 26–36. ACM, 1998.

[146] Vernon KC Bumgardner and Victor W Marek. Scalable hybrid stream and hadoop network analysis system. In *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*, pages 219–224. ACM, 2014.

[147] VK Cody Bumgardner, Victor W Marek, and Ray L Hyatt. Collating time-series resource data for system-wide job profiling. In *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP*, pages 1043–1048. IEEE, 2016.

[148] VK Cody Bumgardner, Victor W Marek, Caylin D Hickey, and Kanna Nandakumar. Constellation: A secure self-optimizing framework for genomic processing. In *e-Health Networking, Applications and Services (Healthcom), 2016 IEEE 18th International Conference on*, pages 1–6. IEEE, 2016.

[149] Aiko Pras, Ramin Sadre, Anna Sperotto, Tiago Fioreze, David Hausheer, and Jürgen Schönwälder. Using netflow/ipfix for network management. *Journal of Network and Systems Management*, 17(4):482–487, 2009.

[150] Tom White. *Hadoop: the definitive guide*. O'Reilly, 2012.

[151] The Apache Software Foundation. Apache HBase. `http://hbase.apache.org`, 2013. [Online; accessed 20-December-2013].

[152] Cristian Estan, Ken Keys, David Moore, and George Varghese. Building a better NetFlow. *SIGCOMM Comput. Commun. Rev.*, 34(4):245–256, August 2004.

[153] MPLS. MPLS. `http://www.cisco.com/en/US/products/ps6557/products_ios_technology_home.html`, 9 2013.

[154] Fprobe. Fprobe. `http://sourceforge.net/projects/fprobe`, 9 2013.

[155] Steve Vinoski. Advanced Message Queuing Protocol. *Internet Computing, IEEE*, 10(6):87–89, 2006.

[156] Apache Software Foundation. Apache Storm. `http://storm.apache.org/`, 1 2017.

[157] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[158] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

[159] Esper. Esper. `http://esper.codehaus.org/`, 9 2013.

[160] Vanessa Wang, Frank Salim, and Peter Moskovits. *The Definitive Guide to HTML5 WebSocket*. Apress, 2012.

[161] Ian Fette and Alexey Melnikov. The WebSocket protocol. *IETF*, 2011.

[162] UKAT. UK DLX. `http://www.uky.edu/ukat/hpc`, 2016. [Online; accessed 18-January-2016].

[163] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[164] Adaptive Computing Enterprises. Moab workload manager administrator's guide, version 7.2. 6, January 2014.

[165] Matthew L Massie, Brent N Chun, and David E Culler. The Ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[166] Lars George. *HBase: The definitive guide*. O'Reilly Media Inc., 2011.

[167] Leo J Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science*, pages 8–21. IEEE, 1978.

[168] Peter Grabusts et al. The choice of metrics for clustering algorithms. In *Proceedings of the 8th International Scientific and Practical Conference*, volume 2, 2011.

[169] Abraham Silberschatz, Peter B Galvin, Greg Gagne, and A Silberschatz. *Operating system concepts*, volume 4. Addison-Wesley Reading, 1998.

[170] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *ACM SIGOPS Operating Systems Review*, 36(5):45–57, 2002.

[171] Asit K Mishra, Joseph L Hellerstein, Walfredo Cirne, and Chita R Das. Towards characterizing cloud backend workloads: insights from google compute clusters. *ACM SIGMETRICS Performance Evaluation Review*, 37(4):34–41, 2010.

[172] Michael J Brusco and J Dennis Cradit. A variable-selection heuristic for k-means clustering. *Psychometrika*, 66(2):249–270, 2001.

[173] Qasim Ali, Vladimir Kiriansky, Josh Simons, and Puneet Zaroo. Performance evaluation of HPC benchmarks on VMware's ESXi server. In *Euro-Par 2011: Parallel Processing Workshops*, pages 213–222. Springer, 2012.

[174] Liuhua Chen and Haiying Shen. Consolidating complementary VMs with spatial/temporal-awareness in cloud datacenters. In *INFOCOM, 2014 Proceedings IEEE*, pages 1033–1041, April 2014.

