

Bringing Best Practices from Web Development Into the Vehicle

James Branigan

Band XI International, LLC
Creedmoor, NC

John Cunningham

Band XI International, LLC
West Hartford, CT

Patrick Dempsey, Brett Hackleman, and Paul VanderLei

Band XI International, LLC
Bracey, VA, Scottsdale, AZ, and Grand Rapids, MI

ABSTRACT

Building embedded systems is nothing like building desktop applications, as the hard real time requirements and relative harshness of the operating environment further constrains design choices to meet real world needs. Those familiar with mainframe or server farm hosted, high volume, wide bandwidth applications know similar harsh computing environments for application development. Given that more man-hours have been devoted to web application development over the past decade than have been devoted to embedded application development, there may be some valuable lessons to be learned that can be adopted by the embedded community for in-vehicle computing. The best web application development teams successfully apply the notions of Representational State Transformation (REST) and Resource Description Framework (RDF) to handle the increasing demands on their sites. We have taken these technologies and applied them to the smaller scale vehicle telematics platforms (PowerPC, ARM, and Atom) to test their viability. This paper describes how we approached the design decisions that enabled us to successfully wrap a commercial J1939 CAN bus with a miniature web server that provides a REST API for applications to interact with an engine control unit. The architecture has been successfully deployed for custom mining and construction equipment.

INTRODUCTION

Software applications for desktop systems enjoy an abundance of resources, such as available memory, disk space, processing power, and network bandwidth. Embedded and server applications struggle with the issue of resource scarcity in contrast to desktop systems. The economics behind the scarcity differ, with the embedded systems characterized by low cost, low power devices and the server market characterized by high cost, high utilization devices. The scarcity of resources manifests itself in both marketplaces and places responsibility for efficient use of those resources in the hands of application developers. As the expectations of embedded applications have increased, we have seen an increase in the importance of application integration and distributed communications to support data sharing and multi-vendor, multi-programming language solutions.

When attempting to address the application integration and distributed communication challenges of embedded systems, we focus the search for potential solutions in problem

domains that share a common respect for the resource scarcity found in the embedded computing domain. The idea behind this approach is that *it is easier to stay efficient than it is to become efficient*. As embedded platform processing power, available memory, disk space and network bandwidth have increased over time, we find that approaches developed for the resource efficient server environments over a decade ago are now applicable and viable in the embedded domain. Unfortunately, some developers have chosen to use the increased embedded platform capabilities to deploy the kinds of applications developed for desktop applications. We have directly observed that this approach tends toward failure and should be avoided.

In this paper, we describe an embedded system architecture, called Arbor, which adopts successful, proven techniques from the World Wide Web application server domain. This approach readily solves both the multi-vendor, multi programming language integration problem

and the distributed communications problem for sharing data with peers and remote servers. We have successfully used this system architecture to develop applications that sense and control custom mining equipment employing SAE J1939 messaging over a Controller Area Network (CAN) bus. We have also used this system to allow third party vendors to develop, integrate, and install prognostic and diagnostic algorithms that provide advanced, *just in time* maintenance capabilities for military vehicle transmissions.

ON DISTRIBUTED COMPUTING

During our research efforts we discovered a paper entitled "*A Note on Distributed Computing*" [1] published in 1994. This paper should be required reading for anyone designing, implementing, or using distributed software. It describes the flurry around, and ultimate failure of, the distributed systems programming models that appear every few years, usually linked to new programming languages. Despite being fifteen years old, this paper remains highly relevant today.

In "*A Note on Distributed Computing*", Waldo *et al* make the case that objects interacting across a distributed system (and network) are fundamentally different from those within a single address space. Designers and implementers must explicitly contend with the challenges presented by latency, memory access models, concurrency, and the higher likelihood of partial failures. Unfortunately, most attempts at building distributed systems failed to adequately support the basic requirements of robustness and reliability.

Based on this paper and our own research and experience building commercial systems, we identified the primary lessons learned, as they apply to any distributed services design:

- Do not attempt to hide the differences between local and remote services
- Do not provide generic marshaling support, it only encourages bad programming habits
- All calls to remote services should be asynchronous
- Plan for partial and full failures when talking to remote services
- Encourage, and where possible, force good programming habits through careful API design

ON SYSTEM INTEGRATION PATTERNS

Integrating a system requires the connection of different parts. These parts often come from multiple vendors. The parts may also be made of multiple materials. In the case of a software system, the different materials are different programming languages and operating systems.

