CLES: A Universal Wrench for Embedded Systems Communication and Coordination

Jason Davis(✉) and Eli Tilevich

Software Innovations Lab, Virginia Tech, Blacksburg, USA
{jdvavis7,tilevich}@vt.edu

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, this paper 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 allocation of decentralized distributed tasks. To meet the traditional constraints of embedded execution, CLES's novel runtime allocates decentralized distributed tasks across a heterogeneous network of embedded devices, overcoming limitations of centralized management and limited operating system integration. We evaluated CLES with performance micro-benchmarks, implementation of distributed stochastic gradient descent, and by applying it to design versatile stateless services for vehicle-to-vehicle communication and military Joint All-Domain Command and Control, thus meeting the data distribution needs of realistic cyber-physical embedded systems.

Keywords: Embedded networking · Low-latency networking · Vehicle to vehicle communication · RESTful architecture

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 is provided via a custom, highly optimized specialized protocol for each pair of connected systems. For safety-critical cyber-physical systems, this approach is deemed as required to meet the timeliness requirements. Unfortunately, the resulting low-level, platform-specific code is hard to write, extend, and maintain.

S. Deng et al. (Eds.): MobiCASE 2021, LNICST 434, pp. 54–68, 2022.
https://doi.org/10.1007/978-3-030-99203-3_4

Web development coordinates heterogeneous distributed components via the RESTful architectural style [6] with its ubiquitous HTTP communication, while high-powered computing manages distributed workloads with container orchestration platforms, such as Docker Swarm and Kubernetes. Unfortunately, developers cannot apply these proven solutions to distributed embedded systems, with their proprietary architectures, operating system limitations, centralized management requirements, and non-deterministic timing constraints.

To support the creation of heterogeneous networked solutions easily integrated into embedded systems, this paper presents CLES[1], a Domain Specific Language (DSL) for implementing simple, stateless, communication protocols in the RESTful architectural style [6]. With its simple syntax and platform APIs, 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 paper 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 access, in order to meet the unique execution constraints of its target domain.
– *An empirical evaluation of CLES with representative performance microbenchmarks, a reference distributed stochastic gradient descent implementation, and realistic case studies.*

2 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

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

explored as a solution for both the U.S. Air Force [7] 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.

2.1 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 controlling each vehicle's steering and speed. With its constituent vehicles communicating and coordinating with each other, a platoon increases the safety and energy efficiency of highway travel [9].

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 [1]. Additionally, Swaroop [10], Arefizadeh [9], and Liangyi [11] introduced algorithms that defined the minimum data transmission required to achieve stable platoons. One question that remains unexplored is what kind of software architecture and abstractions are needed to realize practical platooning solutions (Fig. 1).

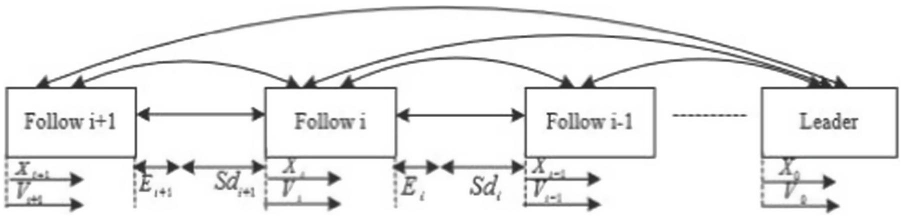

Fig. 1. Platooning networking structure [10]

As baseline networking protocols, our implementation references 802.11p and 5G, which both support OSI IP [8] and uses the defined minimum dataset required for stable platooning: lead vehicle localization data (UTC Time, latitude, longitude, Velocity, and Acceleration), and immediately ahead vehicle localization [10].

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 for transmission 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 is decentralized, DDS requires a significant development effort to properly support vehicle to vehicle communication, as data middleware without high-level language bindings. Lastly, common real-time OS' VxWorks and QNX provides no support for Kubernetes, removing it from consideration.

2.2 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 [4]. 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.

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 unmanned 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 capable of launching a ground strike. The satellite hosts a powerful processor, machine learning algorithms, and other sources of data for fusion.

For the 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. Air Force.

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 strike 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.

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 aircraft.

