Apertis Debug and Logging Design

Author:	Philip Withnall
Contributors:	Simon McVittie
Version:	0.2.0
Status:	Draft
Date:	2016-01-07
Last Reviewer:	Simon McVittie

This design was produced exclusively using free and open source software.

Please consider the environment before printing this document.

DOCUMENT CHANGE LOG

Version	Date	Changes
0.2.0	2016-01-07	- Add a few extra use cases for filtering whole system logs.
0.1.1	2015-12-18	- Clarify a few limitations.
0.1.0	2015-12-17	- New document to propose use cases.

Table of Contents

1 INTRODUCTION

This documents several approaches to debugging components of an Apertis system, either during development, or in the field. This includes debugging tools for reproducing and analysing problems; and logging systems for gathering data about problems and about system behaviour.

The major considerations with a debugging and logging system are:

- Reproducibility: Many of the hardest problems to diagnose are ones which are hard to reproduce. A set of debugging tools should make it easy to reproduce problems, and certainly should not make the problems disappear when being debugged.
- Timing: An important part of ensuring that problems are reproducible is ensuring that timing effects are reproducible, which means that a debugging system must have a low (almost zero) overhead, in order to avoid disturbing timing effects. Secondarily to this, it must allow the developer to see the order in which events occurred during the course of a problem.
- Context: As well as helping reproducibility of a problem, a debugging system should reduce the need to reproduce the problem in the first place — by capturing as much contextual information about it on the initial attempt at debugging.
- Confidentiality: Any system which logs information about a running system must ensure that the logged data remains confidential apart from to developers who need it for debugging. This may mean that logging is not enablable on production systems.

2 TERMINOLOGY AND CONCEPTS

2.1 APPLICATION BUNDLE

An *application bundle* is a group of functionally related components (services, data or programs) installed as a unit. This matches the sense with which ‘app’ is typically used on mobile platforms such as Android and iOS. (See the Applications design document for the full definition.)

2.2 COMPONENT

An application bundle or system service.

2.3 TRUSTED DEALER

An authorised vehicle dealer, garage or other sale or repair location which has a business relationship with the vehicle manufacturer.

3 USE CASES

A variety of use cases for scenarios where a component needs debugging, or where logging data are needed, are given below. Particularly important discussion points are highlighted at the bottom of each use case.

Some of these cases may be already solved in the Apertis distribution in its current state. However, they will all have an effect, to a greater or lesser extent, on this design.

3.1 DEBUG DETERMINISTIC APPLICATION ON SDK

An application developer needs to be able to debug their application when running it on the SDK, diagnosing crashes and looking at log output for that particular application.

3.2 DEBUG NON-DETERMINISTIC APPLICATION ON SDK

An application developer is working on an application whose behaviour appears non-deterministic (for example, due to using a lot of threads, or depending on sensitive timing). They manage to reproduce a particular bug only occasionally, but need to debug it further.

3.3 DEBUG APPLICATION ON TARGET

An application developer needs to be able to debug their application when running it on the target device (connected to an SDK machine during development), diagnosing crashes and looking at log output for that particular application.

3.4 DEBUG APPLICATION IN THE CONTEXT OF THE WHOLE SYSTEM

An application developer has a problem with their application which is dependent on the state of the whole (integrated) target system, rather than just on internal state in their application. They need to be able to correlate system state with their application's internal state.

3.5 EXTRACT LOGS FROM A DEVICE UNDER TEST

An Apertis tester has observed a failure in a development vehicle while doing field testing on it. They need to be able to extract logs from the vehicle after the event, and examine them offline to diagnose the failure.

3.6 TRUSTED DEALER CAN EXTRACT LOGS FROM A DEVICE POST-PRODUCTION

A vehicle owner has brought their vehicle into the garage with a failure in the IVI system. The trusted dealer at the garage extracts logs from the vehicle and passes them to the vehicle vendor for analysis, potentially leading to a fix for the problem in a subsequent release of the CE domain operating system for that vehicle.

3.7 THIRD PARTY CANNOT EXTRACT LOGS FROM A DEVICE POST-PRODUCTION

A vehicle owner likes to tinker with their vehicle, and would like to look at the logs which their trusted dealer can look at, in order to get more information about reverse engineering the IVI system in their vehicle.

They must not be able to access these logs.

3.8 LOGGING STORAGE SPACE IS LIMITED IN POST-PRODUCTION

On a production vehicle, the amount of storage space available for logging is limited, so the system should log only the most important or recent and relevant messages, and not write other messages to persistent storage.