There are two orthogonal concerns that must be addressed when integrating parts of a software system.

- *Call direction*, as in determining which part initiates the call and which part receives the call
- *Call payload*, as in what information is sent with the call and how is the information packaged

There are three popular integration patterns commonly adopted in production systems that address the two concerns.

- A language-to-language binding
- A *publish and subscribe* mechanism
- A *data format* accessible over a stateless, language neutral transport

We will cover the benefits and drawbacks of each approach below, beginning by briefly describing each of the three common integration patterns.

Language to Language Binding

Language-to-language, as an integration pattern, works regardless of the call direction. Work is performed to allow the programming language that is initiating the call to invoke a function or subroutine in a second language. The corollary requires establishing the mechanism by which the invoked language can initiate callbacks into the first language. There is always syntactic mapping occurring here, where the payload may need to be massaged from its representation in one programming language to an equivalent representation in another programming language. There also may be semantic mapping occurring, in the cases where concepts must be represented differently in the two programming languages. One common place where semantic mapping occurs is when mapping between an object oriented language and a procedural language. A concrete example of a *language-to-language* binding is the Java Native Interface, which is commonly used to connect Java code to C code. [2]

Publish and Subscribe Binding

The *publish and subscribe* pattern takes a specific position on call direction. Producers *push*, or publish, information to a central broker. This broker maintains a list of consumers who have registered interest in the published messages. The broker then pushes the producer's publication on to all of the registered consumers. In this integration pattern, consumers may register their interest in only certain types of publications. A publisher may publish regardless of how many consumers care about the publication. The format of the call payload must be reified, so that the loosely coupled

producers and consumers can both interpret the data the same way. This removes the semantic mapping burden that is present in the *language-to-language* binding. However, the syntactic mapping remains as a cost to be incurred for attaching a particular language to the *publish and subscribe* system. The *publish and subscribe* pattern is well described by Hohpe and Woolf in *Enterprise Integration Patterns* [3] and Chappell in *Enterprise Service Bus* [4].

Standard Data Format Over a Stateless, Language Neutral, Transport Binding

The standard *data format* over a stateless, language neutral transport pattern takes a specific position on call direction, but one that is the inverse of the *publish and subscribe* pattern. Here, consumers *pull* information from the producers. This has the benefit of avoiding the need to maintain a list of registered consumers. The stateless property of the system allows the network topology to change without introducing systemic problems. Like the *publish and subscribe* binding, the call payload format is reified, to reduce the semantic mapping costs. To reduce the syntactic mapping costs, a language neutral transport is used for transmission of the call payload. Typically, a transport is chosen which already enjoys full support in a large number of programming languages. The World Wide Web is the largest, most successful embodiment on this approach.

Pattern Evaluation

A *language-to-language* binding is undesirable for many reasons. The primary reason being that it is very difficult to do well. This type of integration often introduces memory leaks, due to errors that occur when mapping the representations of the call payload. *Language-to-language* bindings are inherently brittle and changes to underlying shared objects can impact the integration. In the concrete domain of condition based maintenance applications, this requires work on a *(number of sensors) × (number of algorithms)* basis.

Moving to the *publish and subscribe* system helps with a few of the issues associated with the *language-to-language* binding. Most importantly, it turns the multiplicative complexity into an additive one. Each portion of the system only needs to provide integration to the *publish and subscribe* bus. However, depending on how that integration is done, it is still susceptible to brittleness. New complications are added to a system by using this style of integration. Since all intra-system communication is happening through the *publish and subscribe* bus, it can be very difficult to debug problems which involve multiple messages crossing the bus. Since events sent over the bus are temporal, system wide logic errors can be introduced when a subset of the nodes on a bus are disconnected due to a network partition. This would be unusual in a traditional

enterprise IT shop, but is expected in an ad-hoc mesh-networking environment.

In a *publish and subscribe* system, data producers publish or *push* their information to the central broker. This broker then consults the list of subscribers to that information and pushes the information to them. There are many problems with this approach that place fundamental restrictions on a systems ability to scale. One such problem is that all interested parties are notified at the rate of production, rather than at the rate they are prepared to consume. Another issue is that the list of interested consumers that must be maintained by the central broker. The required subscriber list maintenance becomes problematic when used in an environment where nodes come and go frequently. There is a bootstrapping issue as well, when a node arrives late to a bus and has misses an initial set of publications.