Existing Approach. The CONOPS proposed above is similar to the design drivers cited for the U-2 Kubernetes integration [7]. Although Kubernetes is a powerful tool for distributing tasks via container orchestration, it is limited to certain OS. Only some of the highly heterogeneous participating assets in a Joint All-Domain Operation can host Kubernetes.

3 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 communication and coordination in embedded systems.

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

3.1 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/director 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 function with any node missing.

3.2 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 1).

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 Sect. 3.4. This service has been designed and explored to solve the challenges present in the *Platooning* use case.

The design 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 service, with a single runtime for each embedded device that supports remote task submissions. 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.

Table 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

3.3 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

$$\begin{aligned}
\langle \text{CLES Command} \rangle &\models \langle \text{verb} \rangle\ \langle \text{noun} \rangle && (1)\\
\langle \text{verb} \rangle &\models \textit{pull} \mid \textit{push} \mid \textit{delegate} \mid \textit{bind} \mid \textit{update} && (2)\\
\langle \text{noun} \rangle &\models \langle \text{remote target} \rangle : \langle \text{task name} \rangle / \langle \text{task parameters} \rangle && (3)\\
\langle \text{remote target} \rangle &\models \langle \text{generic string} \rangle && (4)\\
\langle \text{task name} \rangle &\models \langle \text{generic string} \rangle && (5)\\
\langle \text{task parameters} \rangle &\models \langle \text{parameter} \rangle \mid \langle \text{task parameters} \rangle \langle \text{parameter} \rangle && (6)\\
\langle \text{parameter} \rangle &\models \langle \text{generic string} \rangle && (7)\\
\langle \text{generic string} \rangle &\models [a - zA - Z0 - 9]+ && (8)
\end{aligned}$$

Fig. 2. CLES Backus-Naur Form (BNF) grammar

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 (Fig. 2), 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 1.1. CLES Interface

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

3.4 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," which is 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.

3.5 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.

3.6 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 UDP and 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 RQL mobile device runtime [12], 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.

4 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.

4.1 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 (Fig. 3).

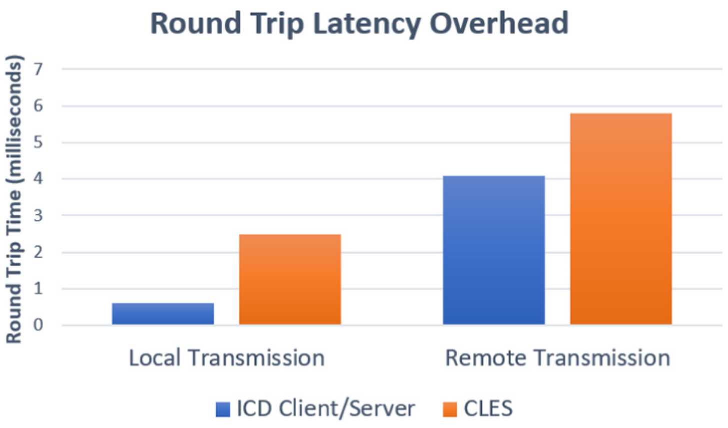

Fig. 3. CLES Latency Micro-benchmark

Each category of analysis has an important takeaway. First, the total latency overhead for a CLES PULL versus a traditional ICD packet is around 1.5 ms. 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 100 ms latency for vehicle platooning [2]. 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 1.5 ms 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 [3] 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.

4.2 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 (Fig. 4).

The Parallel Gradient Descent algorithm [13] was distributed using the Sandblaster Limited-Memory Broyden-Fletchger-Goldfarb-Shanno (L-BFGS) model [5]. 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.

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

```
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 $v$
```

Fig. 4. Parallel Stochastic Gradient Descent [13]

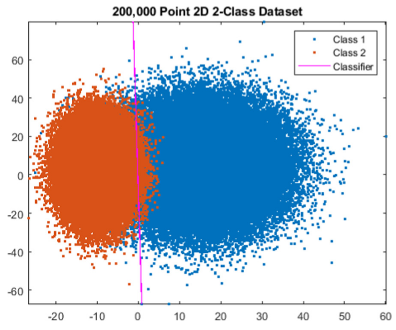

Fig. 5. Data set with classifier

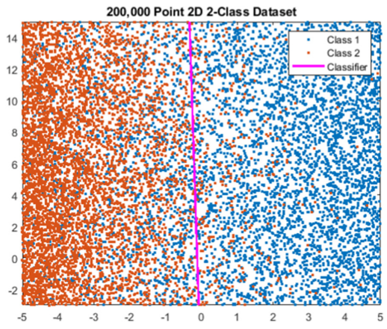

Fig. 6. Zoomed data set with classifier

class (Fig. 5). 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 (Fig. 6).