3.9 RECORD AND REPLAY LOGS FOR INPUT TO AN APPLICATION

An application developer has found a problem in their application which depends on external input to it, and subtle timing sequences of that input. The input includes sensor input (from the SDK API, over D-Bus), and user interactions with the interface using the touchscreen and on-screen keyboard. This makes it a hard problem to reproduce. They want to add a regression test for it to their application, and want to automate it because reproducing the problem manually is too hard. This regression test needs to perfectly reproduce the problem each time it is run.

The application has more than one process (it has one or more agent processes, in addition to the main UI); all the processes communicate with each other at runtime.

3.10 RECORD AND REPLAY LOGS FOR SENSORS TO THE WHOLE SYSTEM

An Apertis tester wants to test the whole system against a variety of road trips, but it would be a waste of time to repeatedly drive a vehicle around a real road system in order to do repeat test runs. They want a replayable log file of all the sensor inputs from the vehicle, which can be replayed to the whole Apertis system on a development machine, to allow repeated testing of how the system responds to those inputs.

3.11 PERFORMANCE PROFILING

An application is performing poorly on the target device, and the developer wants to diagnose the problem so they can fix it.

3.12 DENIAL OF SERVICE ATTACK ON LOGGING

A misbehaving or malicious application is submitting log messages as fast as it can. This should not adversely affect system performance, or cause other log messages to be prematurely dropped.

4 NON-USE-CASES

4.1 RECORD AND REPLAY LOGS FOR ENTIRE SYSTEM BEHAVIOUR

While use case 3.10 is a legitimate use case, it becomes harder to record the *entire* system behaviour (as opposed to just the inputs from the sensor system), as that starts to be affected by differences in the components which are being tested if those components are changed to test new features. For example, if the entire system behaviour were recorded and replayed, it might not be possible to run a debugger on the system while replaying a log, as the debugger would impact the replay state too much.

5 REQUIREMENTS

5.1 CODE DEBUGGER INSTALLABLE ON DEVELOPMENT AND TARGET MACHINES

A code debugger must be available in Apertis, and installable on development and target machines so that it can be used by Apertis and application developers.

The tool must allow interactive walking through the stack, printing expressions, and other common C debugging functions.

See Debug deterministic application on SDK.

5.2 CODE DEBUGGER CAN BE USED REMOTELY

The code debugger must be usable remotely in real time, most likely with a server component running on the target device, and a client component on the developer's machine.

See Debug application on target.

5.3 CODE RECORD AND REPLAY TOOL INSTALLABLE ON DEVELOPMENT AND TARGET MACHINES

A code record and replay tool must be available in Apertis, and installable on development and target machines so that it can be used by Apertis and application developers.

The tool must allow recording all inputs to an Application from the kernel, plus any other system behaviour which would influence the application's behaviour. Those logs must be stored as files, and replayable many times.

When replaying logs, the developer must be able to use a debugger to investigate problems.

See Debug non-deterministic application on SDK, Debug application in the context of the whole system, Record and replay logs for input to an application.

5.4 APPLICATION LOGS AVAILABLE IN ECLIPSE WHEN RUN ON THE SDK

When developing an application in Eclipse, the logging calls the application uses must send their output to the Eclipse console (i.e. stdout or stderr) rather than (or as well as) the SDK system's journal. This allows the developer to easily read those messages.

See Debug deterministic application on SDK.

5.5 WHOLE SYSTEM LOGS ARE AGGREGATED AND TIMESTAMPED

All log messages from all system components and services must be directed to a central logging repository, which must timestamp them all in order (so that all the timestamps are directly comparable).

See Extract logs from a device under test, Debug application in the context of the whole system.

5.6 WHOLE SYSTEM LOGS ARE TAGGED BY PROCESS AND PRIORITY

All log messages from all system components and services must be tagged with the name of the process which generated them, and their priority (for example, 'debug' versus 'warning' versus 'error'). This metadata must be available to the developer to allow them to filter logs for relevant messages.

See Debug deterministic application on SDK, Debug application on target.

5.7 WHOLE SYSTEM LOGS ARE LIMITED BY PRIORITY AND ROTATED

On a production vehicle, the log messages which are written to persistent storage must be limited to only the most recent logs (according to some age cutoff) and the most important logs (according to some priority cutoff). These cutoffs must be configurable at production time.

It may be possible to keep all other log messages in memory while the vehicle is running, for example to allow them to be uploaded to an online diagnosis service in case of a fault. They must not, however, be written to disk.

See Logging storage space is limited in post-production.

5.8 EXTRACT WHOLE SYSTEM LOGS FROM TARGET DEVICE

The aggregated system log on a development target device must be accessible by the developer, who must be able to copy it to their development machine for analysis. The log does not necessarily have to be extractable in real time, though that would be helpful.

