CAN signaling with AGL framework

Performances Analysis

Version 2.0

September 2016

Abstract

This document presents result obtained when using the AGL-2.0 framework for signaling and propagating event delivered by the CAN bus.

Table of contents

Document revisions

Date	Version	Designation	Author
15 Sept. 2016	1.0	Initial release	J. Bollo
16 Sept. 2016	1.1	Review	Fulup
16 Sept. 2016	1.2	Adde CBOR	J. Bollo
20 Sept. 2016	2.0	Review	S.Desneux

1. Description of the test

1.1. The data used for the test

We received from Cogent[1] the file **run11_can.pcap**. This file is a capture of CAN bus encapsulated in UDP frames. Precisely, it is a PCAP[2] file that contains timestamped ETHERNET frames. The data are encapsulated this way: CAN/UDP/IPV4/ETH/PCAP.

This file is a real capture made in Detroit area on a Cadillac SRX SUV 2015 that is the Renesas Skyline vehicles designated SRX[3].

Its characteristics are:

CHARACTERISTIC	VALUE	UNIT
total count of events	9 971 538	events
date of first event	19:06:22	HH:MM:SS
date of last event	19:19:34	HH:MM:SS
duration	00:13:12	HH:MM:SS
	792	seconds
event rate	12 586	events per second
UDP content size	199 430 760	bytes
UDP content rate	251 731	bytes per second

In detail, this trace contains 340 different CAN ids.

Note: this CAN trace is representative of an ADAS profile. A typical IVI message trace would require significantly less bandwith. Typical trace as extracted from an OBD2 end point are usualy under 500msg/s.

1.2. Tested boards

Two boards were used during this test:

- PORTER with 2 cores at 1.5GHz
- SALVATOR with 4 cores at 1.5GHz (the 4 big cores).

1 http://www.cogentembedded.com/
2 pcap files are linked to the project TCPDUMP: http://www.tcpdump.org/
3 https://www2.renesas.eu/videos/america.html?p=80&f=views

1.3. The test

A basic injector programe was implemented to replay CAN messages at real acquisition rate. The injector sends CAN messages as UDP frames from an external computer to Renesas boards. Mesurement performance is done directly on Renesas boards with "top" command.

On target boards CAN messages are received as UDP frames through a standard AGL binder that simulate a HAL(hardware abstraction layer) responsible of decoding binary CAN messages. When one or more subscribers exist, the received CAN frames are converted to JSON events and sent to subscribers using standard AGL-2.0 framework event model.

We evaluated 4 kinds of subscriber connection:

1. HAL only without any subscriber.
2. Subscriber and HAL bindings share a single binder process.
3. Subscriber and HAL bindings use two separated binders processed connected by websocket.
4. Subscriber and the HAL binder processes are linked using DBUS.

For all cases, we measured the overall CPU load.

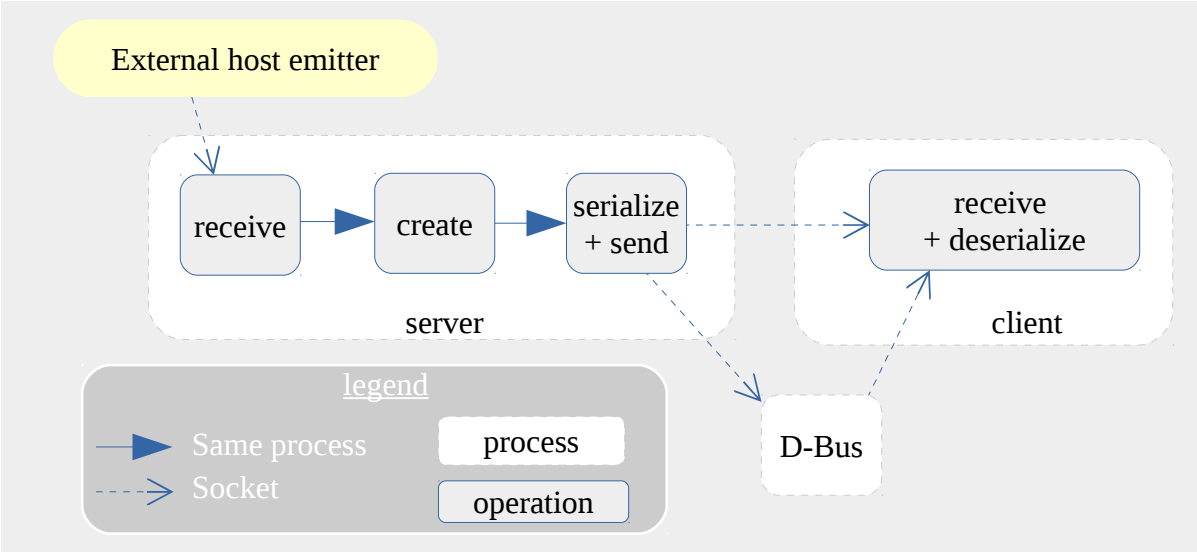
The event sent are JSON representation of the received CAN frames. The example below is the first event of the trace:

```
{"event":"udpcan\/1305","data":{"id":1304,"data":[48,59,230,95,28,245,255,255]},"jtype":"afb-event"}
```

The count of data sent is very important. An accurate estimation of the characteristics of the flow for the websocket, including internal protocol (but not TCP data), is:

CHARACTERISTIC	VALUE	UNIT
Websocket content size	1 120 331 014	bytes
Websocket content rate	1 414 135	bytes per second

1.4. Data processing flow



This figure describes the main parts of the processing and shows how they are linked together. This will allow to deduce needs of operations.