The D-SGD results in Table 2 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 10 ms 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 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

Table 2. 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

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.

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. As discussed above, we selected the Point-to-Point service, as dynamic task allocation is out of scope, while minimal hops with deterministic behavior was desired. 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 [10]. 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.

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.

5 Conclusion

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 paper 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.

Acknowledgements. The authors thank the anonymous reviewers, whose insightful comments helped improve this paper. NSF supported this research through the grant #1717065.

References

1. Boban, M., Kousaridas, A., Manolakis, K., Eichinger, J., Xu, W.: Connected roads of the future: use cases, requirements, and design considerations for vehicle-to-everything communications. IEEE Veh. Technol. Mag. **13**(3), 110–123 (2018). https://doi.org/10.1109/MVT.2017.2777259
2. Campolo, C., Molinaro, A., Araniti, G., Berthet, A.O.: Better platooning control toward autonomous driving: an LTE device-to-device communications strategy that meets ultralow latency requirements. IEEE Veh. Technol. Mag. **12**(1), 30–38 (2017). https://doi.org/10.1109/MVT.2016.2632418
3. Cominardi, L., Contreras, L.M., Bcrnardos, C.J., Berberana, I.: Understanding QoS applicability in 5G transport networks. In: 2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB), pp. 1–5. IEEE (2018). https://doi.org/10.1109/BMSB.2018.8436847. https://ieeexplore.ieee.org/document/8436847/
4. Congressional Research Service: Joint all-domain command and control (jadc2) (2020). https://fas.org/sgp/crs/natsec/IF11493.pdf
5. Dean, J., et al.: Large scale distributed deep networks. In: Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems, vol. 25. Curran Associates, Inc. (2012). https://proceedings.neurips.cc/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf
6. Fielding, R.T.: Architectural styles and the design of network-based software architectures, vol. 7. University of California, Irvine, Irvine (2000)
7. Force, U.A.: U-2 federal lab achieves flight with kubernetes. https://www.af.mil/News/Article-Display/Article/2375297/u-2-federal-lab-achieves-flight-with-kubernetes/
8. Martínez, I.S.H., Salcedo, I.P.O.J., Daza, I.B.S.R.: IoT application of WSN on 5G infrastructure. In: 2017 International Symposium on Networks, Computers and Communications (ISNCC), pp. 1–6 (2017). https://doi.org/10.1109/ISNCC.2017.8071989
9. S. Arefizadeh, A.T., Zelenko, I.: Platooning in the presence of a speed drop: a generalized control model (2017). http://arxiv.org/abs/1709.10083
10. Swaroop, D., Hedrick, J.K.: Constant spacing strategies for platooning in automated highway systems. J. Dyn. Syst. Measur. Control **121**(3), 462 (1999)
11. Yang, L., Dihua, S., Fei, X., Jian, Z.: Study of autonomous platoon vehicle longitudinal modeling. In: IET International Conference on Intelligent and Connected Vehicles (ICV 2016) (2016)
12. Song, Z., Chadha, S., Byalik, A., Tilevich, E.: Programming support for sharing resources across heterogeneous mobile devices. In: Proceedings of the 5th International Conference on Mobile Software Engineering and Systems - MOBILESoft '18, pp. 105–116 (2018)
13. Zinkevich, M., Weimer, M., Li, L., Smola, A.: Parallelized stochastic gradient descent. In: Lafferty, J., Williams, C., Shawe-Taylor, J., Zemel, R., Culotta, A. (eds.) Advances in Neural Information Processing Systems, vol. 23. Curran Associates, Inc. (2010). https://proceedings.neurips.cc/paper/2010/file/abea47ba24142ed16b7d8fbf2c740e0d-Paper.pdf