See Extract logs from a device under test.

5.9 EXTRACT WHOLE SYSTEM LOGS FROM TARGET DEVICE IN POST-PRODUCTION

The aggregated system log on a production target device must be extractable by a trusted dealer so that It can be sent to an Apertis developer for analysis. Extracting the log may require physical access to the vehicle.

See Trusted dealer can extract logs from a device post-production.

5.10 PROTECT ACCESS TO WHOLE SYSTEM LOGS ON PRODUCTION DEVICES

The aggregated system log on a production device must only be extractable by a trusted dealer or other authorised representative of the vehicle manufacturer.

See Third party cannot extract logs from a device post-production.

5.11 CODE RECORD AND REPLAY TOOL CAN HANDLE MULTIPLE PROCESSES

The code record and replay tool must be able to record and replay a single log for multiple processes, such as an application and its agents. They must all see the same timing information.

See Record and replay logs for input to an application.

5.12 RECORD AND REPLAY SDK SENSOR DATA

It must be possible to record all D-Bus traffic to and from the SDK sensors API for a given time period (a ‘trip’), and later replay that log to the whole system instead of using current sensor data.

See Record and replay logs for sensors to the whole system.

5.13 PROFILING TOOLS INSTALLABLE ON DEVELOPMENT AND TARGET MACHINES

A variety of profiling tools must be available in Apertis, and installable on development and target machines so that they can be used by Apertis and application developers.

See Performance profiling.

5.14 RATE LIMITING OF WHOLE SYSTEM LOGS

To prevent denial of service attacks on the system log, rate limiting must be applied to log message submissions from each application. If an application submits log messages at too high a rate, the extras must be dropped.

See Denial of service attack on logging.

6 EXISTING DEBUG AND LOGGING SYSTEMS

Open question: What existing debug and logging systems are relevant to do background research on?

7 APPROACH

Based on the above research (section 6) and requirements (section 5), we recommend the following approach as an initial sketch of a debug and logging system.

7.1 GDB AND GDBSERVER

For real-time debugging of applications, both on a local SDK system and on a remote target system, GDB should be used. For debugging remote systems, gdbserver should be set up on the remote system and GDB used as a client to control it.

They must both be available in the development repository, and hence installable on development and target devices.

7.2 RECORD AND REPLAY (RR)

For debugging of non-deterministic problems and problems which depend on context or state outside of the application, Mozilla’s Record and Replay (rr) tool should be used. It works by recording all input and output to a process (especially the input and output via kernel APIs), and allowing that log to be replayed while re-running the application. This eliminates all sources of non-determinism in the replay, ensuring that the conditions which triggered the original problem can be reproduced every time.

Crucially, rr works with D-Bus: as all socket input and output for an application is recorded, this includes all D-Bus traffic — this is reproduced faithfully in any re-runs of the application. As many of the Apertis SDK APIs are provided via D-Bus, this is a crucial feature.

In addition, rr can record a group of processes to a single log, and replay to the same group later on. This can be used for debugging an application together with its agents, for example.

Note, however, that rr is a *replay* tool and not an *interactive* debugger — a developer cannot replay a log recorded against one version of an application with a newer version of the application (for example, with changes which the developer hopes will fix the bug they’re investigating). This is because it would change the program’s output behaviour and hence its effects on external processes.

For example, consider a bug where a program is writing a network packet to the wrong socket out of two it has open. rr has recorded the response from the socket the program was originally sending to (the wrong socket) — when a fixed version of the program is run, the log file rr is using will not have a response stored for the second (correct) socket.

This must be available in the development repository, and hence installable on development and target devices.

7.3 SYSTEMD JOURNAL

All log output from processes on the target system should be sent to the systemd journal, allowing it to provide a single source of log data for the entire system, with all log messages in a single ordering. This includes debug messages, errors, warnings, and other log output. All messages should be sent with a priority level, plus additional metadata if relevant. The journal automatically adds the sending process' name to log entries.

When developing on a local SDK system, the log should be queried using the `journalctl` command line tool.

If a program is run manually from a console, or from within Eclipse, all log output must also be sent to stdout or stderr so that it appears on the console or the Eclipse console[1].

7.4 DIAGNOSTIC LOG AND TRACE

When testing a component on a target system, the developer should use diagnostic log and trace (DLT) from GENIVI — this is a client-server system where the DLT daemon runs on the target system and forwards systemd journal messages over the network to the developer's system, where they are presented in the DLT Viewer UI, which allows filtering, ordering, and other analysis to be performed on the logs.