Defining a *data format* over a stateless, language neutral transport provides a better solution than either a *language-to-language* binding or a *publish and subscribe* system. As long as the chosen programming languages natively support the transport, you can safely avoid the brittleness problem. The *data format* stays consistent, which enables portions of the system to be upgraded or patched, without impacting the other unchanged portions. Additionally, since data is *pulled* rather than *pushed*, we avoid the network connectivity issues of the *publish and subscribe* bus. The pulling of data also avoids the need for centralized registration lists of parties interested in a piece of information. Pulling can occur at any time, so the bootstrapping problem is also avoided.

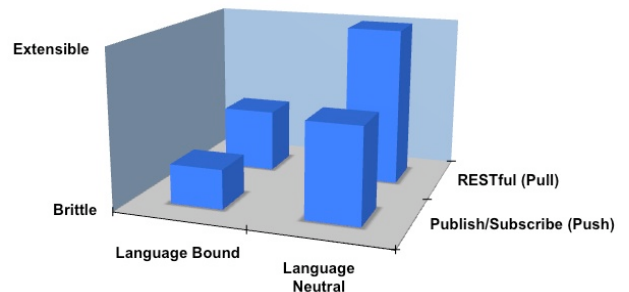


Figure 1: Language & Notification Brittleness

Our design objective is to create an extensible system that can accommodate both legacy and newly developed components – in other words, a system that is easily extensible, rather than brittle. Figure 1 summarizes the qualitative nature of selecting combinations of underlying integration technologies. The conclusion, based on experience, is that pull is better than push and language neutral is better than language bound. When combined, the choice falls to employing a solution with a standardized data format over a language neutral transport.

Choosing a Stateless, Language Neutral Transport

We surveyed the common distributed systems and integration patterns in production use and chose the approach taken by the most successful large distributed system in the world today: *the World Wide Web (WWW)*. The HyperText Transfer Protocol (HTTP) [5] was developed as the basis of the World Wide Web. The architectural style, known as **RE**presentational **S**tate **T**ransfer (REST), used to design HTTP is described in Dr. Roy Fielding's PhD thesis [6]. We refer to applications adopting this design as being *RESTful*.

Ubiquitous support in all modern programming languages was a considerable factor in choosing HTTP as the language neutral transport. There is wide experience and an array of information about HTTP available on the Web and in literature. Practically speaking, that means a large group of developers won't need to learn anything new to adopt this approach. This helps to lower the overall system cost by reducing the training and support burden. This means that developers can get up and running quickly.

CHOOSING A STANDARD DATA MODEL AND FORMAT

There are many different technologies that can be used to describe a data model: Unified Modeling Language (UML) [7], eXtensible Markup Language (XML) Schema [8], or Resource Description Framework Schema (RDF-S) [9] and many others.

In determining the data model to use, we had one primary issue to consider. *Does the data model have an open worldview or a closed worldview?* The distinction between the two can be most clearly seen when one attempts to extend a system beyond the use cases for which it was first designed. In closed worldview systems, new use cases can result in significant rework and extremely high cost. This is because the data model must be changed to incorporate the new data that the new use cases require. Existing systems must be upgraded or retrofitted to support handling the new data model. By contrast, in an open worldview system, additional use cases can be added easily, since the existing parts of the system don't make assumptions about things they don't understand. A concrete example of the philosophical difference between the two approaches can be seen in basic Boolean logic. In a closed world, if a statement cannot be proven to be true, then it is assumed to be false. In an open world, if a statement cannot be proven to be true, then no assumption can be made about the truth of the statement.

When looking at the data models available, we ended up choosing to use RDF-S to describe our data model. RDF-S has an open worldview, in contrast with XML Schema. Additionally, RDF-S has multiple data format representations and it is possible to make additional formats.

This provides the flexibility to use one data model with different formats in use, depending on the capabilities of a specific system. This also allows us to avoid the performance penalty of relying heavily on XML based formats on embedded systems, while still making XML representations available on enterprise systems.

There are many different popular data formats in use today: XML, RDF-XML, JavaScript Object Notation (JSON), Comma Separated Values (CSV) and others. We have defined representations for our data model in RDF-XML, CSV and JSON. We have also evaluated the possibility of representing our data model in the Common Data Format (CDF), which is in wide spread use within the agencies of the US Federal Government.

