# Implementing a RESTful Software Architecture to Coordinate Heterogeneous Networked Embedded Devices

Jason T. Davis

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science & Applications

Eli Tilevich, Chair
Ali R. Butt
H. Pat Artis

September 10, 2021
Blacksburg, Virginia

Keywords: Distributed Computing, Embedded Systems, RESTful Architecture

# Implementing a RESTful Software Architecture to Coordinate Heterogeneous Networked Embedded Devices

Jason T. Davis

(ABSTRACT)

Modern embedded systems—autonomous vehicle-to-vehicle communication, smart cities, and military Joint All-Domain Operations—feature increasingly heterogeneous distributed components. As a result, existing communication methods, tightly coupled with specific networking layers and individual applications, can no longer balance the flexibility of modern data distribution with the traditional constraints of embedded systems. To address this problem, the investigation herein presents a domain-specific language, designed around the Representational State Transfer (REST) architecture, most famously used on the web. Our language, called the Communication Language for Embedded Systems (CLES), supports both traditional point-to-point data communication and management and allocation of decentralized distributed processing tasks. To meet the traditional constraints of embedded execution, CLES' novel runtime allocates processing tasks across a heterogeneous network of embedded devices, overcoming limitations from other modern distribution methods: centralized task management and limited operating system integration. CLES was evaluated with performance micro-benchmarks, implementation of distributed stochastic gradient descent, and application to the design of versatile stateless services for vehicle-to-vehicle communication and military Joint All-Domain Command and Control (JDAC). From this evaluation, it was determined that CLES meets the data distribution needs of realistic cyber-physical embedded systems.

# Implementing a RESTful Software Architecture to Coordinate Heterogeneous Networked Embedded Devices

Jason T. Davis

(GENERAL AUDIENCE ABSTRACT)

As computers become smaller, cheaper, more powerful, and energy efficient, they are increasingly used in cyber-physical systems such as planes, trains, and automobiles, as well as large-scale networks such as power plants and smart cities. The field of embedded computing is facing new challenges involving the communication and coordination of large numbers of different devices. Some of the software challenges within embedded device communications are: flexibility both in ability to run on different devices and use different communication links such as cellular, Wi-Fi, or Bluetooth, performance constraints of low-power embedded devices, latency and reliability to ensure safe operations, and the schedule and cost of development. To address these challenges, this thesis presents a new programming language, designed around the Representational State Transfer (REST) architecture, most famously used in HTTP to drive the web. Our language, called the Communication Language for Embedded Systems (CLES), supports both traditional point-to-point data communication designed to prioritize latency and reliability, as well as a standalone application or runtime that can be run on an embedded device to accept requests for processing tasks. CLES and its supporting Software Development Kit (SDK) is designed to allow for quick and cost effective development of flexible low-latency device to device communications and large scale distributed processing on embedded devices.

# Dedication

*This Thesis is dedicated to my family who has always encouraged my pursuit of further education, my fiancée Sammi Rocker for helping me through the challenges of completing this degree, and my employer and coworkers for giving me the flexibility and support professionally to tackle this rewarding experience.*

# Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Eli Tilevich, whose continued guidance enabled me to get through this process from start to finish.

I would also like to thank the rest of my committee, Dr. H. Pat Artis for advising me both personally and professionally since the beginning of my undergraduate program, and Dr. Ali Butt for his unique teaching approach to the complex course of Operating Systems that inspired me to pursue a career in the field.

Thank you to my parents, who instilled in me a strong work ethic and value of higher education from a young age. Without their support and encouragement throughout my life, I would not have been able to achieve this academic milestone.

Finally, I would like to thank the Virginia Tech Computer Science Department, and especially the Department Head Dr. Cal Ribbons and Graduate Director Dr. Cliff Shaffer. The BS/MS program and the flexibility of the department created the opportunity for me to pursue a graduate degree from Virginia Tech while working full-time from across the country. Thank you for accepting my unique challenges and constraints and giving me the opportunity and tools to succeed.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

CLES Communication Language for Embedded Systems

DSL Domain Specific Language

HTTP Hypertext Transfer Protocol

JDAC Joint-All-Domain Command and Control

OS Operating System

P2P Point-to-Point

SGD Stochastic Gradient Descent

V2V/V2X Vehicle to Vehicle/Vehicle to Everything

CLES is the RESTful language definition and middleware presented herein as a solution for embedded device communications and coordination.

A Domain Specific Language is a programming language developed to solve a specific problem such as postgresql for database control.

The Hypertext Transfer Protocol is a RESTful language designed for distributing webpages and other hypertext media across the internet

Joint-All-Domain Command and Control is the U.S. Military development effort of incorporating communication and coordination across multiple services and their technologies.

The Operating System is the software running on a device that hosts all running applications and manages device resources such as processor, memory, and networking.

Point-To-Point communications are the direct passing of information between two devices or applications.

Stochastic Gradient Descent is algorithm for optimizing an objective function through iterative using its error rate derivative to approach a solution.

Vehicle to Vehicle/Vehicle to Everything communications are a sub-field of autonomous vehicles focused on the passing of information between autonomous vehicles and infrastructure to improve safety and reliability of self-driving cars.

# Chapter 1

# Introduction

Modern embedded systems are increasingly heterogeneous, collaborative, and networked. Disparate systems, each with its own computing architecture, operating system, and purpose coordinate with each other to achieve a common goal. Their communication and coordination functionality traditionally is provided via a custom, highly optimized and specialized protocol for each pair of connected systems. For safety-critical cyber-physical systems, this approach is deemed as required to meet the timeliness and reliability requirements. Unfortunately, the resulting low-level, platform-specific code is inflexible, not reusable, and difficult to write, extend, and maintain.

Web development coordinates heterogeneous distributed components via the RESTful architectural style [14] with its ubiquitous HTTP communication. HTTP supports stateless communication through flexible plain-text messages while maintaining common functionality and semantics through a limited set of commands referred to as "verbs". This allows for easy implementation across different platforms.

The field of High-powered computing primarily manages distributed workloads with container orchestration platforms, such as Docker Swarm and Kubernetes. Unfortunately, developers cannot apply these proven solutions to embedded systems because of the limitations existing container orchestration platforms. These platforms rely on proprietary software architectures, have strict operating system limitations, required centralized management of processing tasks, and have non-deterministic timing performance.

To support the creation of heterogeneous networked solutions easily integrated into embedded systems, this Thesis presents CLES[1], a Domain Specific Language (DSL) for implementing simple, stateless, communication protocols in the RESTful architectural style [14]. With its simple syntax and Application Programming Interface (API), CLES keeps the complexity of networking architecture and protocols abstracted from the application developer, thus providing a flexible and robust communication mechanism designed to minimize the required learning curve and development burden. Designed to meet the communication and coordination requirements of embedded systems, CLES is implemented in standard C++. CLES integrates point-to-point communication to maintain deterministic timeliness constraints, interfacing with the OSI Internet Protocol (IP) network stack instead of low-level wireless protocols for flexibility. CLES also provides a powerful runtime that meets advanced distributed computing requirements, such as decentralized task allocation.

This thesis makes the following contributions:

- *An application of the RESTful architecture for the communication and coordination needs of modern embedded systems.* We demonstrate how the proven benefits of REST, including first-class support for heterogeneity, uniformity, and simplicity, can be extended to embedded systems, without sacrificing this domain's timeliness constraints.

- *CLES—a platform independent DSL with supporting SDK and runtime for implementing RESTful architecture's communication and coordination in embedded systems.* The CLES runtime introduces novel asymmetric task registration and remote resource access, in order to meet the unique execution constraints of its target domain.

- *An empirical evaluation of CLES with representative performance micro-benchmarks, implementation of distributed stochastic gradient descent, and realistic case studies.*

[1] CLES stands for **C**ommunication **L**anguage for **E**mbedded **S**ystems

# Chapter 2

# Background

The field of distributed embedded systems consists of a wide variety of different problems. These problems can roughly be collected into two distinct categories:

- Heterogeneous device to device communications with driving design constraints of latency, safety, and reliability.
- Distributed computing though middleware supported orchestration of computing tasks across a network of embedded devices.