[175] Alex D Breslow, Leo Porter, Ananta Tiwari, Michael Laurenzano, Laura Carrington, Dean M Tullsen, and Allan E Snavely. The case for colocation of high performance computing workloads. *Concurrency and Computation: Practice and Experience*, 2013.

[176] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[177] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[178] Alexander Heinecke, Michael Klemm, and Hans-Joachim Bungartz. From GPGPU to many-core: NVidia Fermi and Intel many integrated core architecture. *Computing in Science & Engineering*, 14(2):78–83, 2012.

[179] Maya B Gokhale and Paul S Graham. *Reconfigurable computing: Accelerating computation with field-programmable gate arrays*. Springer Science & Business Media, 2006.

[180] M. Calzarossa and G. Serazzi. Workload characterization: A survey. *Proceedings of the IEEE*, 81(8):1136–1150, Aug 1993.

[181] Walfredo Cirne and Francine Berman. A comprehensive model of the supercomputer workload. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 140–148. IEEE, 2001.

[182] A. Khan, X. Yan, Shu Tao, and N. Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 1287–1294, April 2012.

[183] Francois Jeanmougin, Julie D Thompson, Manolo Gouy, Desmond G Higgins, and Toby J Gibson. Multiple sequence alignment with clustal x. *Trends in biochemical sciences*, 23(10):403–405, 1998.

[184] Sabeel Ansari, SG Rajeev, and HS Chandrashekar. Packet sniffing: A brief introduction. *IEEE potentials*, 21(5):17–19, 2002.

[185] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, RFC Editor, April 1992.

[186] Mark Berman, Jeffrey S Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. Geni: A federated testbed for innovative network experiments. *Computer Networks*, 61:5–23, 2014.

[187] Anastasius Gavras, Arto Karila, Serge Fdida, Martin May, and Martin Potts. Future internet research and experimentation: the FIRE initiative. *ACM SIGCOMM Computer Communication Review*, 37(3):89–92, 2007.

[188] Luis Sanchez, José Antonio Galache, Veronica Gutierrez, Jose Manuel Hernandez, Jesús Bernat, Alex Gluhak, and Tomás Garcia. Smartsantander: The meeting point between future internet research and experimentation and the smart cities. In *Future Network & Mobile Summit (FutureNetw), 2011*, pages 1–8. IEEE, 2011.

[189] Paul Grace, Juan Echevarria, Martin Serrano, Luis Sanchez, David Gomez, Miao Ting, Jaeseok Yun, Tarek El Saleh, and Francois Carrez. Analysis of IoT platforms and testbeds: FIESTA D2. 2. *1*, 2015.

[190] Glenn Ricart. US Ignite testbeds: Advanced testbeds enable next-generation applications. In *Teletraffic Congress (ITC), 2014 26th International*, pages 1–4. IEEE, 2014.

[191] BIRD. Bird. `http://bird.network.cz/`, 2016. [Online; accessed 26-June-2016].

[192] Internet2 AL2S Roadmap. Internet2 AL2S Roadmap. `http://www.internet2.edu/products-services/advanced-networking/layer-2-services/al2s-roadmap`, 2016. [Online; accessed 26-June-2016].

[193] David D Clark. Adding service discrimination to the internet. *Telecommunications Policy*, 20(3):169–181, 1996.

[194] Ted Faber and Rob Ricci. Resource description in GENI: Rspec model. In *Presentation given at the Second GENI Engineering Conference (March 2008)*, 2008.

[195] Cédric Adjih, Emmanuel Baccelli, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noel, Roger Pissard-Gibollet, Frédéric Saint-Marcel, Guillaume Schreiner, Julien Vandaele, et al. FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed. In *Proceedings of the 2nd IEEE World Forum on Internet of Things (WF-IoT)*, 2015.

[196] S. Cirani, G. Ferrari, N. Iotti, and M. Picone. The IoT Hub: A fog node for seamless management of heterogeneous connected smart objects. In *Sensing, Communication, and Networking - Workshops (SECON Workshops), 2015 12th Annual IEEE International Conference on*, pages 1–6, June 2015.