THE ARBOR DATA MODEL

The codename 'Arbor' is used for our data model and system architecture. As mentioned, we chose to use RDF-S to create our data model. We used a technology known as the Web Ontology Language (OWL) [10,11] to represent the classes and properties of our data model. By using OWL and restricting ourselves to a specific subset of its capabilities (OWL-DL), it is provable that data based on our data model can be reasoned about in a deterministic, finite amount of time. This is important for algorithm authors, because they need to use probabilistic and automated techniques to develop prognostic and diagnostic algorithms from vehicle sensor data. Using a data model that provides computational guarantees is a win for algorithm authors.

The Arbor data model is composed of four OWL Classes and several properties. We cover the classes and the more important properties below.

- Control
- Datapoint
- Class
- Entity

In the Arbor data model, the unit of recording a sensor value, or posting a new value, is the *Datapoint*, comprised of the three atomic pieces of information.

Unique Event Timestamp	Control Identifier	Value
------------------------	--------------------	-------

In the CSV format, each event is presented on its own line.

1001, temperature, 85

In the JSON format, each event is presented in its own JSON object.

```
{
  "timestamp": "1001",
  "control": "temperature",
  "value": "85"
}
```

The data presented here are just individuals of the *Datapoint* class. The *Unique Event Timestamp* is a property on the individual. The *Control Identifier* is another property and the *Value* is the third property. In Arbor, we require that the Control Identifier actually be a reference to an existing individual of the type *Control*. The *Control* individual holds all of the critically important information for applications dealing with the data. However, *Control* properties don't vary with each data point, and as such are not stored with each event. In Arbor, we call these properties *extra data*.

A *Control* with the control identifier *temperature* might have the following *extra data* properties (in CSV format). The URL prefixes of the property keys are not shown in this example for layout purposes.

```
uri          http://www.bandxi.com/engineTemp
encoding     UTF8
type         float
name         Engine Temperature
units        C
highWarnThreshold 75
highErrorThreshold 90
```

There are two additional parts of the data model.

- A Class has a unique identifier, which should be a URL, and also has a list of Control URI property values. A *Class* individual states that a implementer of that *Class* type must contain controls instances with the same URI values as those specified in the *Class*.
- *Entities* list their specific control identifiers and the *Classes* that the entity implements.

Here is a specific example of a class based on the *Control* already specified.

```
Class
Identifier = http://www.bandxi.com/example/Foo
Uri = http://www.bandxi.com/engineTemp
```

```
Entity
Identifier = {Vehicle VIN Number}
Control = temperature
Control = ...
Class = http://www.bandxi.com/example/Foo
```

Entities and *Classes* are used for several purposes. They enable easy synchronization of sensor values between peers in mesh networking environments. They also allow programmers to understand the relationships between sensor values and to know that if a given entity declares that it implements a particular *Class*, then the URIs defined by that class will be available.

RESTFUL INTERFACE: DEVICE SERVER

The discussion to this point has centered on the data model and data formats. It is equally important to understand the RESTful interface to the data and the available mechanisms available to manipulate and query the data. There is a component in Arbor called the *Device Server* that implements this RESTful interface.

The *Device Server* is responsible for maintaining history of *Datapoints*, as well as descriptive information on *Entities*, *Classes*, and *Controls*. The *Device Server* has the following capabilities:

- Create a New Control
- Read an Existing Control
- Delete an Existing Control
- Create a New Entity
- Read an Existing Entity
- Delete an Existing Entity
- Modify an Existing Entity
 - Add a Class Declaration
 - Remove a Class Declaration
 - Add a Control Instance
 - Remove a Control Instance
- Create a New Class
- Read an Existing Class
- Delete an Existing Class
- Read the latest Control value
- Read the previous Control value history
- Write a Control value
- Lookup all Entities which declare a Class
- Lookup a control identifier by Control URI

Local programs make use of these capabilities as well as remote peer systems, which can also use these capabilities to inspect the state of the system.

EXAMPLE: CONDITION BASED MAINTENANCE

It is useful in understanding Arbor to run through an example, for instance adoption in the field of Condition Based Maintenance. For simplicity, let's call logic executing on the system an *algorithm*. Algorithms will need to know the input sensors that they require input from in order to execute. They will also need to know the output that they will be producing and making available to the rest of the system. Each algorithm will be able to identify the sensors that it requires by their URL. Additionally, it can identify its output by a URL as well.

