# Infinite Prognostics & Diagnostics: A System Architecture to Support P/D Algorithms Before They Even Exist

**James Branigan**
*Band XI International, LLC*
Creedmoor, NC

**John Cunningham**
*Band XI International, LLC*
West Hartford, CT

**Patrick Dempsey**
**Brett Hackleman**
**Paul VanderLei**
*Band XI International, LLC*
Bracey, VA, Scottsdale, AZ, and Grand Rapids, MI

## ABSTRACT

*A combination of real world experience and new research initiatives will open up the universe of prognostic and diagnostic algorithms that can be created in the future. This presents the challenge of creating a system architecture that enables effective support of an infinite set of future algorithms even before they have been conceived, designed, implemented, tested, and approved for use. The Arbor architecture enables five critical elements to meet this challenge: (1) clean integration between legacy and new software, (2) remote, over the air provisioning of algorithms, (3) flexible data structures capable of evolving, (4) control points for the algorithm to report findings to in-vehicle occupants, and (4) a data collection strategy for failure incident reporting. Many algorithms are impossible to develop until we collect real world performance and failure information from on the vehicle. The Arbor system collects this information and feeds it off-board for analysis. Researchers analyze the data and develop diagnostic or prognostic algorithms that can then be deployed to a single vehicle experiencing odd behaviors or to an entire fleet, preemptively. A prognostic algorithm written, or modified, as an Arbor application can define its own outputs, which are then visible to the vehicle operator. These same outputs can be broadcast to a service technician with a diagnostic scan tool or to a remote operational command site, contingent on available communications links. Effective deployment of prognostic algorithms enables costly failures to be predicted ahead of time, thereby improving safety, reducing costs, and minimizing down time for equipment in order to effect more efficient fleet operations.*

## INTRODUCTION

The lifecycle of a vehicle model spans many years. Actual field deployments uncover usage patterns that stress the vehicle and its systems in new ways that result in failures. Such failures may occur in unforeseen patterns or even in carefully designed and rigorously tested subsystems or individual components. Ideally, design engineers would benefit from having an agent resident on the vehicle to observe vehicle behaviors, collect observations, and transmit data back to them for analysis. Unfortunately, they cannot know *a priori* what data, or with what frequency, needs to be collected so that this capability can be built into the base system. With access to the right data, engineers could analyze the actual field performance and develop effective diagnostic and prognostic algorithms that could aid personnel in at least diagnosing problems, if not providing them advance warning of an impending failure. The subsequent challenge revolves around deploying these algorithms as vehicle agents. Each algorithm should be a deployable unit to the vehicle. Each algorithm should be capable of providing useful information to the occupants of the vehicle about failures, encountered or impending, as well as collecting specialized relevant data and transmitting it back to the engineers for further analysis or record keeping.

Supporting this scenario requires an open, component-based system architecture that facilitates integration of heterogeneous components and evolution of underlying data models. Unfortunately, nearly all embedded software is developed as monolithic applications over very long periods

that include extensive and punishing system integration testing. Their complexity alone often precludes modification once they have been completed and deployed. When updates and upgrades are available, these must pass through the same rigorous testing procedures and then flow through to an arduous manual installation procedure. This process can be time consuming, costly, and requires manual intervention.

This paper introduces the Arbor system architecture for in-vehicle computing to address the challenges of creating an open component-based in-vehicle computing platform that can grow stronger in response to the challenges encountered over time. Arbor offers a strong root system that integrates closely with the vehicle bus and on board sensors, with the ability to reach deeper into the vehicle and extend more broadly to new sensors. Additionally, the component-based model facilitates the growth of new branches to the system that can extend the available functionality to serve new missions, accommodate new subsystems, and more richly support existing capabilities.

Arbor enables a class of applications, running on remote devices that sense and interact with their environment. These applications minimally consist of a sensing/controlling portion, a logic portion, and a notification portion. The sensing/controlling may interact with a multitude of hardware sensors, including but not limited to Global Positioning System devices (GPS), SAE J1939 devices (CAN), Radio Frequency Identification (RFID) devices, Programmable Logic Controllers (PLC), cameras, and radar devices. The logic portion may run on the embedded computing device, a handheld device, or remotely in a server environment. The notification portion may utilize a screen to notify an operator in the case of an in-vehicle system. Alternatively, in the event of a remote sensor alarm, the notification portion may send a high priority message to headquarters or broadcast a peer-to-peer notification over a deployed mesh network. There are many notification cases along the spectrum that the Arbor system can support.