This Thesis explores heterogeneous device communication in the context of vehicle-to-everything (V2X) and vehicle-to-vehicle (V2V) communications through the representative problem space of vehicle platooning. The traditional methods for point-to-point communication involve either strictly defining data transmission via an inflexible, custom serialized packet with an Interface Control Document (ICD), or by defining the data format within the data link layer itself, such as in traditional military communications Link-16 and Multifunction Advanced Data Link (MADL). We argue that a RESTful request and response communication model, similar to the ubiquitous HTTP, supported through a simple C++ support package designed with in interface similar to the Java `net.http` package, can offer a more flexible point-to-point alternative that meets the same execution requirements with significantly reduced development time and complexity.

This Thesis explores distributed task orchestration in the context of heterogeneous, distributed sensor networks, in which devices have a wide range of computing power and specialized hardware. These networks must efficiently distribute processing tasks across multiple devices for real-time situational awareness. We argue that our approach can support decentralized, asymmetric, task allocation through simple portable C++ plugins loaded by the CLES runtime. When it comes to software engineering benefits, we argue that our approach offers an attractive alternative to Docker Swarm and Kubernetes, the state-of-the-art container orchestration platforms, commonly used in data-centers and web-service domains, and explored by the U.S. Air Force for supporting Joint-All-Domain Command and Control [8, 15].

## 2.1 V2X

Vehicle to Everything (V2X) contains the sub-components of Vehicle to Vehicle (V2V), Vehicle to Infrastructure (V2I), and Vehicle to Pedestrian (V2P) communications. These paradigms for communication infrastructure can be divided into two categories based on purpose and minimum requirements. The first category is real-time, safety-critical information for safe vehicle operation, most commonly used for V2V communication. A representative example is vehicle platooning: vehicles share their dynamics and localization to autonomously travel in a group, reducing collisions, fuel usage, and driver workload. This category of safety-critical communications with timeliness and reliability requirements also includes some V2I communication, such as stop lights and pedestrian crossing. The second category is non-realtime, general purpose information, such as traffic congestion, speed limits, and weather information. Most V2I data streams involve status information that is broadcast at regular intervals. The critical design drivers for these non time-sensitive

broadcasts are ease of integration, backwards compatibility, and flexibility.

## 2.2 Containers: Docker

Containers are a hybrid between standalone applications on a host Operating System (OS) and a Virtualized OS or Virtual Machine (VM). Containers combine applications and associated dependencies into a single deployable package, similar in functionality to deploying a VM to a target, thus eliminating the guess work of dependency management and potential software environment conflicts with other applications. Virtual Machines protect applications from outside conflicts or corruption via virtualized isolated memory and OS system calls, but at the cost of significant performance overhead due to the need to translate VM system calls to the host system calls. Containers solve this problem by providing isolated system interfaces directly to the host OS kernel, as well as virtualized memory and I/O for isolation and protection. Containers are compiled to run directly on the host architecture and OS, thus removing the overhead of architecture virtualization [11].

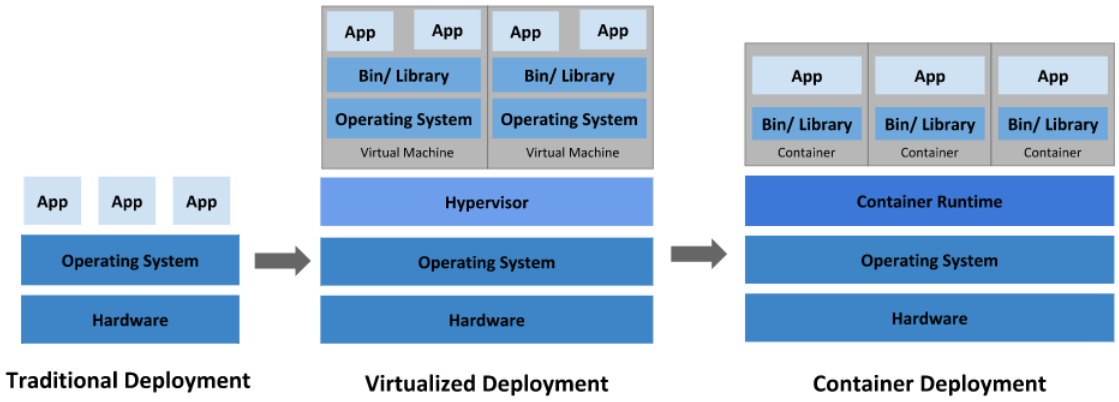


Figure 2.1: Docker: Container Deployment [2]

## 2.3 Container Orchestration

The portability of containers makes them a good fit for distributed computing. The concept behind container orchestration associates containers with traditional computing tasks, so they can be deployed dynamically to support the distribution of processing tasks across multiple machines. For example, if a user submits a task to a container orchestration server, the server will forward the task to a separate computing node and start the associated container for the task on this node.

There are two well established usages of container orchestration. One is high powered computing for load-balancing large, distributable workloads, such as training machine learning models across a cloud server farm. Another is server back-end load balancing, in which a container is launched to handle each user connection.

Container orchestration platforms include Docker Swarm, Kubernetes, Amazon Elastic Container Service (ECS), Red Hat Open Shift, and others [16]. This work uses Kubernetes as a reference architecture, due to its broad acceptance, portability to offline non-cloud hosted servers, and available documentation.

### Kubernetes

Kubernetes is a Docker-based container orchestration engine, whose deployment comprises a central coordination node, known as the Control Plane, with optional fallback Control Planes and a collection of worker nodes (Pods) [22]. Kubernetes hosts Docker containers in its Control Plane with the API Server and dynamically distributes and launches the containers across the Pods. Kubernetes supports health monitoring of Pods, automated restart for worker failure recovery, and dynamic load balancing. Kubernetes can also dynamically

update Docker containers with soft roll-outs, a capability that has solidified Kubernetes as the dominant solution for Continuous Integration and Continuous Delivery (CI/CD).

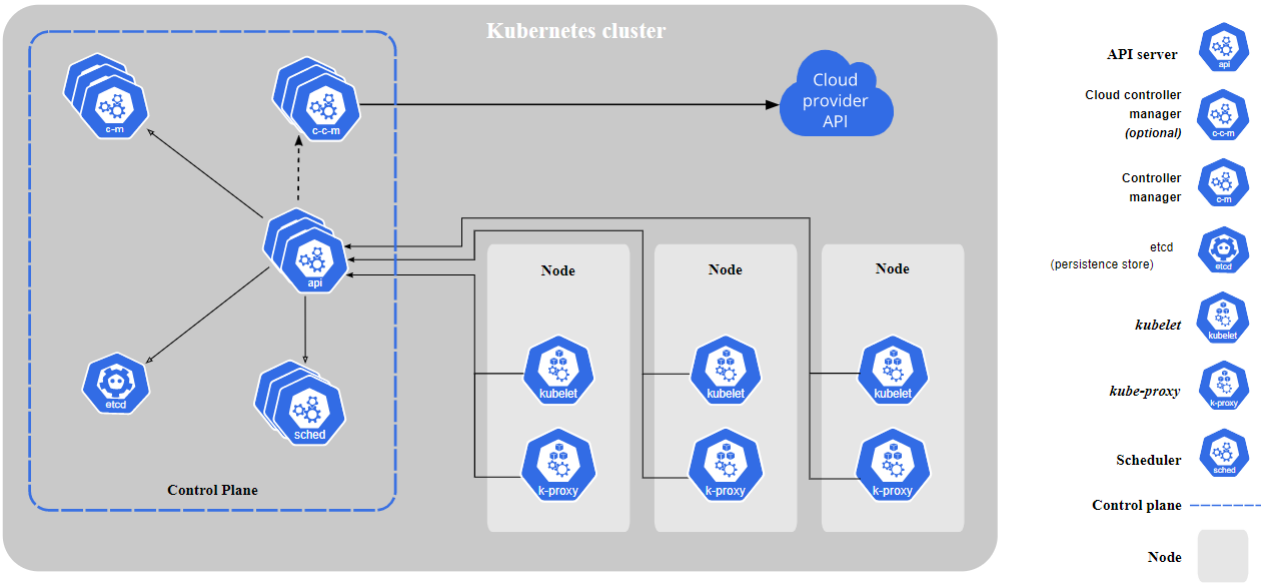


Figure 2.2: Kubernetes Cluster [2]

## 2.4 RESTful Architecture

The RESTful architectural style codifies several design principles of client-server communication, with stateless servers and a communication protocol based on a fixed number of Verbs combined with an infinite number of Nouns [14]. The most widely used implementation of the RESTful style is the Hypertext Transfer Protocol (HTTP) used on the web, with its verbs: GET, POST, PUT, PATCH, and DELETE. Combined with URL nouns and optional additional parameters, these verbs form the foundation of the modern internet infrastructure. For example, requesting the webpage for Google.com is as easy as sending the verb GET with path (noun) "/" to www.google.com.