However, DLT is only as useful as the log messages sent to it by the components on the system. Certain components may need to be modified to emit more log messages.

The DLT daemon exposes itself on the network and on the serial port with no authentication, so must not be installed by default on production systems.

7.5 EXTRACTING LOGS FROM A POST-PRODUCTION SYSTEM

For extracting logs from a post-production system, a new *journal export service* must be written which provides and authenticates access to the systemd journal.

This service would essentially run the `journalctl -o export` command to retrieve a full copy of the system's logs in a stable format suitable for sending to another system for review[2].

The service would need to listen on some external interface which a trusted dealer could connect to. This could, for example, be a network port; or it could be a physical connector on the IVI system's main board. In any case, the service must require authentication before exporting any logs.

Open question: What external interface can the journal export service listen on?

The authentication mechanism chosen depends partially on the characteristics of the interface the service listens on. It would most likely be a challenge–response protocol[3] issued by the journal export service, where the trusted dealer proves knowledge of a secret which has been issued by the vehicle manufacturer.

Open question: Should the logs be exported in an encrypted form, to keep them

[1] https://git.gnome.org/browse/libgsystem/tree/src/gsystem-log.c#n128

[2] http://www.freedesktop.org/wiki/Software/systemd/export/

[3] https://en.wikipedia.org/wiki/Challenge%E2%80%93response_authentication

confidential while being stored by a trusted dealer?

7.6 D-BUS MONITORING

As many of the Apertis SDK APIs are provided via D-Bus, an easy way to see what they're doing is to log all D-Bus traffic on the system and session buses. This can then be exposed by the DLT Viewer (or the local `journalctl` tool) and analysed.

A new *D-Bus logging service* (similar to the `dbus-monitor` tool, but presented as a systemd service which is enablable by developers, and only on development images) should be written which logs all traffic for a specified D-Bus bus to the systemd journal.

Note that this does not allow for log replay. For specific cases, this will be handled using the trip logger (section 7.7).

7.7 TRIP LOGGING OF SDK SENSOR DATA

In order to record 'trip logs' of the sensor data sent to and from the SDK sensor API and the entirety of the rest of the system, a *D-Bus record and replay tool* should be written. When recording, this could monitor the D-Bus session bus and record all traffic to and from the sensor API. When replaying, it would replace the SDK sensor service on the bus, and impersonate all its APIs, replaying responses from the log. This program must be aware of the semantics of D-Bus messages so, for example, it would not store the serial number of a message reply, but would instead use the serial number corresponding to the method call at the time of replay. Similarly, it must be aware of common D-Bus interfaces such as `org.freedesktop.DBus.Properties` and know that the value of a property remains unchanged unless a notification signal has been emitted for it.

One implementation option would be to implement this based on the `dbus-monitor` code: log all messages to or from the sensors API, and extract ones with known semantics, such as `org.freedesktop.Dbus.Properties` method calls and signals. The replay code would maintain a queue of pairs of (expected method call, reply), and for each incoming method call, would return and remove the first matching reply from the queue; or would return an error otherwise. For calls to known interfaces like `org.freedesktop.DBus.Properties`, the property state would be emulated with the correct semantics. Asynchronous events, such as signal emissions from the sensors API, would be emitted at the appropriate time relative to their surrounding events, rather than based on the absolute timestamp they were originally logged at. For example, if the log contained a signal emission after method call A and before method reply B, that signal would only be emitted in the replayed log after the program under test had made method call A.

An alternative implementation, which would be faster to implement but less generic and hence could not be repurposed for logging other SDK services in future, would be to use python-dbusmock[4] to build a specific mock service for the sensors API. This service would have full knowledge of the semantics of all the D-Bus messages it sent and received — the

[4] https://github.com/martinpitt/python-dbusmock

full sensors SDK API, rather than just the standard D-Bus interfaces. The log file would be generated similarly to in the first implementation — by monitoring and interpreting the D-Bus traffic for the sensors API. The file would contain an initial set of values for the properties of all the sensors, followed by timestamped updates to each value as it changed during logging.

7.7.1 EXAMPLE TRIP FILES

To give application developers some baseline situations to test against, it would be helpful if Apertis or OEM variants of it shipped with several example trip logs, demonstrating some common or uncommon driving situations which applications must handle.

Open question: Should example trip files be produced by Apertis, or by OEMs so they are specific to vehicles?

7.8 SECURITY

The security issues from logging are all concerned with confidentiality of system information, which may include sensitive data from a variety of processes.

This data must be kept confidential, both within the system (for example, applications must not have access to the logs of any process which is not in their trust domain), and from external attackers.