This paper focuses specifically on Arbor's capability to accommodate the needs of the prognostics and diagnostics community. Arbor has already been fielded for commercial equipment used in mining and construction, such as salt harvesters and foundation drills. These fielded systems monitor and direct a variety of sensors and actuators connected to the J1939 CAN bus within the vehicle and also collect, present, store, and forward diagnostic messages for the vehicle occupant and engineers.

## THE CHALLENGE

In a nutshell, the problem we are address revolves around how to create an open platform for developing and deploying applications that:

- Acquire sensor data,
- Analyze the data,
- Turn the data into useful information, and
- Relay that information to someone to use.

The fact that these systems will be deployed to in-vehicle computing platforms that operate far from typical data centers or software engineering facilities further complicates the problem. These applications need to be installed, patched and managed remotely and in quantity. General connectivity may be intermittent, at best, and possibly quite low bandwidth even when available. We assume that periodically, the vehicles will be in range of base motor pools where local connectivity would be stable and quite fast. Additional constraints arise, because in-vehicle hardware platforms typically consist of slower and less powerful cores than standard desktop PCs.

Furthermore, in-vehicle applications, specifically algorithms, will be developed and deployed over the full lifetime of a fielded vehicle model, by a variety of vendors using a range of programming languages. In order to achieve the goal of building such an open platform, it is critical to define both a data model and a system architecture that supports the development of an open-ended, modular, dynamically updatable, multi-programming language capable application platform. The Arbor system architecture and data model describes a reference implementation that has been refined and executed over time to meet these objectives.

## DOD BROAD SYSTEM OBJECTIVES

Over the last decade, the United States military has made a dedicated effort toward adopting open, standards-based and commercial off the shelf (COTS) computing software and hardware. Programs such as the Department of Defense (DoD) Open Systems Joint Task Force (OSJTF) [1] and the Navy's Open Architecture Computing Environment (OACE) [2] have shown how using open standards-based solutions results in reduced cost, faster development, and greater flexibility.

For example, the open source Linux operating system has gained tremendous traction in the military due to its level of POSIX conformance, widespread availability, large developer and user base, and resulting lower development and maintenance costs. Simply put, the cost of technology change in open standards and open source architectures is significantly less than it is with proprietary architectures.

Beyond the platform level (hardware, device drivers, and operating systems), open standards are developing for tooling platforms (Eclipse Foundation) [3] and application architectures and deployment (Open Services Gateway Initiative Alliance) [4].

Prior interactions with the US Army Research, Development, and Engineering Command (RDECOM) have drawn out the following broad objectives in developing future vehicle based systems:

- Open System Architecture
- Scalable, Integrated, and Modular
- Reliable, Available, and Maintainable
- Long Term Evolution Capability
- Reduced Procurement and Maintenance Costs
- Security Capabilities
- Training and Knowledge Transfer

## LEGACY INTEGRATION

Proposing any new system architecture inevitably faces resistance to change. Such resistance is often based in very real, economic concerns regarding existing legacy systems. Large investments have been made in the design, development, testing, deployment, improvement, and maintenance of the existing software and systems. No responsible proposal would advocate a discontinuous jump to any new system architecture without providing a means to integrate the legacy base. Arbor's design explicitly held legacy integration as a primary, and critical, requirement.

Arbor provides an open, standards compliant mechanism for supplying existing applications with a clean and simple mechanism for pulling low-level data from devices. Additionally, Arbor uses a similar mechanism enabling legacy applications to post information for recording or controlling devices. A more detailed discussion of these mechanisms follows below through the example described.

## PROVISIONING

Any well-designed, open, modular system architecture should include a story for providing updates and upgrades to the base system. Here, we define an *update* as any correction or improvement to existing functionality and *upgrade* to mean the addition of new functionality. We refer to the process, procedures, and technologies employed to deliver updates and upgrades as *provisioning*. We name a given update or upgrade packaged for provisioning an *installable unit*. The contents of an installable unit may consist of code and/or data. The code may be in any programming language and may be in source form (shell scripts) or compiled (shared object libraries or executables). The data may be a resource (language fonts, images, etc.) or configuration files.