## Applying REST beyond the Web

The RESTful architectural style has been successfully applied to implement communication in domains other than the Web. One such domain is co-located mobile devices. The Resource Query Language (RQL) [6, 25], implemented as an application layer communication language and runtime for Android and iOS mobile devices, exposes a public interface for common mobile resources that include GPS, cell-tower connection, and processing cycles. As its key design drivers, RQL provides heterogeneous device interoperability, flexibility, and ease of integration.

Following the RESTful design principles, RQL defines its own set of verbs and semantics. While sharing some functionality with HTTP, RQL is designed for the express purpose of device-to-device communication and coordination. RQL removes some of the flexibility and portability of RESTful architrave by requiring mobile devices to manage all networking and data distribution through the RQL runtime.

Despite its innovative integration of the RESTful style and device-to-device oriented DSL, RQL is tailored specifically for the mobile device space. The RQL runtime is designed for and coupled with the specific features of the Android and the iOS platforms; it leverages the mobile specific Android Interface Definition Language (AIDL) [10] for networking. Its platform-specific design, inability to meet deterministic timing constraints, and non-extensible runtime make RQL inapplicable to meet the advanced requirements of distributed communication and coordination in embedded systems beyond mobile devices.

# Chapter 3

# Problem Domain: Embedded Distributed Computing

Both the automotive and defense industries are paving the way for large-scale, decentralized distributed computing systems. The defining characteristics of their distributed networks are dynamic connectivity graphs, heterogeneous systems, real-time operating systems, low-power devices, decentralized requirements, high reliability, and limited bandwidth. Kubernetes is currently being explored as a solution for both the U.S. Air Force [15] and the automotive industry. Kubernetes and other container orchestration solutions are not ideal for this problem space, as they are typically centralized, limited to non-real-time OS, inapplicable to hardware-integrated platforms (e.g., FPGA data processing), and lacking non-container interfaces into the distributed processing network.

## 3.1 Use Case: Vehicle Platooning

A fundamental challenge of autonomous vehicles and vehicle to everything (V2X) communication is vehicle platooning. *An automated vehicle platoon* is a series of vehicles, most notably trucks, that maintain a group (platoon) on the highway, with a software control module in each vehicle controlling each their speed and steering. With its constituent vehicles communicating and coordinating with each other, a platoon increases the safety and

energy efficiency of highway travel [21].

A traditional Vehicle to Vehicle (V2V) communication problem, platooning is constrained by networking challenges and algorithmic design. Notably, 802.11p and 5G are the two proven network solutions that satisfy the minimum requirements for latency, reliability, and security for vehicle platooning [4]. Additionally, Swaroop [23], Arefizadeh [21], and Liangyi [24] introduced algorithms that define the minimum data transmission required to achieve stable platoons. One question that remains open is what kind of software architecture and abstractions are needed to realize practical platooning solutions at the scale considered by auto-manufacturers and government authorities.

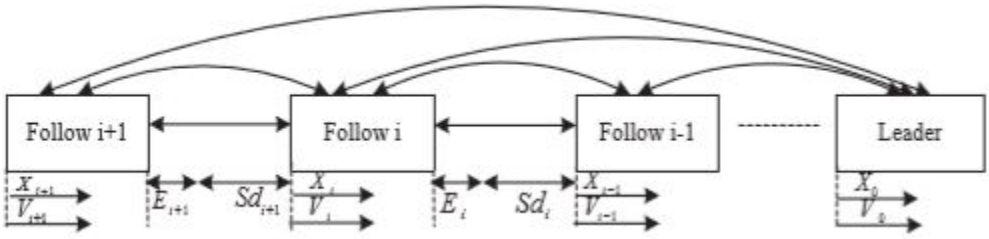


Figure 3.1: Platooning Networking Structure [23]

As baseline networking protocols, our implementation references 802.11p and 5G, which both support OSI IP [19] and we use the defined minimum dataset required for stable platooning: lead vehicle localization data (UTC Time, latitude, longitude, Velocity, and Acceleration), and localization data of the vehicle immediately ahead of each vehicle [23].

## Traditional challenges

The traditional methods for point-to-point communication involve either defining a transmission via an inflexible, interface control document (ICD) serialized packet, or defining the data structure within the data link layer itself, such as in military communications Link-16 and MADL.

In this use case, the traditional approach that integrates the communication functionality directly into the data link layer increases development time, reduces flexibility for enhancements, hinders upgradability, and complicates integration with other wireless solutions. Using an ICD defined serialized packet over a network abstraction such as IP for data transmission pulls the complexity into the application layer which increases hardware interoperability, but suffers from similar limitations. The time to market for this solution can be very quick, and the integration time is minimal, but the ICD's restricted structure of the packet definition heavily constrains flexibility and upgradability.

More modern solutions for this data distribution problem include ActiveMQ and DDS; However, ActiveMQ requires a centralized data broker. While DDS can be configured to be decentralized, DDS requires a significant development effort to properly support vehicle to vehicle communication. This is because DDS functions primarily as data middleware one level above IP without high-level language bindings. This leaves the application layer of defining the structure of the data transmission.

Lastly, Kubernetes and other container orchestration platforms provide no support for common real-time OS' used in in the automotive industry, VxWorks and QNX, removing them from consideration.

## 3.2 Use Case: Joint All-Domain Command and Control (JDAC)

One of the core design requirements for the U.S. Military Joint All-Domain Command and Control (JDAC) is integration of heterogeneous assets to form a distributed sensor network that both informs decision makers and improves strike capabilities [8]. The benefits behind

the a networked military include air, land, and sea dominance and faster and more informed decision making to both protect the United States and its service men and women and reduce the collateral damage and risk to civilians. JDAC includes heterogeneous, distributed sensor networks, in which devices have a wide range of computing power and specialized hardware. These networks must efficiently distribute processing tasks across multiple devices for real-time situational awareness and cooperative action.


Figure 3.2: DARPA Mosaic Warfare [1]

For the purpose of demonstrating the core challenges required to meet this goal, we construct an example concept of operations (CONOPS). This CONOPS includes two different aircraft, one satellite, and a mission operations center. The first aircraft is a reconnaissance aircraft, equipped with an Electro-Optical/Infra-Red (EO/IR) camera capable of sending encoded Full Motion Video (FMV). The second aircraft is an fighter jet with its own sensor suite and weapon system capable of launching a ground strike. The satellite hosts a powerful processor, machine learning algorithms, and other sources of data for fusion.

For this network architecture the satellite only has a data link to the reconnaissance aircraft.

Both aircraft can communicate through data links with the mission center, but not with each other. This sparsely connected graph of connectivity reasonably represents a realistic scenario with modern, proprietary, incompatible data links in the U.S. Military.

The first operational requirement for the reconnaissance aircraft is to collect FMV and route the data stream to the satellite for machine learning driven processing. The satellite returns a "Track" for all targets synthesized from the data. The reconnaissance aircraft sends Track information to the mission control station. The mission control station enables officers to make informed decisions. Finally, the operation lead sends a strike command to the fighter aircraft, with the associated Track from the satellite/reconnaissance aircraft fusion. The final result is a more accurate strike achieved by fusing the supplied Track with data on-board the strike vehicle. More importantly, the decision to strike can be made quickly and confidently by reducing the data-to-decision time provided by the sensor network integrated ground station.