On production devices, especially, access to full system logs is a valuable goal for an attacker, as it gives insight into how the system is configured and further potential attack targets. For this reason, it may be worthwhile considering whether to reduce or disable logging on production systems.

Conversely, log entries from production devices are very useful for debugging unreproduceable post-production problems. Therefore, the choice of logging verbosity on production systems becomes a trade-off between the risk of confidentiality breaches, and the practicality of being able to debug problems.

Open question: What level of logging should be enabled for production systems versus development systems?

7.9 DISK USAGE AND PERFORMANCE

Storing log entries persistently consumes an unbounded amount of disk space. A limit must be applied to the number or age of log entries which are stored before being dropped. The systemd journal must have a disk space or age limit applied; this can be done by editing `/etc/systemd/journald.conf` and adding the following, for example:

```
SystemMaxUse=100M
```

To limit the priority level of messages which are stored to disk, the following configuration option can be used; it is highly recommended to set it to ‘debug’ on development systems and ‘error’ for production systems[5].

```
MaxLevelStore=error
```

Logging must not have a large runtime overhead — each call from a process to the logging API must be fast. Furthermore, rate limiting must be applied to prevent a misbehaving application from overfilling the system logs. This can be achieved using the following configuration options for the systemd journal; the following values limit each process to at most 1000 messages in a given 30 seconds:

```
RateLimitInterval=30s
RateLimitBurst=1000
```

7.10 PROFILING TOOLS

A variety of profiling tools should be packaged for the Apertis development repository:

- perf
- valgrind
- google-perftools
- strace
- ltrace
- systemtap
- gprof

7.11 SUGGESTED ROADMAP

GDB and DLT are already packaged, so no further work is needed there; as are all the profiling tools.

rr is not yet packaged, but should be.

Integration of everything into the systemd journal, plus adding additional debug messages to various system services to improve debuggability of those services.

The journal export service, D-Bus logging service and D-Bus record and replay tools are all self-contained, so could be produced individually as later stages in the implementation.

7.12 REQUIREMENTS

- 5.1, Code debugger installable on development and target machines: GDB is the debugger.

[5] The full range of options is documented in `man 5 journald.conf`.

- 5.2, Code debugger can be used remotely: GDB can be used with gdbserver.
- 5.3, Code record and replay tool installable on development and target machines: rr is the record and replay tool.
- 5.4, Application logs available in Eclipse when run on the SDK: Outputting log entries to stdout or stderr if running on a console.
- 5.5, Whole system logs are aggregated and timestamped: All system logs are forwarded to the systemd journal. D-Bus messages are logged to the journal via a new D-Bus logging service.
- 5.6, Whole system logs are tagged by process and priority: Done by the systemd journal by default.
- 5.7, Whole system logs are limited by priority and rotated: Done with suitable configuration of the systemd journal.
- 5.8, Extract whole system logs from target device: DLT is used to extract logs and transfer them to a developer machine in real time.
- 5.9, Extract whole system logs from target device in post-production: New journal export service exposing an authenticated interface for exporting systemd journal logs.
- 5.10, Protect access to whole system logs on production devices: Journal export service requires authentication.
- 5.11, Code record and replay tool can handle multiple processes: rr supports logging and replaying to multiple processes.
- 5.12, Record and replay SDK sensor data: D-Bus record and replay tool will be used for this.
- 5.13, Profiling tools installable on development and target machines: Various profiling tools will be packaged.

8 OPEN QUESTIONS

- 6: What existing debug and logging systems are relevant to do background research on?
- 7.5: What external interface can the journal export service listen on?
- 7.5: Should the logs be exported in an encrypted form, to keep them confidential while being stored by a trusted dealer?
- 7.7.1: Should example trip files be produced by Apertis, or by OEMs so they are specific to vehicles?
- 7.8: What level of logging should be enabled for production systems versus development systems?

9 SUMMARY OF RECOMMENDATIONS

As discussed in the above sections, we recommend:

- Packaging Mozilla’s Record and Replay (rr) tool for the development repository.
- Ensure that all system components and services are logging exclusively to the systemd journal.
- Configure the systemd journal to handle log expiry, rotation and priority storage levels to avoid consuming unbounded disk space.
- Potentially add more debug log messages to various system services to give more context when debugging applications.
- Write a journal export service for exporting the systemd journal with authentication from a production system.
- Write a D-Bus logging service for logging all D-Bus traffic to the systemd journal to give more context when debugging applications.
- Write a D-Bus record and replay tool for producing trip logs from the SDK sensor API.
- Audit the confidentiality of the systemd journal and ensure it is only accessible to developers and the journal export service.