The Arbor system architecture incorporates provisioning technology developed under the auspices of the Eclipse Foundation and widely used to manage and deploy installations of the Eclipse IDE itself, as well as runtime applications built using Eclipse runtime technologies. This provisioning technology is codenamed *P2*. Band XI has played a central role in driving P2 to address the unique challenges of resource constrained embedded platforms. Although Eclipse built P2 using Java, we have helped drive the Eclipse team to separate provisioning concerns in such a way as to allow a non-Java, native provisioning agent to reside on the embedded platform. Furthermore, P2 can provision installable units that contain arbitrary resources and code. This allows us to provision algorithms written in C, MATLAB, or any other programming language.

The Arbor system can safely provision new algorithms without worrying about impacting already installed algorithms. It can also patch the sensing portion of the application without worrying about destabilizing the integration with the algorithm. This architecture is open, scalable, modular, reliable, maintainable, easy to learn and use, and provides a powerful model for long-range reuse and evolution. As Figure 1 illustrates, what you are really after is unparalleled operational *quality* in the systems you deploy. We achieve this quality by combining reliability, availability, and maintainability.

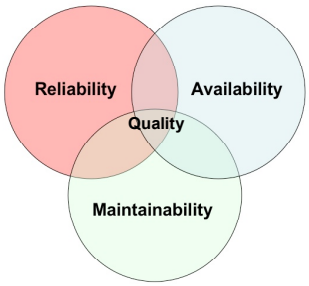


**Figure 1: Operational Quality = Reliability ∩ Availability ∩ Maintainability**

By leveraging the capabilities of P2, Arbor allows the teams responsible for vehicle operations to maintain and enhance the products and services in the field – *24 x7 x 365*. The software in products, and the services around them, can now be updated dynamically via secure networks, to fix problems, or add new functionality that adds value to the soldier's devices or to make services available on a range of different-brand devices deployed across the service. These innovative capabilities open completely new opportunities for lower-cost maintenance and repair, or the availability to add valuable new features without taking units out of service. Expensive recalls and high development costs can be reduced to meet increasing mission demands.

## ARBOR SYSTEM ARCHITECTURE

We've already mentioned that legacy application integration and application provisioning significantly drove the design of Arbor. It's important to also understand the general system engineering principles that drove the development of Arbor.

The principles of quality for computer architecture are well described by Dr. Fred P. Brooks and Dr. Gerrit A. Blaauw in their book Computer Architecture: Concepts and Evolution. [5] We appeal to these principles when faced with design decisions. Weighing a design decision against each of the principles helps to guide us to the right decision for the architecture by enhancing its quality, extending its usefulness, and lowering its long run cost. Brooks and Blaauw state that high quality architectures should have the following properties.

- Consistent
- Proper
- Orthogonal
- General
- Parsimonious
- Transparent
- Open-ended
- Complete

There are five high-level constraints that we must meet with the Arbor system architecture and data model.

- Must perform on embedded platforms
- Must be capable of remotely provisioning new features and bug fixes
- Must be programming language agnostic
- Must have vendor independent interfaces
- Must support existing data formats

We have learned a great deal from our past projects building embedded systems. Filtering that experience through the guiding principles listed above has lead us to develop a set of more concrete, derived principles.

- Data formats must be open
- Don't invent unless necessary
- Simplicity is the answer for complexity, not more complexity
- Programming language to programming language interfaces should be avoided
- Avoid using push style notification, because pull notification scales, while push notification does not
- Do not build a *walled garden*, a closed set or exclusive set of services that create a monopoly

The Arbor Data Model is conceptually based on the World Wide Web Consortium's (W3C) Resource Description Framework (RDF). [6,7] The RDF is an incredibly simple and flexible resource model and the basis of the Semantic Web [8]. The most well known form of RDF is the RDF-XML format, however there are other published representations of the RDF data model. The data model vocabulary is described in the Web Ontology Language (OWL). [9,10] Using OWL to describe the data model provides myriad benefits, once the data is safely off boarded. Data can be reasoned about using automated and probabilistic methods. This makes the task of creating new prognostic algorithms simpler and faster.

The Arbor Data Model uses the RDF data model to model the discrete sensor events received and the actuator events sent by the embedded platform. Arbor stores and marshals this data in several simple formats that are known to be efficient on embedded platforms: Comma Separated Values (CSV) and JavaScript Object Notation (JSON) [11]. Band XI has also investigated supporting the Common Data Format (CDF) [12], using COBRA data from a Bradley Fighting Vehicle.

From a vehicle systems domain perspective, the core functions of the system architecture are to support the following:

- Sensing & Logic Integration
- Provisioning
- Operator User Interface
- Storing and Offboarding Data
- Peer-to-Peer Messaging

We explore each of these functional requirements below.