### 3.2.1 Traditional challenges

The traditional approach to developing this sort of joint operation would involve each primary contractor for the different platforms to develop custom interfaces between each asset. These interfaces would require long development and test cycles, and commonly be too specific to allow for re-usability as this network is expanded to support additional assets.

### 3.2.2 Existing Approach

The CONOPS proposed above is similar to the design drivers cited for the U-2 Kubernetes integration [15]. Although Kubernetes is a powerful tool for distributing tasks, it is limited to certain OS. Only some of the highly heterogeneous participating assets in a Joint All-Domain

Operation can host Kubernetes. The approach currently being pursued by this U-2 effort is requiring each new participating platform in the network to carry an additional computer dedicated to hosting Kubernetes and other JDAC compliant software. This is expensive and in the case of lightweight, single-purpose platforms, outside the acceptable weight and power limits.

# Chapter 4

# RESTful Architecture for Distributed Embedded Systems

Our Communication Language for Embedded systems (CLES) and Software Development Kit (SDK) support a broad problem space with a wide variety of inter-process and inter-device scenarios, while overcoming some of the most salient constraints of embedded systems development. Incidentally *clé[s]* is "wrench" in French, and our objective has been to create a universal wrench for the communication and coordination of embedded systems.

For low latency point-to-point communications, we argue that a RESTful request and response communication model embedded into the parent application, similar to the ubiquitous Java net.http package, can offer a more flexible point-to-point alternative that meets the same execution requirements as the traditional ICD or data link design patterns. For distributed workloads, our approach supports decentralized, asymmetric, task allocation through simple portable C++ plugins loaded by a standalone runtime.

## 4.1 Requirements

Distributing an application across devices removes the applicability of all inter-process communication within an OS, such as shared memory, memory mapped I/O, pipes, OS messaging, semaphores, etc. The next descriptor of system requirements is *heterogeneity*: systems

running on any hardware or OS should be able to interface with the communication functionality.

The example problem of a sensor network implies that the network consists of at least some nodes that are highly specialized, possibly low-powered devices with an array of sensors. These devices need to maintain their network coordination with *minimal overhead* and software.

The final key requirement to consider is *decentralized*. In our problem domain, one cannot assume that the node serving as the leader or manager will always be present. Decentralized distributed processing is important for distributed sensor networks, as well as V2X and military applications, because often times there is no clear "leader" and the computing cluster must be able to maintain some functionality with any node missing.

## 4.2 CLES Core Language Design

Despite their vastly dissimilar objectives and limiting factors, the use cases of vehicle platooning and military JDAC represent some of the most prevalent problems in the development of modern day embedded systems.

To satisfy the additional design drivers of programmability, flexibility, and interoperability, our CLES domain specific language features a limited but powerful set of verbs and a plain-text, JSON-formatted, response structure. The primary benefits to the RESTful architectural style is the ability to expose a limitless set of capabilities as nouns while constraining their interface semantics to a simple set of 5 verbs (Table 4.1).

A unique benefit of following the RESTful style for the automotive and defense industries is that the communication interfaces between applications need not expose the back-end

functionality. That is, upon receiving a declarative instruction comprising a verb applied to a noun, a remote component processes the verb in an appropriate fashion and returns only the result back to the caller, thus protecting any proprietary or classified functionality or software.

CLES addresses the minimal latency and overhead requirements with a lightweight CLES service for point-to-point communication. With this solution, the computationally weaker nodes do not need to support a full CLES runtime, only a direct point-to-point interface, designed to integrate directly into a host application. CLES leverages this point-to-point service to solve the timeliness requirements as discussed in Section 4.4. This service has been designed and explored to solve the challenges present in the *Platooning* use case.

The requirement of *decentralized* task management was tackled by designing and implementing a separate standalone runtime for CLES that allows for registration of tasks that can then be invoked remotely. Rather than integrating into an application like the point-to-point service, the CLES runtime is designed to be run as a standalone application, with a single runtime for each embedded device. Each device's runtime accepts processing tasks remotely submitted using the RESTful DSL. The runtime has been designed and discussed in future sections to solve the JDAC distributed processing problems while maintaining flexibility and interoperability with the point-to-point service to allow low-power devices to submit tasks using only the lightweight point-to-point interface.

Table 4.1: CLES Verbs

| Verb | Usage |
|---|---|
| Pull | Get Data Once |
| Push | Send Parameter and Data |
| Delegate | Send Parameter, Get Result |
| Bind | Register to Get Persistent Updates |
| Update | Send Update to Bound Users |

$$\langle \text{CLES Command} \rangle \models \langle \text{verb} \rangle \langle \text{noun} \rangle \quad (4.1)$$
$$\langle \text{verb} \rangle \models \textit{pull} \mid \textit{push} \mid \textit{delegate} \mid \textit{bind} \mid \textit{update} \quad (4.2)$$
$$\langle \text{noun} \rangle \models \langle \text{remote target} \rangle : \langle \text{task name} \rangle / \langle \text{task parameters} \rangle \quad (4.3)$$
$$\langle \text{remote target} \rangle \models \langle \text{generic string} \rangle \quad (4.4)$$
$$\langle \text{task name} \rangle \models \langle \text{generic string} \rangle \quad (4.5)$$
$$\langle \text{task parameters} \rangle \models \langle \text{parameter} \rangle \mid \langle \text{task parameters} \rangle \langle \text{parameter} \rangle \quad (4.6)$$
$$\langle \text{parameter} \rangle \models \langle \text{generic string} \rangle \quad (4.7)$$
$$\langle \text{generic string} \rangle \models [a - zA - Z0 - 9]+ \quad (4.8)$$

Figure 4.1: CLES Backus-Naur Form (BNF) Grammar

## 4.3 CLES SDK Interface Definition

The CLES library interface for integrating with applications is simple but powerful. The interface constructs a *CLES_Service*, which requires a device name to expose externally and a network interface with the port to bind to. After constructing a service, each CLES verb has its own registration interface, whose *Register<Verb>Function* definitions bridge the application to the CLES external interface. For example, to expose the capability for other devices to retrieve this application's timestamp, the developer would first craft a function such as *CLES_Response getSystemTime()* which internally fills and returns a *CLES_Response* object with a key-value pair for system time. Next, the developer would associate this function with the verb PULL by calling *RegisterPullFunction"systemTime", getSystemTime)*. After registering all verbs and associated functions, the *CLES_Service* can be started with *run()* to accept external connections. Finally, a second device with its own CLES service could request the system time from the first device by calling *makeCLESRequest("PULL device1:systemTime")*. To pass parameters, such as timezone into *getSystemTime(vector<string>args)*, the CLES Request can be extended as per the DSL grammar (Figure 4.1), with the example: "*PULL device1:systemTime/UTC*".

To enable the *systemTime* capability at a device level, create a DELEGATE capability in the CLES runtime with a similar process substituding PULL for DELEGATE. The SDK compiles all application interfaces into a plugin that can be loaded by the CLES runtime rather than compiled into an application.

Listing 4.1: CLES Interface

```
1        CLES_Service(deviceName, interface, port);
2        bool register<Verb>Function(name, function);
3        void run();
4        void stop();
5        CLES_Response makeCLESRequest(CLES_Request);
```

## 4.4 CLES Point-to-Point Service

The CLES Point-to-Point service reduces the number of additional data transmissions and translations (hops) from source to destination. This design facet also makes it unnecessary for all CLES users to support the full runtime. Henceforth, we refer to "Point-to-Point" as "P2P," not to be confused with "Peer-to-Peer.". The P2P CLES service closely mirrors the semantics of the Java "net.http" interface. Our goal is to flatten the learning curve for developers already familiar with the ubiquitous HTTP, so they can quickly transfer their knowledge to the domain of embedded systems. Like the "net.http" interface, the P2P service achieves minimal hops and overhead by integrating an in-line function call to the networking layer from the parent application.

