REMOTE VEHICLE INTERACTION (RVI) 0.3

This document gives a brief introduction to the codebase of the RVI project and explains the reasoning behind some of the technical choices.

ADDITIONAL DOCUMENTATION AND RESOURCES

For a high level description, with an exhaustive master usecase walkthrough, please see the High Level Design document here

For build instructions, please check the build instructions: Markdown | PDF

For configuration and launch instructions, please check the configuration documentation: Markdown | PDF

The RVI mailing list is available at:
AGL

The RVI wiki is available at:
AGL

PROJECT MISSION STATEMENT

The Remote Vehicle Interaction project will specify, design, plan and build a reference implementation of the infrastructure that drives next generation's connected vehicle services.

- **Specify**
  Requirement specifications, test suites, integration tests.
- **Design**
  High Level Description. Detailed Description. Use Cases.
- **Plan**
  Roadmap. Milestones. Deliverables. Budgeting. Resource planning.
- **Build**
  Implement. Document. Test. Demonstrate. Deploy for download.
- **Reference Implementation**
  Provides a baseline and starting point for organizations' production-grade connected vehicle projects.

TECHNICAL SCOPE

RVI provides P2P based provisioning, authentication, authorization, discovery and invocation between services running inside and outside a vehicle.

- **P2P**
  Internet connection not required for two peers to exchange services.
- **Provisioning**
  Add, delete, and modify services and network nodes.
- **Authentication & Authorization**
  Proves that a service is who it claims to be, and has the right to invoke another service.
- **Discovery**
  RVI is considered a safe bet that can be used without unforeseen consequences.
- **Invocation**
  RVI shall be able to function over transient and unreliable data channels, but also over a reliable in-vehicle LAN.

PROJECT MILESTONES

The demonstrator milestones will act as proof of concepts and progress. Each milestone will deliver additional core functionality, up to the point where the RVI system is complete at the final deliverable.

1. **Remote HVAC control (Mid Sep 2014)**
   Pre-set the climate control of your vehicle from your mobile phone.
2. **Software Over The Air (SOTA) (Late Oct 2014)**
   Transfer, install, and validate a software package from a backend server to an IVI unit.
3. **Remote CAN bus monitoring (TBD)**
   Remotely subscribe to specific CAN frames, and have them delivered to a backend server.
4. **Remote control of IVI nav system (TBD)**
   Use remote mobile device to setup POI in vehicle's navigation system.

TECHNOLOGY CHOICES

The following chapters describe the technology choices made for the reference implementation. An overall goal was to avoid technology lock-in while easing adoption by allowing organiztions to replace individual components with in-house developted variants using their own technology.

INTERCHANGEABILITY

The only fixed parts of the entire RVI project are the JSON-RPC protocol specifications used between the components themselves and their connected services.

AGL provides an RVI implementation as a starting point and a reference. The adopting organization is free to use, rewrite, or replace each component as they see fit. A typical example would be the Service Discovery component, which must often be extended to interact with organization-specific provisioning databases.

COMMUNICATION AGNOSTICISM

In a similar manner, RVI places no requirements or expectations on the communication protocol between two nodes, such as between a vehicle and a backend server, in an RVI network. An organization is free to integrate with any existing protocols, or develop new ones, as necessary.

The reference implemantation provides an example of how to handle a classic client-server model. The RVI design, however, can easily handle such cases such as wakeup-SMS (used to get a vehicle to call in to a server), packet-based data, peer-to-peer networks without any dedicated servers, etc. The adopting organization can implement their own Data Link and Protocol components to handle communication link management and data encoding / decoding over any media (IP or non-IP).

ERLANG

Erlang was chosen to implementation the core components of the RVI system. Each component (see the HLD for details) run as an erlang application inside a single erlang node.

Several reasons exist for this somewhat unorthodox choice of implementation language:

- **Robustness**
  Erlang has the ability to gracefully handle component crashes / restarts without availability degradation. This makes a deployment resiliant against the occasional bug and malfunction. In a similar manner, redundant sites can be setup to handle catastrophic failures and geographically distributed deployments.

- **Tool availability**
  There are a multitude of open source erlang components available to handle SMS, GPIO, CAN buses, GPS, PPP links, and almost any protocol out there. Since erlang was designed to handle the mobile communication requirements that is at the center of the connected vehicle, integrating with existing systems and protocols is often a straight-forward process.

- **Scalability**
  The concurrent nature of an erlang system sets the stage for horizontal scalaility by simply adding hosts to a deployment, allowing an organziation to expand from a pilot fleet to a full fledged international deployment.

- **Carrier grade availability**
  The robustness and scalaiblity, in conjunction with the built-in erlang feature of runtime code upgrades, is a part of erlang's five-nines uptime design that is rapidly becoming a core requirement of the automotive industry.

- **Proven embedded system solution**
  Erlang has been adapted to operate well in embeded environments with unreliable power, limited resources, and the need to integrate with a wide variety of hardware.

PYTHON

Python is used to implement all demonstrations, beginning with the HVAC demo available in the `hvac_demo` subdirectory.

By using Python for the demos, which is better known than erlang, examples are given on how to write applications and services interfacing with the RVI system.

PERFORMANCE

Performance is **not** a goal of the RVI reference implementation. Instead, code readability and component interchangeability takes priority in order to ease design understanding and adoption.

One example is the use of JSON-RPC (over HTTP) to handle internal communication between components in a single Erlang node.

Using a traditional erlang solution such as genserver, the overhead for internal transactions could be cut to a few percent in comparison with the current JSON-RPC implementation. That route, however, would force all components of the RVI system to be implemented in Erlang, thus severly limiting an organizations abilities to replace individual components with their own versions.

CODE STRATEGY

All code in the RVI reference implementaion and its demonstrations are written with a minimum of complexity and "magic". Readability is paramount, even if it severly impact performance and memory usage.

All components in the RVI are kept small and distcinct, with a well-defined JSON-RPC external interface and a simple call flows.

Only three external modules (lager, bert and exo) are used by the code, with two more (setup and edown) used for release and documentation management.

The reason for minimizing external module usage is to make the code comprehensible and minimize the time a developer has to travesrse through obscure libraries trying to understand what a specific call flow actually does.

The entire reference implementation (as of the first alpa release) is 2800 lines of code, broken down into six standalone modules and one library of shared primitive functions.

JSON-RPC

JSON-RPC is used for all communication between components in an RVI system, and also to communicate with services connected to it. The ubiquity of JSON-RPC, and its close relationship with Java/Javascript, provides maximum of freedom of technology choices when new components, services, and applications are developed.