### *Sensing & Logic Integration*

Considering the earlier critical constraint of embracing legacy applications, we already know that the sensing and logic portions will be written in different programming languages. This elevates the need to bridge the gap between the languages in order to transmit the sensor data. The need for a system level mechanism for integrating the sensing and logic portions led us to consider multiple different integration patterns. We chose an integration pattern based on the design of the World Wide Web. The details of this design decisions are described in our paper *"Bringing Best Practices from Web Development into the Vehicle"* [13].

### *Operator User Interface*

The in-vehicle operator's user interface acts as a client to the sensing platform, just like an algorithm is a client to the sensing platform. The operator user interface discovers the data for the sensors it cares about and then locates the latest values of those sensors and displays them to the user. The HTTP interface to the data, along with the language

independence of provisioning, makes it possible to implement the user interface in a variety of technologies and languages (C, Java, Flash, JavaScript, et al). This is a huge advantage, as many user experience experts are used to developing systems in languages other than those which may be ideal for interfacing with sensors or expressing an algorithm. By using a language neutral interface, we allow the authors of each portion of the system to work with the most appropriate tool for their problem and platform.

### *Storing and Offboarding Data*

Algorithms will need historical trend data in order to diagnose problems. Embedded platforms are inherently resource constrained. This presents contention over the amount of data that should be retained. We've taken the position that no algorithm can be useful if the platform dies, because it runs out of memory or storage. We've implemented a purging task within the Arbor device server that prunes historical data in order to keep the system running smoothly. This purging task runs periodically. When it purges data, it leaves a breadcrumb in the trace data. This information is valuable for algorithms, because it tells them the point past which no data is available. In certain situations, it may be important to distinguish between the absence of any data from the sensor (i.e., a broken sensor) and the fact that data was reported but has been deleted to save space.

## PEER TO PEER MESSAGING

There are scenarios where multiple embedded platforms, operating in a mesh network configuration want to read each other's sensor values. An example of this is a team with a set of devices at a large sporting event using chemical sensors to monitor the perimeter for threats. If any single device detects an alarm, it is critical that the other devices and their operators are made aware of the alarm. We use the *zeroconf* [14] technology to address this scenario. With *zeroconf*, each node in the mesh can listen for the other to put up a notification flag, which means that it has an alarm. The other nodes will detect this flag very quickly and then fetch the sensor values from the device that raised the flag. Fetching the values happens over the HTTP interface. By using *zeroconf* and HTTP, along with common, open data formats, we support this scenario easily, even if each of these devices are supplied by different vendors and the resident applications have been implemented different vendors employing their own choices of programming language.

## PROGNOSTIC & DIAGNOSTIC APPLICATIONS

Our scenario calls for the ability to arbitrarily create new algorithms in support of diagnostic and prognostic needs for a variety of vehicles and other mobile or remote equipment. Given that mission, we know that we could face a wide array of potential *controls* (*sensors* or *actuators*) in an infinite set of potential configurations. Additionally, as time passes, we will gain knowledge and experience that should help guide all in designing better algorithms in the future. Therefore, our expectation is that we will need to deploy new algorithms to the mobile or remote equipment from time to time in the future at irregular intervals in response to circumstances.

An illustrative example communicates the capabilities of Arbor best. With that in mind, we describe the context and the implementation of a simple, multi-language example. For the moment, we will set aside the mechanics of how algorithms are provisioned to the remote platform. Instead we will assume that a new prognostic algorithm has been provisioned to the platform and explore how it is integrated and provides feedback to the vehicle occupant and a remote observer capable of *peeking in* on the vehicle.

### *Example Scenario*

In this example scenario, we assume a commercial or military vehicle out in the field with a standard set of operating sensors that include the following controls:

- Fuel level sensor
- Clock
- Odometer
- Global positioning system (GPS) device

This simple algorithm will monitor the fuel level sensor to determine the amount of fuel being used over time and the odometer to determine how far the vehicle has traveled. By monitoring these sensors, the algorithm should be able to calculate the following outputs:

- Fuel economy
- Distance to empty

A more interesting feature that could be added with the availability of the GPS and access to a database of fueling depots would be to tell us which fuel depots are likely to be in range of the vehicle. If the vehicle had a display unit inside the vehicle, we would expect to see a map indicating the location of the vehicle itself and color coding of fuel depots positioned on the map indicating that they are in-range (green), likely-in-range (yellow), or out-of-range (red), given our prognosis for *fuel economy* and *distance to empty* from the current location.