The flexibility of CLES stems from its declarative syntax that follows the RESTful style and JSON-formatted data for transmission. What makes the language specification for CLES requests simple to implement is its straightforward verb noun syntax, with adjective modifiers

to specify additional options. For example, a request for coordinating vehicle platooning with another vehicle can be expressed in CLES as: "*BIND vehicle2:localization/si*" to subscribe to all localization data provided by vehicle 2 with the adjective *si* to specify the data format as the standard international system of units. Conversely, to alert surrounding cars to potential hazards, a vehicle that spots a downed tree could broadcast a CLES message to all vehicles as: "*POST all:hazard*" with this JSON package:

```
{"hazard": {
  "Type": "Blockage",
  "Lanes": "all",
  "Location": [
      {"Latitude": 123.45},
      {"Longitude": 67.89},
    ]
  }
  "Cause": "Tree",
  }
}
```

This CLES message request and JSON package response has one limitation: like in traditional wireless protocols, messages must follow a defined standard to be useful. For the previous example, there must be a defined package type "hazard" with required fields such as "type", "lanes", and "location". While this constraint also applies to traditional serialized packets, a plain-text, parsable JSON message allows for optional fields (e.g., "Cause") to be added both for higher-level functionality, such as fire department alerts or future systems upgrades. However, unlike in traditional rigid communication, these additional fields do not disrupt existing capabilities, as the message is not serialized into a predefined packet defined

by an Interface Control Document (ICD).

## 4.5 CLES Runtime and Task Distribution

The CLES runtime introduces novel asymmetric task registration and remote access to meet the unique execution constraints of its target domain. The CLES runtime is designed around the concept of "a distributed collection of threadpools." In this design, each device that processes externally-provided tasks must have a standalone CLES runtime service, similar to the service of the worker nodes in container orchestration architectures. One key difference between CLES and container orchestration solutions is that every machine with a CLES runtime serves both as a *leader* and *worker* node. In this way, each node can process tasks passed to it from an external source with the DELEGATE verb, as well as pass commands to other nodes. Internally, each runtime comprises a threadpool that processes all received commands. This can be extended to support task sharing and stealing, much like a modern threadpool. The established distributed synchronization properties of a threadpool ensure mutual exclusion, thus preventing all deadlocks, livelocks, and task duplication.

Another difference between CLES and container orchestration is how capabilities are added to a node. Instead of dynamically deploying containers, incurring high bandwidth costs, each CLES runtime locally registers plugins at startup. The asymmetric nature of device-specific plugins and leaderless coordination support the flexibility and timeliness requirements in highly heterogeneous embedded environments.

While the CLES runtime supports all of the previously mentioned verbs, it uniquely supports processing of the DELEGATE verb. DELEGATE represents the CLES equivalent of adding a task to a threadpool, while the remaining verbs represent common actions of point-to-point communication. A notable usage of the runtime outside of DELEGATE paradigm is creating

runtime plugins that expose PULL interfaces to data shared across all applications on a device, such as UTC time, processor load, or RAM usage.

## 4.6 CLES Implementation

Given OS and language limitations stemming from the *heterogeneous* requirement, we developed CLES in accordance with the C++17 standard, without third-party libraries. C++ remains an industry standard and the primary development language for both the defense and automotive industries.

To address the variety of network architectures and achieve interoperability, CLES supports native IP communication protocols User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) for its networking layer. IP is also universally accepted and nearly all modern networking protocols, i.e. WiFi, data link, Bluetooth, etc. support IP as a method of routing communication between devices. The CLES service wrapper abstracts this interface, which opens opportunity for future development to extend this interface to include memory mapped I/O and common data distribution platforms such as Google ProtoBuf, ActiveMQ, and DDS. Embedding the network interface into the CLES SDK interface caters to the desires of the embedded systems community because unlike solutions such as the the RQL mobile device runtime [6], and data brokers like ActiveMQ, P2P CLES avoids passing of information between third-party runtimes or brokers. This helps reduce latency, but also supports deterministic real-time scheduling as defined by the parent application because this interface is called in-line directly by the parent with no additional non-deterministic processing constraints or data transmissions.

The Software Development kit (SDK) for CLES consists of a supporting static C++ library for extending an application with point-to-point service, the CLES runtime, and all necessary

interfaces to create a plugin to extend the runtime capabilities. Plugins for the runtime follow the traditional C++ DLL interface, which is common across plugin architectures. A developer can create a plugin by using the provided CLES development SDK library, and exposing the required interfaces to the CLES runtime, which are as simple as defining an interface to the desired capability.

Currently, the automotive industry relies on real-time operating systems to host V2X applications, with the top competitors including QNX by blackberry, VxWorks by Wind River Systems, and the newer Real-Time Linux (RT-Linux) also by Wind River. This SDK has been compiled and run on both Windows and Linux based systems, which guarantees compatibility with QNX and RT-Linux, and is theoretically able to port to VxWorks and other Operating Systems with minimal modification to system calls for IP for network interface configuration and socket control.

# Chapter 5

# Evaluation

We evaluated CLES via micro-benchmarks, a representative application of Distributed Stochastic Gradient Descent, and design case studies. The benchmarks isolate the performance characteristics of relevant parameters; the application implementation of Distributed Stochastic Gradient Descent validates CLES usability and flexibility for distributed workloads; the case studies demonstrate the applicability of CLES in meeting the tight timing requirements of vehicle platooning and the flexibility requirements of task allocations with U.S. Military Joint All Domain Command and Control. Our evaluation is driven by the following questions: (1) Does CLES meet the timeliness requirements for vehicle platooning? (2) Is the plain-text packet structure of CLES compatible with the network protocols of distributed embedded systems? (3) How does the developer workload of CLES compare to that of traditional programming models? Given that CLES consists of both a Point-to-Point service and task allocation runtime, each was evaluated against different criteria.

## 5.1 P2P CLES Performance and Micro-Benchmarks

CLES has been designed to prioritize programmability while maintaining timeliness constraints. To capture the performance of the CLES P2P service, we evaluated the round trip time overhead, packet size, and implementation source lines of code. This process was also completed for the traditional method of constructing a serialized packet with an ICD

definition and implementing a minimal TCP client-server connection.

Both methods were tested with the task of having a client request a localization packet from a server analogous to basic V2V requirements. The CLES verb PULL was used to capture both the overhead of a server parsing a CLES message and constructing a JSON object with the response. PULL represents the worst case overhead because it exercises both a send and receive using all basic CLES functions that add overhead. Additionally, both the ICD and CLES implementation use TCP as their network protocol. Benchmarking was conducted both locally on a single machine, and across a WiFi network to isolate the CLES overhead in the total Round-Trip-Time (RTT). The localization packet contained a timestamp, latitude, longitude, forward velocity, and forward acceleration.