[197] Lixia Zhang, Deborah Estrin, Jeffrey Burke, Van Jacobson, James D Thornton, Diana K Smetters, Beichuan Zhang, Gene Tsudik, Dan Massey, Christos Papadopoulos, et al. Named Data Networking (NDN) project. *Relatório Técnico NDN-0001, Xerox Palo Alto Research Center-PARC*, 2010.

[198] Alan Murphy. Enabling long distance live migration with F5 and VMware vMotion. `https://f5.com/resources/white-papers/enabling-long-distance-live-migration-with-f5-and-vmware-vmotion`, 2011.

Vita

- Education
  - University of Kentucky, Lexington, KY,
    B.S. Computer Engineering, 2003 - 2009
- Professional Positions
  - University of Kentucky, Lexington, KY,
    Director of Research Computing, 2016 - Present
  - University of Kentucky, Lexington, KY,
    Chief Technology Architect, 2010 - 2016,
  - University of Kentucky, Lexington, KY,
    Director Enterprise Computing and Development, 2008 - 2010
  - University of Kentucky, Lexington, KY,
    Lead Systems Programmer, 2002 - 2008
  - Netsource, Lexington, KY,
    Director of Technical Services, 2001 - 2002
  - Nuvox, Lexington, KY,
    Senior Technical Manager, 2000 - 2001
  - Communitronics Inc., Lexington, KY,
    Senior Network Engineer, 1997 - 2000
  - Pomeroy Computer Resources,
    Lexington, KY, Systems Consultant, 1996 - 1997
  - Morehead State University, Morehead, KY,
    Physics/IT Co-op, 1995 - 1996
- Scholastic Honors
  - Outstanding PhD Student in Computer Science,
    University of Kentucky, Spring 2016.
- Publications
  - Bumgardner, V.K.C, et al. "Cresco: A distributed agent-based edge computing framework." The International Workshop on Green ICT and Smart Networking (GISN 2016). IEEE, 2016.

– Bumgardner, V.K.C, et al. "Constellation: A Secure Self-Optimizing Framework for Genomic Processing." 18th International Conference on E-health Networking, Application & Services (HealthCom). IEEE, 2016.

– Bumgardner, V.K.C, and Victor W. Marek. "Collating time-series resource data for system-wide job profiling." Proceedings of IEEE/IFIP International Workshop on Analytics for Network and Service Management AnNet. IEEE, 2016

– Bumgardner, Vernon. OpenStack in Action. NewYork: Manning, 2016. Print

– Bumgardner, V.K.C, and Victor W. Marek, "Educational Applications of Cloud", Book Chapter "Encyclopedia of Cloud Computing, Wiley-IEEE Publication, 2016, Print.

– Bumgardner, V.K.C, and Victor W. Marek. "Scalable hybrid stream and Hadoop network analysis system." Proceedings of the 5th ACM/SPEC international conference on Performance engineering. ACM, 2014.

– Mueller, T., Neelakantan, S., Rienzi, E., Mijatovic, B., Castrignano, A., Pike, A., Bumgardner, C. 2012. Cloud Computing, Web-Based GIS, Terrain Analysis, Data Fusion, and Multivariate Statistics for Precision Conservation in the 21st Century.

– Neelakantan, S., T.G. Mueller, B. Lee, B. Lee, P. Finnell, V. Bumgardner, and D. Carey. 2011. Web 2.0 spatial data browser for visualizing land-use assessment information from soil surveys. J. of Soil and Water Conservation and Management. 66:37A-39A

– Immersive Real-Time Tele-Collaboration of Complex Volumetric Medical Imaging for Surgical Planning. Mastrangelo MJ Jr, Sheetz M, Bumgardner C, George I, Mylniec P, Bellinghausen J. Internet2/National Library of Medicine / Radiological Society of North America Annual Meeting, Chicago, Illinois. November 30, 2004

– Alex Gandsas, M. D., Katherine Mclntire, Kevin Montgomery, Cody Bumgardner, and Linda Rice. "The personal digital assistant (PDA) as a tool for telementoring endoscopic procedures." Medicine Meets Virtual Reality 12: Building a Better You: The Next Tools for Medical Education, Diagnosis, and Care 98 (2004): 99.