### *The Root System: Device Server*

At the base of the architecture we have implemented what we call the *Device Server*. The Device Server stores the data from available *sensors* and *actuators*, more broadly categorized as *controls*. This provides the system with the ability to both listen to sensors to obtain readings and to send messages to actuators directing them to perform some

action. For example, we may listen to the speedometer to obtain the vehicle speed, which represents an interaction with a *sensor*. We may also have the capability to direct a solenoid to change a drill angle, thereby interacting with an *actuator*. For commercial applications, most all sensor and actuator communication travels across a Controller Area Network (CAN) bus using a common message specification, such as SAE J1939, J1750, or various others. It is also possible to interact with any variety of other control connections, such as I2C, GPIO, SPI, MIL STD 1553, standard serial (RS-232, RS-485, RS-422, or MIL STD 188), and many others.

We constructed the reference implementation for the Device Server using embedded Java, the OSGi architecture, and the Eclipse Equinox runtime [15]. The implementation employs the Eclipse DeviceKit framework to assist in managing the low-level device interfaces in order to talk to the various connections. On top of this foundation rests a device Servlet that handles interaction with applications that want to communicate with the controls.

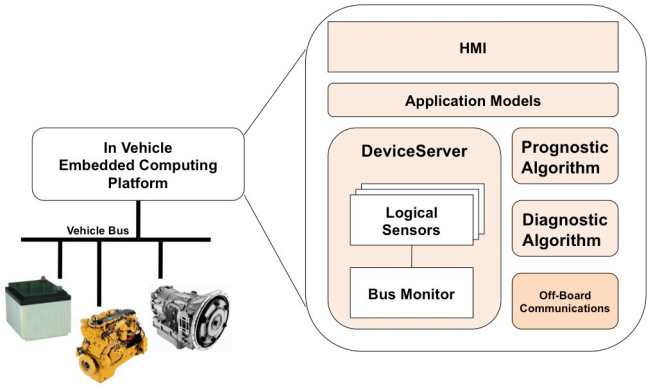


**Figure 2: Device Server in Context**

As shown in Figure 2, the Device Server wraps some logical models of the controls and communicates with them through the bus monitor. Application models, prognostic algorithms, or diagnostic algorithms can easily interact with the Device Server through the HTTP API. In this way, the Device Server represents the data vault for Arbor, storing data from the hardware interfaces to anchor the full application platform.

### *Anatomy of an Algorithm*

The algorithm itself is implemented in C/C++ and can be segmented into two families of components: (1) algorithm specific code, and (2) Arbor API specific code. As shown in Figure 3, the algorithm specific code components are shown in black boxes, while the Arbor API specific code is shown in white boxes. The implementation makes use of the **curl** [16] library, colored in gray, to perform the calls to the Device Server.

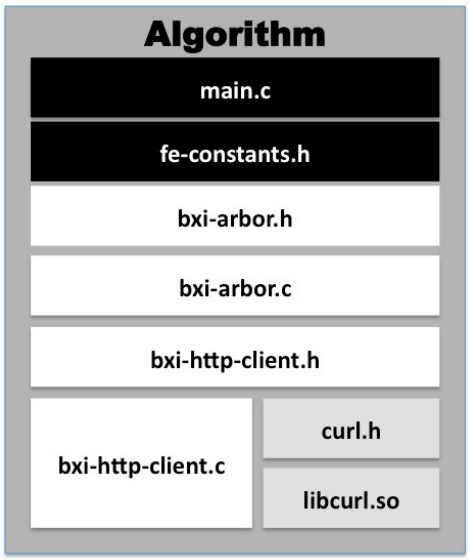


**Figure 3: Language & Notification Brittleness**

### *Algorithm Initialization*

Each algorithm that is deployed to the platform will need to perform some initialization activities to register with the Device Server. Once completed, the algorithm will have the capacity to read from actual underlying controls. Additionally, algorithms can request the creation of virtual controls to which they can report, or post, their calculated findings so that they can be reported to the vehicle occupants or logged for later transmission.