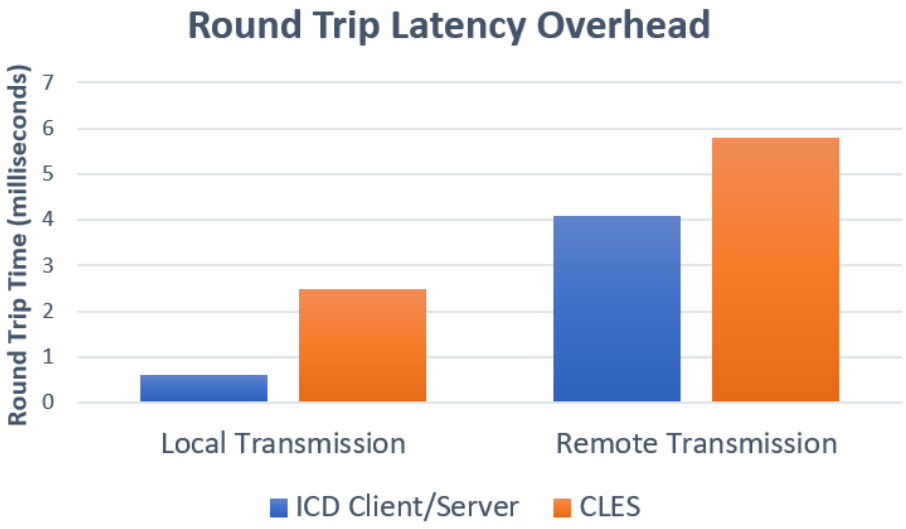


Figure 5.1: CLES Latency Micro-benchmark

Each category of analysis has an important takeaway. First, the total latency overhead for a CLES PULL versus a traditional ICD packet is around $1.5ms$. When considering the CLES verb paring BIND and UPDATE, continuous updates remove half of the round trip time, and as such, this additional overhead is within 10% of the total allowable $100ms$ latency for vehicle platooning [5]. Another factor to consider with total latency overhead is

scaling. The recursive descent parser used within the language definition for CLES, while theoretically achieving a maximum performance scaling of $O(n)$, is reduced to a constant time $O(1)$ performance because the scaling factor is based on the size of the CLES request itself, which is limited to a single verb and noun pair with additional adverbs being passed directly to a registered function. On the receiving end of the function, parsing a JSON object is bound by a complexity of $O(n)$ relative to the size of the message, which in most cases, like with localization, remains constant to constrain the parse to a realistic operational complexity of $O(1)$. Given that the request and response are handled directly in-line with the parent application, and both ends reduce to a given constant time complexity $O(1)$, this experimental result of $1.5ms$ additional overhead is directly transferable across all CLES interfaces with only minimal differences in the target behavior from processor frequency and JSON response size.

Second, the packet size overhead, while being larger by a factor of three (132 bytes for CLES), is still a small fraction of the 65,535 byte maximum allowable transmission size for TCP or UDP. The CLES JSON Response is also well within the single transmission frame size of 1500 bytes for 5G [7] which indicates that no additional overhead will be required to transmit the larger JSON packet. Finally, the implementation of CLES requires 30 lines of code, which represents 1/10th as many lines of code as the traditional ICD method with a TCP client and server. Equally as important as lines of code are the complexity and programmability. Implementing the CLES solution required no knowledge of the TCP/IP stack and sockets, thus significantly reducing the learning curve for extending networked capabilities within an application.

## 5.2 CLES Runtime Evaluation

The CLES Runtime is designed to provide flexibility and programmability for distributing processing tasks across multiple devices. To evaluate these criteria, the CLES Runtime was used to implement Distributed Stochastic Gradient Descent (D-SGD) to demonstrate that it can be effectively distributed with CLES, achieving satisfactory performance. Because Stochastic Gradient Descent is a fundamental mathematical principle of machine learning, this use case confirms that CLES can be successfully applied to implement distributed processing solutions in this and similar domains.

**Algorithm 1** SGD($\{c^1, \ldots, c^m\}, T, \eta, w_0$)

**for** $t = 1$ **to** $T$ **do**
  Draw $j \in \{1 \ldots m\}$ uniformly at random.
  $w_t \leftarrow w_{t-1} - \eta \partial_w c^j(w_{t-1})$.
**end for**
**return** $w_T$.

**Algorithm 2** ParallelSGD($\{c^1, \ldots c^m\}, T, \eta, w_0, k$)

**for all** $i \in \{1, \ldots k\}$ **parallel do**
  $v_i = \text{SGD}(\{c^1, \ldots c^m\}, T, \eta, w_0)$ on client
**end for**
Aggregate from all computers $v = \frac{1}{k} \sum_{i=1}^{k} v_i$ and **return** $v$

Figure 5.2: Parallel Stochastic Gradient Descent [26]

The Parallel Gradient Descent algorithm [26] was distributed using the Sandblaster Limited-Memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) model [9]. This model divides the data set into batches called data shards that are processed in parallel using a central management node to coordinate their distribution and subsequent consolidation.

Two machines, a workstation PC and a laptop, each hosted a CLES runtime with a plugin for single-threaded stochastic gradient descent on a supplied data shard. The data set was loaded on each machine to remove the additional overhead of data transfer, thus isolating

the performance impact of CLES. The management node divided up the data set into data shards, represented as parameters for data set access, and used CLES to DELEGATE the processing of those shards to the CLES runtimes.

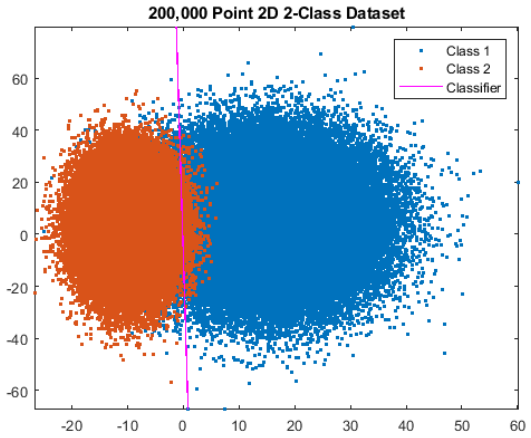


Figure 5.3: Data Set with Classifier

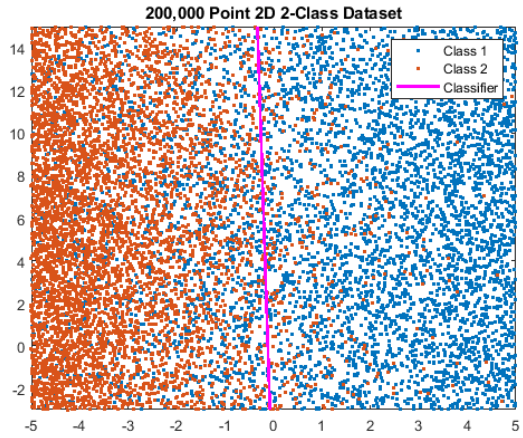


Figure 5.4: Zoomed Data Set With Classifier

The data set used to demonstrate the D-SGD was a 2-Dimensional, 2 class linear classification problem with 100,000 normally distributed data points per class (Figure 5.3). To evaluate the additional overhead of CLES, the dataset was classified both individually on the laptop and the PC using a single thread, and four threads. These results were then compared to equivalent tests using the CLES tuntime and management node to control the distribution of tasks using CLES DELEGATE.

The D-SGD results in Table 5.1 validate a design pattern common to Sandblaster L-BFGS and other data processing distribution methods of minimizing communication necessary to coordinate nodes. In the D-SGD workload, relatively few messages with a low total overhead are used to coordinate large processing workloads. The minimal cost of CLES is demonstrated as $10ms$ locally in the comparison of running four threads on the PC, and DELEGATING 4 data shards to the PC CLES runtime from a manager on the same machine using CLES. The overhead, less than 0.5% of the total processing time, is a minimal cost for

Table 5.1: D-SGD Performance Results

| **Local Multi-Threaded D-SGD** | | |
|---|---|---|
| Test | Average Execution Time (ms) | Mean-Squared Error |
| 1 Thread PC | 8150 | 0.0198995 |
| 1 Thread Laptop | 9360 | 0.0198995 |
| 4 Threads PC | 2180 | 0.0198977 |
| 4 Threads Laptop | 2500 | 0.0198977 |
| **CLES DELEGATE D-SGD** | | |
| 4 PC Threads, PC Request | 2190 | 0.0198977 |
| 4 PC Threads, Laptop Request | 2290 | 0.0198977 |
| 8 Threads Both Machines, Laptop Request | 2150 | 0.0198978 |

allowing distribution beyond a single machine. The difference between managing CLES D-SGD locally from the PC (2190$ms$) and remotely (2290$ms$) represents the additional network overhead of 100$ms$. This highlights the CLES performance impact as minimal compared to overall network performance.

The previously demonstrated minimal overhead of coordination relative to absolute data processing time emphasizes the design priority of flexibility and programmability for distributed workloads. The CLES implementation for extending SGD into a plugin and creating a management node to make the DELEGATE requests and combine responses requires less than 300 source lines of C++. This implementation could be similarly adapted to any distribution domain with similar manager-worker semantics.