In order to make use of data from a particular sensor, the algorithm must have internalized the properties that it needs to run. As a simple example, imagine an algorithm that only

monitors engine temperature from a J1939 bus. At development time, the algorithm would know about the concept of a *highWarnThreshold* and a *highErrorThreshold*. The algorithm would know that these *extra data* properties are of the type *float* and may be positive or negative. It would also know that the interpretation of the value is dependent on the value of the *units* property (Celsius or Fahrenheit). Since this information is baked into the code of the algorithm, it doesn't need to fetch the URLs. It's important to leave the properties as URLs, because this provides a namespacing capability, which keeps one algorithm's notion of units from colliding with another. The verbosity isn't a concern for the extra data information, because it only exists once per sensor.

Once the algorithm has executed, it needs to produce an output. It does this by creating a control and writing its output value. The control is associated with the entity of the current vehicle. Any necessary class declarations can be made on the vehicle entity after the control is created.

When the vehicle is sensed by a peer system, such as a scan tool in a motor pool, the scan tool can communicate to the HTTP interface and retrieve the sensor values and algorithm output values.

EXAMPLE: MINING & CONSTRUCTION

The above described system architecture has been employed to build the operator displays for mining and construction equipment. For example, one system monitors and controls the engines, auger, and elevators of a salt harvester over a J1939 vehicle bus. A salt harvester operates much like a grain combine, as it drives across a salt flat, grinding salt off the ground and feeding it onto elevators which carry it upward and outward for deposit into large dump trucks riding alongside. The Arbor based salt harvester application achieved Technical Readiness Level 8/9, as it is now operating *in the wild* on the salt flats. Additionally, another system to monitor and control the engine, hydraulics, and solenoids of a foundation drill is ready for testing and should be fielded soon.

EMBEDDED PLATFORMS

At the beginning of the paper, we described the embedded platforms as resource constrained. We have successfully deployed this system on several ruggedized, low-power embedded systems. Examples of existing systems include:

- 300 MHz PowerPC, 64Mb RAM, 64Mb ROM, running Linux
- 400 MHz ARM, 32Mb RAM, 32Mb ROM, running Windows Mobile 5

SUMMARY

"We are all agreed that your theory is crazy. The question that divides us is whether it is crazy enough to have a chance of being correct."

The above quote is attributed to famous physicist Niels Bohr in regard to Wolfgang Pauli's non-linear field theory of elementary particles. When we first began discussing packing a web application server architecture into an embedded device, it sounded like crazy talk. However, being good scientists, we decided to run some experiments before dismissing it. After all, our basis for thinking it was a crazy idea was no more robust than was our basis for conceiving it in the first place! In the end, taking this approach has proven successful and relaxed some of the design constraints that had previously challenged us.

SPONSORSHIP

The work described in this paper was funded under a Small Business Innovation Research grant and funding from RDECOM/TARDEC under contract W56HZV-07-C-0525. Matt Skalny is the Contracting Officer's Representative and James Bechtel and Kenneth Fischer are Technical Points of Contact.

REFERENCES

- [1] Waldo, J., Wyant, G., Wollrath, A., and Kendall, S., *A Note on Distributed Computing, Technical Report SMLI TR-94-29*, Sun Microsystems Labs, 1994, <http://research.sun.com/techrep/1994/smlitr-94-29.pdf>
- [2] Liang, S., *Java™ Native Interface: Programmer's Guide and Specification*, Prentice Hall, 1999.
- [3] Hohpe, G. and Woolf, B., *Enterprise Integration Patterns*, Addison-Wesley, 2003.
- [4] Chappell, D., *Enterprise Service Bus*, O'Reilly Media, 2004.
- [5] Fielding, R., et al *Hypertext Transfer Protocol HTTP/1.1: Request for Comment 2616*, The Internet Engineering Task Force: Network Working Group, 1999. <http://www.w3.org/Protocols/rfc2616/rfc2616.html>
- [6] Fielding, R., *Architectural Styles and the Design of Network-based Software Architectures*, Dissertation at The University of California at Irvine, 2000. <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
- [7] Klyne, Graham and Jeremy Carroll, *Resource Description Framework (RDF): Concepts and Abstract Syntax*, World Wide Web Consortium (W3C), 2004.
- [8] Manola, Frank and Erica Miller, Editors, *RDF Primer*, World Wide Web Consortium (W3C), 2004.
- [9] Antoniou, Grigoris, *A Semantic Web Primer, 2nd Edition*, MIT Press, 2008.
- [10] Smith, Michael, Chris Welty, and Deborah McGuinness, *OWL: Web Ontology Language Guide*, World Wide Web Consortium (W3C), 2004.
- [11] Lacy, Lee, *Owl: Representing Information Using the Web Ontology Language*, Trafford Publishing, 2005.