Often, the first step in the process is to obtain a complete inventory of the available *shortnames* for the set of available controls. We accomplish this by querying the device *extra-data*, as shown below in Figure 4, which responds with a list of *shortnames* in the body of the response. *(Note: The reader may wish to refer to [13] for more information regarding terminology used in this section).*

The next step in the process is to get the complete *extra-data* for the *shortname* of interest, in this case `Odometer`. We will likely be iterating over all of the *shortnames* returned to build a full list of the available controls, but here we just look at the one of interest to us. The invocation of the query is shown below, along with an excerpt of the expected response. Within the list of *extra-data* items, the one attribute that we are immediately interested in identifying is the *URI*, which will provide us with the pointer to where we will ultimately pull the data for the algorithm's input.

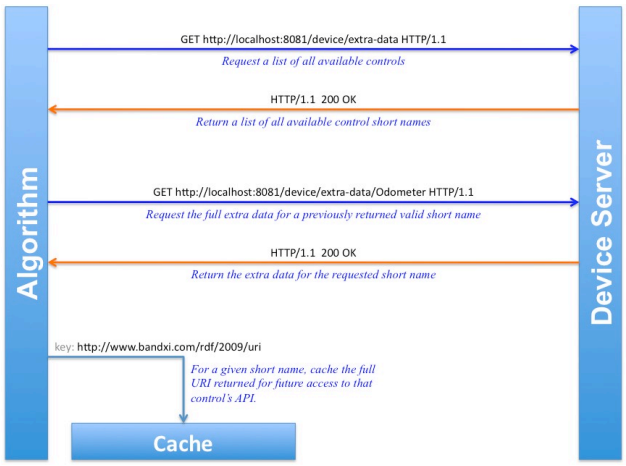


**Figure 4: Getting Available Controls**

The attribute that we will need to reference is the key-value pair of:

http://www.bandxi.com/rdf/2009/uri ⇒

http://www.bandxi.com/cbm/2009/Odometer

Our algorithm only knows that the full URI specified as the value above is the one in which we are truly interested. Therefore, when we see it, we must infer that the `Odometer` *shortname* is the actual s*hortname* that we will use as a reference to obtain the input data going forward. It is important not to assume that we know the *shortname a priori*, because it may not hold the information we actually want. Only once we have matched the actual long URI, can we be certain that the s*hortname* is the correct one.

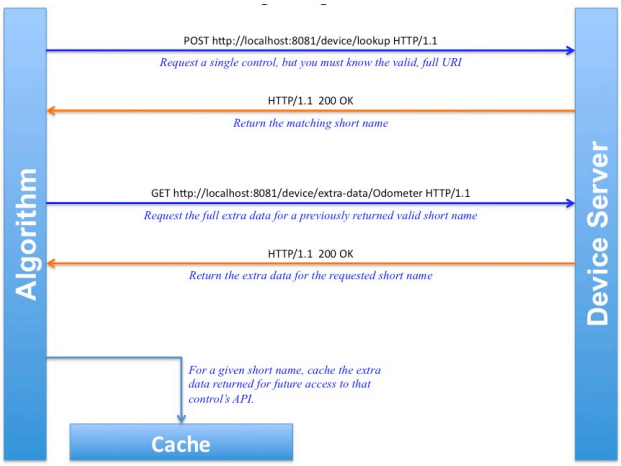


**Figure 5: Getting A Single Control**

In many cases, the algorithm may know the exact full URI for which the *shortname* is being sought. A convenience API is provided in which the long URI can be sent and the correct *shortname* returned, as indicated in Figure 5**.** The *shortname* is returned in a <*key*>=<*value*> pair format, so that the requested long URI can be validated against the <*key*> to ensure the correct *shortname* has been returned.

### *Monitoring the Controls*

Now that we have confirmed the existence and identity of the *shortname* of interest as `Odometer`, we can use it to obtain the current control value by sending the request illustrated below in Figure 6. The algorithm pulls the data at whatever polling interval it requires in order to perform its calculations.

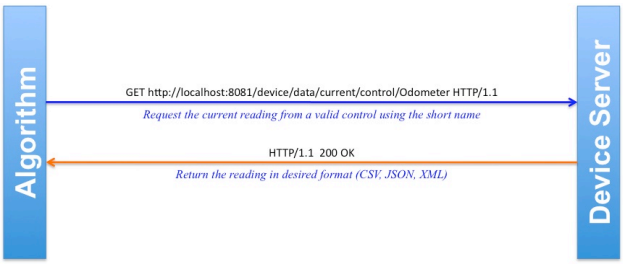


**Figure 6: Getting Current Control Reading**

In the same manner, the algorithm can also collect readings for the fuel level sensor. Given knowledge concerning the miles traveled read from the odometer, fuel consumed read from the fuel level sensor, and the amount of time that has passed in order to synchronize the readings, the algorithm can calculate the mission's *FuelEconomy* by simply assessing the number of miles per gallon utilized. Additionally, knowing the amount of remaining fuel and the fuel economy, the algorithm can also estimate the *MilesToEmpty*, that is how much further the vehicle can travel before running out of fuel. In this example, we have a very simple diagnostic (*FuelEconomy*) and a very simple prognostic (*MilesToEmpty*).