### 5.2.1 Platooning CLES Solution

CLES fits the design requirements: it is quick to implement, compatible with real-time OSs, reliant on IP networking, reducing latency via a Point-to-Point service, while its RESTful DSL request and response structure supports quick upgradability, backwards compatibility,

and inter-manufacturer operability. Optionally, a C++ vehicle control module allows for directly including CLES to meet real-time deterministic timing constraints.

After establishing CLES as a viable middleware solution, the integration with CLES was designed. We selected the Point-to-Point service because dynamic task allocation is out of scope and the design drivers are latency, minimal data transmission hops, and deterministic performance. The minimal request and response structure is for each vehicle to request persistent updates of localization data from the vehicle directly in front of it, and the platoon leader [23]. The design requirement of persistent updates naturally fits with the CLES verb pairing BIND and UPDATE. A CLES BIND request for localization messages from all vehicles on the local network created by 802.11p or 5G covers all required functionality. Finally, to fulfill this BIND, each vehicle would post an UPDATE of their localization, which would be sent to all bound vehicles in the local network.

To develop the previously designed solution, the first step is to download the CLES SDK and link the P2P service into existing controls module software. In the vehicle control source code, first the CLES Service is constructed and initialized as referenced in the interface section. In this case, the only requirement is that the control software calls the service "Register_Update(String Access_Name)" with the parameter "Localization". This allows for the other vehicles to bind to the update using the keyword "Localization" and enables the control software to later call "*Update(String Access_Name, CLES_Response response)*" which pushes the new JSON Based CLES_Response object to all vehicles bound to the key "Localization".

This second interface would be the CLES BIND request. The bind request "*BIND all:localization*" would be passed to the CLES P2P service with an optional conditional variable to notify the host control program that an update has been received. In this case, that can be ignored because the control software will be designed to use all localization data from the CLES

Service at the beginning of each control frame. This is done by adding a function call to the control software that requests the *CLES_Response* object for each vehicle using the function "*CLES_Response Get_Update("Localization", "VehicleX")*".

Optionally, the "*CLES_Response* object can contain responses from all vehicles in the case of the key "all" being used. It is important to note that this function call is retrieving the most recent updates from inside the CLES Service memory, rather than the similar PULL verb, which when when used would send a message to each vehicle and request a one time data update, effectively doubling the maximum possible latency.

### 5.2.2 JDAC CLES Solution

The OS flexibility, integration with the OSI IP stack, and ease of development of CLES make it a suitable solution for implementing Joint All-Domain Command and Control (JDAC).

In this scenario, given the core design requirement of task allocation, such as requesting a processing task from the satellite, and a strike task from the strike assets, the CLES runtime was explored as a solution. First, the satellite exposes its ability to process FMV into *Tracks* by integrating the CLES runtime and creating a plugin for the DELEGATE capability *FMV_Processing*. This capability accepts required metadata to begin receiving FMV and returns the calculated *Track*. Second, the reconnaissance aircraft integrates the CLES runtime with its Operational Flight Program (OFP). The OFP of the aircraft upon collecting FMV uses the CLES runtime to DELEGATE remote processing to any asset with the *FMV_Processing* capability. The aircraft OFP after receiving the *CLES_Resonse* with a *Track*, leverages the CLES P2P service and the POST verb to send the track to all other bound parties. The mission center with a CLES P2P service, BINDS to the *Tracks* from the reconnaissance aircraft. The data is then presented to humans in the loop to make critical

strike decisions. After a strike decision, the command station then sends a DELEGATE strike task to an available asset that registered a compatible strike capability such as the fighter aircraft.

# Chapter 6

# Future Work

The research herein comprises the problem space of embedded system communication and the design and development of the CLES language and runtime. This novel solution built on the ubiquitous RESTful architecture solves design criteria for heterogeneous device coordination and distributed workloads presented in section 3. The field of distributed embedded computing is wider than Vehicle Platooning and JDAC. CLES presents an open opportunity to be extended with more advanced capabilities to solve a broader variety of challenges.

## 6.1 TLS/SSL Encryption and JWT

CLES with its semantic similarities to HTTP, the dominant RESTful language that drives the internet, can be extended with the same techniques. HTTPS supports secure communications through TLS/SSL (Transport Layer Security/Secure Socket Layer). TLS/SSL provides secure communication by having a host device send its client a certificate issued by a certificate authority to identify itself and a public key for the client to encrypt data sent back to the host [20]. The client validates the certificate with a registered certificate authority to prove authenticity of the host, then sends data encrypted with the public key that can be decrypted only by the host.

CLES can incorporate TLS/SSL encryption though minimal extension of the language and built in support in the SDK and runtime. The language grammar can be extended with

adverbs such as supported TLS version, the runtime and P2P service would be supplied a certificate and certificate authority on startup, and the CLES middleware would complete the handshake, certificate validation, and encrypt messages on send. Using a standard TLS interface definition within CLES also allows for applications created independently of the CLES SDK to either implement their own TLS handling or communicate unsecurely with a local CLES runtime that handles TLS/SSL with an external device.

In addition to TLS/SSL for creating an encrypted communication channel, the stateless nature of CLES from it RESTful design is capable of passing JWT (JSON Web Token) through its requests as adjectives/parameters and in its responses through the JSON packet. JWTs are traditionally used to validate user's credentials in a logged in session by supplying a JWT signature in each packet after a log in has been established [17].

The use cases of encrypted communication at the application and transport layers are useful across military, banking, autonomous vehicles, and more. Notably for CLES, a primary concern of V2V and platooning is security. The concern is that a bad actor could falsify vehicle information and cause an accident. While TLS/SSL is not applicable to V2V networks that are purely point-to-point with protocols like 802.11p, other protocols such as 5G require a connection to a cellular tower for full functionality which would guarantee access to a dedicated V2V certificate authority. In cases of military communication, it is common practice for all devices in a system of systems to be manually loaded with an encryption key on startup, or be connected to a military certificate authority.

## 6.2 Further Testing

The initial testing completed in this research effort contains representative micro-benchmarks to validate the core use cases, but there are many opportunities to expand the scope of testing

and validation.

### 6.2.1 Computing Clusters

CLES is designed for scalability. A logical step to extend the evaluation of CLES is to distribute the CLES runtime across a computing cluster designed to support large-scale distributed computations. This could be supported through the Virginia Tech rlogin multi-machine system, or through a request to the Virginia Tech Advanced Research Computing (ARC) organization.

### 6.2.2 Advanced Machine Learning

Stochastic Gradient Descent is a fundamental principal for machine learning, but validation of CLES with machine learning would ideally involve implementing a more complex solution such as "Distributed Deep Neural Network Training on Edge Devices" [3]. The complexity in extending CLES for this problem is the lack of pre-existing library support. Machine Learning tools such as TensorFlow and PyTorch contain the algorithms necessary, but both implement their own custom communication and distribution layer. This presents a non-trivial implementation challenge of either recreating the neural network training algorithms from scratch, or modifying the open source software of TensorFlow or PyTorch to inlude CLES. This tradeoff helped inform the decision of selecting SGD in this research effort with similar justification as Benditkis et al. of representative performance to neural network training [3].

### 6.2.3 Representative Hardware and Networks

The target for CLES is embedded systems. A useful study would be the performance characteristics and portability onto various embedded platforms and networks.

While CLES is designed to be easily portable to VxWorks, QNX, and other RTOS and embedded Operating Systems, going through the exercise of porting CLES would validate its flexibility and the workload required for porting. Deploying CLES to a RTOS would also allow for straightforward benchmarking to prove the deterministic processing claims made in this research effort. This would also be a good opportunity to study the processor and memory load of CLES on common embedded devices such as the Xilinx System-on-Chip (SoC)/System-On-Module (SoM), Raspberry Pi, Intel Edison, etc. In addition to validating CLES for embedded devices, CLES could be tested and validated on networks such as 802.11p, 802.11ac/ax WiFi, 5G, and Bluetooth.