SUBSCRIBER	Receive	Create	Serialize + send	Receive +deserialize	D-Bus
None (case 1)	Yes				
Shared binder (case 2)	Yes	Yes			
Websocket (case 3)	Yes	Yes	Yes	Yes	
D-Bus (case 4)	Yes	Yes	Yes	Yes	Yes

2. Results of the test

2.1. The raw results

Here after the table of measured loads when processing reference CAN trace. The load is the overall CPU load (100% when all CPU are fully busy) and for all processes.

SUBSCRIBER	PROCESS	PORTER	SALVATOR
No subscriber, receive only	Server	18%	4.75%
Same process	Server	25.5%	6.5%
Linked with Websocket	Server	37.5% } 70%	10.75% } 20.75 %
	Client	32.5%	10%
Linked with D-Bus	Server	OVERLOADED	14.5%
	Client		16% } 51.5%
	D-Bus		21%

2.2. The deduced results

From the data processing flow, we deduce load generated by each action as decribe in previous data processing flow diagram.

OPERATION	PORTER	SALVATOR
Receive	18%	4.75%
Create	7.5%	1.75%
Serialize + send	12% } 44.5%	4.25% } 14.25%
Receive + deserialize	32.5%	10%
DBUS[4]	?	30.75%

2.3. About results

2.3.1. Cost of transmission

The first operation, receiving raw CAN data. This initial receiving step already consumes a significant amount of CPU. In our testing scenario the reception is done using CAN/UDP to simulate a real device CAN bus acquisition. Cogent CAN sample trace payload is fixed at 8 bytes which might not be the case with a real CAN device.

4 123 = 84 + (58-43) + (64-40)

In fact when connected to a real CAN bus, we may expect a small decrease of this load. This because with a real CAN device ETHERNET+IP+UDP as well as netfiltering would not used.

Nevertheless and independantly of transmission mode, receiving such an amount of data consume ressources. On PORTER, receiving action represent already 1/5th of available CPU power, on SALVATOR it is better but nevertheless uses arround 1/20th of CPU ressources.

Tests done show that moving from binary CAN messages to JSON simple serialisation mode as currently used by AGL-2.0 framework increase the size of data by a ratio of 5.6; which obviously increase to a significant level the global cost requirer to move such a huge amount of data from one binder to an other. Total cost of serialisation/deserialisation + transmition on Porter represents about halt of avaliable CPU power (44.5%). On Salvador load is obviously much smaller but nevertheless remains significant at 14.25%.

2.3.2. No measure for D-Bus on PORTER

During D-Bus test, Porter board was overloaded and not all events were delivered by D-Bus that closes its connections when saturated.

2.3.3. No measure of interpreting the data

Current tests don't evaluates the cost of understand and processing CAN events/signals. We only evaluates the cost of events creation and transmission.

3. Conclusions and perspectives

3.1. Lessons learnt:

- SALVATOR is a significant improvement over PORTER
- D-BUS is not the right tool for transmitting such an amount of data
- The cost of AGL framework events model remains light when bindings share a common process, but increase when binding are in different processes.

On SALVATOR, when both receiving+consuming bindings share the same process the total load including receiving data represents 6.5% of total CPU power, which leave over 90% of CPUs to applications.

3.2. ToBeDone to improve performances:

They are different obvious places where event model performances could/should be improved.

Reduce the size of the transmited data and avoid conversion to/from string of numbers:

Current messages exchanged are somehow verbose and include redundancies. Improving these messages should be possible and would improve performances.

Current version of AGL-2.0 framework relies on a simple JSON serialisation. It exists more advanced format as BJSON[5], BSON[6], MessagePack[7] or CBOR[8] that could reduce significantly the size of transmitting messages.

Provide more efficient data handling:

AGL-2.0 relies on the library json-c to handle events and requests both internally and externally. This library has very good features but is not optimised for our usage. One option would be to improve library json-c and an other option would be to swich to an other library.

Use a faster transport mechanism:

Websocket is clearly faster than DBUS. Nevertheless some other technologies based on zero-copy model might be even faster and would deserve more investigations.

Lower the count of messages:

What ever object model will be used transmitting such a huge amount of data will consume a lot of ressources. The only valid option is to reduce the total amount of transmitted messages.

A model to implement filtering and processing of messages at the lower possible layer should be described and implemented. In best case this filtering should be delegated to hardware in worse case it should end to HAL layer.

5 http://bjson.org/
6 http://bsonspec.org/
7 http://msgpack.org/
8 http://cbor.io/ and https://tools.ietf.org/rfc/rfc7049.txt