### *Providing Feedback Through Virtual Controls*

When a running algorithm has information to report concerning the vehicle health, we need to provide a way to report that information. However, before we can provide spot updates of a particular health assessment from the algorithm, we need to create a virtual control inside the Device Server that will host the new information that we intend to report. The way in which this is done is illustrated below in Figure 7 for *FuelEconomy*, where we do a POST to define the virtual control that we will be updating for recording and consumption by others.

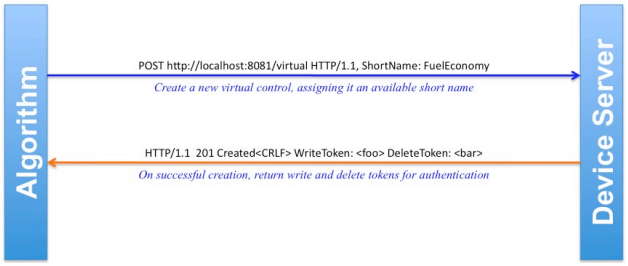


**Figure 7: Creating a Virtual Control**

As illustrated in Figure 8, once the virtual control is defined, it is a simple POST action to write the current value. The device server will log this value and it is available to anyone interested in the current value, just the same as any real control would be available.

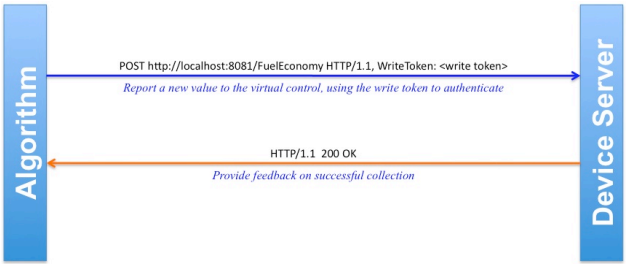


**Figure 8: Reporting to a Virtual Control**

It would be reasonable to have an application monitoring the algorithm's output to provide information to the vehicle operator or commander regarding what conditions are being predicted. Figure 9 below shows a simple example user interface for the examples of the *FuelEconomy* diagnostic and the *MilesToEmpty* prognostic algorithms. The values are presented as gauges on a dynamically displayed graphical instrument cluster that was provisioned along with the algorithm package to inform the vehicle occupants of findings in real time. The application reads these values through the exact same mechanism that the algorithms used to read the fuel level sensor and odometer.


**Figure 9: Virtual Control Display for Example**

As described, any application can use the RESTful HTTP API to send or receive readings on the platform. The algorithm described was written in C/C++, but just as readily could have been written in Java, MATLAB, FORTRAN, C#, or any other language. Likewise, although the screen for the gauges was written in Java, it could have readily been written in C, Flash, JavaScript, or any technology available for the platform on which it was running.

As an aside, because the API is defined using HTTP, the various components are no longer constrained to running on the same computer. In fact, it could be three separate computers: (1) the Device Server monitoring the CAN bus, (2) the display unit embedded in the dashboard, and (3) an alternate sandboxed, single board computer that runs algorithms. In fact, during the development process we have used this capability to have a developer in North Carolina start his monitoring application and connect to the Device Server resident in Arizona, because only one engine control unit was available at the time. We can easily firewall this capability for production deployments using standard network configurations, so it does not present a security liability.

## COMMERCIAL APPLICATION SUCCESS

For the past ten years, the authors of this paper have been developing these kinds of open, extensible, remotely maintainable information systems for a wide variety of applications, but particularly for vehicles and mobile units. The system architecture has been refined through application to a several relevant domains, including:

- telematics systems (safety & security, vehicle status reporting, infotainment)
- vehicle diagnostics and prognostics
- chemical, biological, radiological, nuclear, and explosives (CBRNE) detection
- radio frequency identification (RFID) for retail and military logistics
- situational awareness information sharing (video, voice, photo, and text)
- medical and remote biometric instrumentation
- mining and construction heavy equipment, such as salt harvesters, foundation drills, freight engines, and wheel loaders.