### 6.2.4 System of Systems Development or Simulation

An opportunity for cross department research is the development or simulation of a full CLES system of systems to demonstrate a problem domain explored in this research effort. This could be in cooperation with the Aerospace or Mechanical Engineering departments to develop a group of coordinated air or ground robots to function either as a coordinated swarm like with vehicle platooning, or a heterogeneous sensing network to represent JDAC. Large scale distributed simulations could also be developed jointly with the Electrical or Computer Engineering departments with an emphasis on simulated wireless networks and hardware limitations.

## 6.3 Broader Problem Domain

This research effort has primarily focused its attention on V2V, JDAC, and distributed machine learning, but the flexibility of CLES lends itself to other domains. The field of embedded systems also includes infrastructure and Industrial Control Systems (ICS). Both city infrastructure and ICS domains such as power plants and manufacturing have strict computing power, timing, security, safety, and flexibility constraints similar to V2V and JDAC.

One of the most prevalent fields of embedded computing is cell phones. RESTful architecture has been explored for celluar devices through native HTTP support, and the RQL runtime [6], but CLES presents a viable solution to support multi-network requirements such as Bluetooth and cellular simultaneously. This multi-network support also naturally positions CLES for the prominent domains of the Internet of Things (IoT), and Smart Homes.

As is the case with all distributed systems, the issue of handling partial failure would become prominent when applying CLES to build applications that coordinate the execution of heterogeneous embedded devices in large-scale realistic environments. Such environments can be marked by high volatility and unpredictability of their network connectivity and device operation. To provide effective fault tolerance in the programmer-friendly fashion enabled by CLES, fault tolerance can be exposed as reusable components [18]. In the context of RESTful architectures, it has been proposed to specify various fault handling strategies declaratively and translating them into platform-specific code modules [12, 13]. This strategy can be explored as a future work direction for enhancing CLES with declarative fault tolerance.

# Chapter 7

# Conclusions

Modern embedded systems—autonomous vehicle-to-vehicle communication, smart cities, and military Joint All-Domain Operations—feature increasingly heterogeneous distributed components. Existing embedded system solutions for communication and networking are inflexible, tightly coupled to wireless protocols, and expensive to develop to satisfy the requirements. On the other extreme, modern software solutions for distributing data, allocating dynamic tasks, and deploying applications cannot satisfy embedded system requirements because of centralized management, operating system constraints, and heavyweight middleware.

This Thesis has presented a Representational State Transfer (REST) architecture, designed and implemented to uniquely complement the constraints of embedded systems development, such as language, operating system, latency, and networking protocols. Our solution features a domain-specific language, called the Communication Language for Embedded Systems (CLES), that supports both traditional point-to-point data communication and allocation of decentralized distributed tasks. We demonstrated how CLES can increase programmability and flexibility of developing communication in embedded systems with marginal performance impacts through representative micro-benchmarks, a distributed stochastic gradient descent use case, and application case studies.

# Bibliography


[1] DARPA tiles together a vision of mosaic warfare. URL https://www.darpa.mil/work-with-us/darpa-tiles-together-a-vision-of-mosiac-warfare.

[2] Kubernetes - concepts - overview. URL https://kubernetes.io/docs/concepts/overview/.

[3] Daniel Benditkis, Aviv Keren, Liron Mor-Yosef, Tomer Avidor, Neta Shoham, and Nadav Tal-Israel. Distributed deep neural network training on edge devices. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, SEC '19, pages 304–306. Association for Computing Machinery. ISBN 978-1-4503-6733-2. doi: 10.1145/3318216.3363324. URL http://doi.org/10.1145/3318216.3363324.

[4] M. Boban, A. Kousaridas, K. Manolakis, J. Eichinger, and W. Xu. Connected roads of the future: Use cases, requirements, and design considerations for vehicle-to-everything communications. *IEEE Vehicular Technology Magazine*, 13(3):110–123, 2018. doi: 10.1109/MVT.2017.2777259.

[5] C. Campolo, A. Molinaro, G. Araniti, and A. O. Berthet. Better platooning control toward autonomous driving : An lte device-to-device communications strategy that meets ultralow latency requirements. *IEEE Vehicular Technology Magazine*, 12(1):30–38, 2017. doi: 10.1109/MVT.2016.2632418.

[6] S. Chadha. Supporting heterogeneous device development and communication. Master's thesis, Virginia Tech, Blacksburg, Virginia, USA, 2016.

[7] Luca Cominardi, Luis M. Contreras, Carlos J. Bcrnardos, and Ignacio Berberana.

Understanding QoS applicability in 5g transport networks. In *2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 1–5. IEEE. ISBN 978-1-5386-4729-5. doi: 10.1109/BMSB.2018.8436847. URL https://ieeexplore.ieee.org/document/8436847/.

[8] Congressional Research Service. Joint all-domain command and control (jadc2), November 2020. URL https://fas.org/sgp/crs/natsec/IF11493.pdf.

[9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf.

[10] Android Developers. Android interface definition language (aidl). URL https://developer.android.com/guide/components/aidl.

[11] Docker. What is a container? | app containerization | docker. URL https://www.docker.com/resources/what-container.

[12] John Edstrom and Eli Tilevich. Reusable and extensible fault tolerance for restful applications. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 737–744, 2012. doi: 10.1109/TrustCom.2012.244.

[13] John Edstrom and Eli Tilevich. Improving the survivability of restful web applications via declarative fault tolerance. *Concurr. Comput.: Pract. Exper.*, 27(12):3108–3125, August 2015. ISSN 1532-0626. doi: 10.1002/cpe.3197. URL https://doi.org/10.1002/cpe.3197.

[14] Roy T Fielding. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Irvine, 2000.

[15] U.S. Air Force. U-2 federal lab achieves flight with kubernetes. URL https://www.af.mil/News/Article-Display/Article/2375297/u-2-federal-lab-achieves-flight-with-kubernetes/.

[16] G2. Kubernetes alternatives & competitors. URL https://www.g2.com/products/kubernetes/competitors/alternatives.

[17] Michael Jones, John Bradley, and Nat Sakimura. JSON Web Token (JWT). RFC 7519, May 2015. URL https://rfc-editor.org/rfc/rfc7519.txt.

[18] Young-Woo Kwon, Eli Tilevich, and Taweesup Apiwattanapong. Dr-osgi: Hardening distributed components with network volatility resiliency. In Jean M. Bacon and Brian F. Cooper, editors, *Middleware 2009*, pages 373–392, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[19] I. S. H. Martínez, I. P. O. J. Salcedo, and I. B. S. R. Daza. Iot application of wsn on 5g infrastructure. In *2017 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–6, 2017. doi: 10.1109/ISNCC.2017.8071989.

[20] Kerry A McKay and David A Cooper. Guidelines for the selection, configuration, and use of transport layer security (TLS) implementations. URL https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-52r2.pdf.

[21] A. Talebpour S. Arefizadeh and I. Zelenko. Platooning in the presence of a speed drop: A generalized control model, Sep 2017. URL http://arxiv.org/abs/1709.10083.

[22] O. Sheikh S. Dikaleh and C. Felix. Introduction to kubernetes. *Proceedings of the*

*27th Annual International Conference on Computer Science and Software Engineering*, November 2017.

[23] D. Swaroop and J. K. Hedrick. Constant spacing strategies for platooning in automated highway systems. *Journal of Dynamic Systems, Measurement, and Control*, vol. 121 (no. 3):p. 462, 1999.

[24] X. F. Xie Fei Y. L. Yang Liangyi, S. D. Sun Dihua and Z. J. Zhu Jian. Study of autonomous platoon vehicle longitudinal modeling. *IET International Conference on Intelligent and Connected Vehicles (ICV 2016)*, 2016.

[25] A. Byalik Z. Song, S. Chadha and E. Tilevich. Programming support for sharing resources across heterogeneous mobile devices. *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems - MOBILESoft '18*, pages 105–116, 2018.

[26] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. Parallelized stochastic gradient descent. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010. URL https://proceedings.neurips.cc/paper/2010/file/abea47ba24142ed16b7d8fbf2c740e0d-Paper.pdf.