With this approach, we can quickly address new application requirements by defining new services and building new code bundles that sit on top of the existing frameworks, components and applications. This means that our development cycle times for new applications can be measured in days or weeks, instead of in months and years.

## BENEFITS

While there is *No Silver Bullet* [17], the Arbor architecture has clear advantages over traditional approaches and moves us closer to the ideal vision where software development becomes a true engineering discipline.

**Legacy Continuity:** Given the ubiquity of HTTP support in all languages, the Arbor architecture levels the playing field for legacy and new applications. Any application can be extended with an adapter that interacts with the Device Server, rather than directly to the device driver. Not only does this reduce system brittleness, but it allows several applications to share access to the controls. Performance has not been a problem for commercial applications, but we would not recommend this for fire control systems as yet.

**Reduced Cycle Times:** On many occasions, we have shortened development cycle times by an order of magnitude. In one case, a team that had devoted three months to developing an interface to the Media Oriented Systems Transport (MOST), an optical vehicle bus for media systems. Using an earlier approach similar to Arbor, we were able to build a working implementation enabling pre-existing audio control screens to connect to the radio and CD player on the MOST bus. That effort took two days. Later, on, another project we were asked if we had any ideas on convoy spacing management. Again, with three days of effort we had a prototype feature running that provided convoy spacing information and collision warnings by leveraging the pre-existing GPS services, user interface shell, wireless communication services, and audio alert services.

**Higher Quality:** With the cleaner, simpler interfaces, the system design achieves the objective of being highly cohesive, but loosely coupled. The resulting clean interfaces can be tested using simulators well before bench testing and equipment testing are ready. In that way, quality is designed in from the start as testing is done from day zero. Additionally, by relying on the Device Server, which is well tested and robust, any problems encountered can usually be isolated with repeatable test cases that prove where the error lies. While building a complex RFID system for a worldwide company, this approach of clean interfaces and testing against simulators achieved a quality that was immediately recognized by management. The system had two orders of magnitude fewer known bugs when it shipped than any other product in the division.

**Cost Savings:** Software development is an inherently labor-intensive practice. Reduced cycle times and higher quality translate directly into lower labor costs – whether for new development, integration, or maintenance.

## SUMMARY

Arbor brings a significant number of benefits to system integrators. These benefits include shorter development cycles, cost savings, and higher quality, while protecting investments in legacy applications. Arbor based systems have achieved a high level of technology readiness and are primed to help solve the prognostic and diagnostic challenges of the future.

## REFERENCES


[1] http://www.acq.osd.mil/osjtf/index.html

[2] http://www.nswc.navy.mil/wwwDL/B/OACE/

[3] http://www.eclipse.org

[4] http://www.osgi.org

[5] Brooks, Dr. Fred P., and Gerrit Blaauw, Computer Architecture: Concepts and Evolution, Addison Wesley Professional, 1997.

[6] Klyne, Graham and Jeremy Carroll, Resource Description Framework (RDF): Concepts and Abstract Syntax, World Wide Web Consortium (W3C), 2004.

[7] Manola, Frank and Erica Miller, Editors, RDF Primer, World Wide Web Consortium (W3C), 2004.

[8] Antoniou, Grigoris, A Semantic Web Primer, 2nd Edition, MIT Press, 2008.

[9] Smith, Michael, Chris Welty, and Deborah McGuiness, OWL: Web Ontology Language Guide, World Wide Web Consortium (W3C), 2004.

[10] Lacy, Lee, Owl: Representing Information Using the Web Ontology Language, Trafford Publishing, 2005.

[11] Crockford, D., *"JavaScript Object Notation: Request for Comment 4627"*, The Internet Engineering Task Force: Network Working Group, 2006.

[12] http://cdf.gsfc.nasa.gov/

[13] Branigan James, John Cunningham, Patrick Dempsey, Brett Hackleman, and Paul VanderLei, *"Bringing Best Practices from Web Development into the Vehicle"*, Proceedings of the 2009 Ground Vehicle Systems Engineering and Technology Symposium, Detroit, 2009.

[14] http://www.zeroconf.org/

[15] http://www.eclipse.org/equinox/

[16] http://en.wikipedia.org/wiki/CURL

[17] Brooks, F. P., *"No Silver Bullet: Essence and Accident in Software Engineering"*, Proceedings of the IFIP Tenth World Computing Conference, pp. 1069-1076, 1986.