Libpanda - A High Performance Library for Vehicle Data Collection

Matthew Bunting
mosfet@email.arizona.edu
University of Arizona
Tucson, Arizona, USA

Rahul Bhadani
rahulbhadani@email.arizona.edu
University of Arizona
Tucson, AZ, USA

Jonathan Sprinkle
sprinkjm@email.arizona.edu
University of Arizona
Tucson, AZ, USA

Abstract

Cyber-Physical Systems (CPS) generally involve time-critical components due to physical dynamics, therefore necessitating high-performance subsystems. This is also true in data collection scenarios to infer physical phenomena. This paper covers Libpanda as an example of a component that has been designed to address performance issues in CPS implementations. Libpanda is a C++ library that interfaces software with a Comma.ai Panda device. Pandas are used for installation in modern vehicles to read the vehicle CAN bus, providing rich sensor data and limited vehicle control through message injection. The motivation to design libpanda stems from the lack of performance in Python-based code that runs on inexpensive hardware like a Raspberry Pi. In such situations, Python code would result in utilizing 92% CPU while also dropping around 40% of the CAN packet due to bottlenecks. Without using different tools, inconsistent data collection means a loss of time-based vehicle state interpretation. Libpanda addresses these issues through implementation in a different language and implementation of different design paradigms involving asynchronous calls and multithreading. The Panda also features a GPS module that allows multiple instances to synchronize clocks for large-scale data collection scenarios. Libpanda has been designed with time-synchronization in mind to aid in the measurement of inter-vehicle dynamics. The performance improvements of libpanda have resulted in it becoming an important component in automotive dynamics research that requires a higher technical performance in large-scale experiments.

CCS Concepts: • **Computer systems organization → Embedded software**; Reliability; • **Networks → Cyber-physical networks**.

DICPS '21, May 18, 2021, Nashville, TN, USA

ACM ISBN 978-1-4503-8445-2/21/05…$15.00
https://doi.org/10.1145/3459609.3460529

Keywords: Cyber-Physical Systems, Software Libraries, Data Collection

ACM Reference Format:
Matthew Bunting, Rahul Bhadani, and Jonathan Sprinkle. 2021. Libpanda - A High Performance Library for Vehicle Data Collection. In *The Workshop on Data-Driven and Intelligent Cyber-Physical Systems (DICPS '21), May 18, 2021, Nashville, TN, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3459609.3460529

1 Introduction

Many vehicle manufacturers are on the stepping stones in integrating the infrastructure needed to have full self-driving functionality. Cruise control is perhaps one of the earliest integrations of this automation, letting the vehicle operate one of the driving aspects for the driver. Engine Control Units (ECUs) have been integrated to optimize engine performance and emissions and to share diagnostic information with mechanics. Since 1996, the United States has required the implementation of OBD-II (On-Board Diagnostics) for ensuring emissions standards are met by vehicles [13].

One of the electrical protocols that have become a standard in vehicles is the Controller Area Network (CAN) specification [7]. Devices communicate over a bus using frames that contain an address, timestamp, and data. This lets many of the micro-controller-based devices in a vehicle communicate using minimal wiring through sharing a single bus. CAN is used as part of the OBD-II interface and has been enabling the full drive-by-wire design of vehicles [1]. Though initially the CAN bus may have only been used for ECUs and emissions tracking, CAN buses are now used for things like turn signals and radio functions. Implementation of other advanced behaviors like Adaptive Cruise Control (ACC) has also made use of CAN to listen to radar proximity data and send control commands to throttle and braking mechanisms [20]. This has also been used for Lane Keep Assist (LKA) units where camera data can be used to operate steering torque [5], again using a CAN bus for device communication.

Since many modern vehicles are using a standard electrical protocol and are designing mechanical systems to be drive-by-wire, it is possible to tap into a vehicle's infrastructure and either collect CAN data or even inject other CAN messages to control the vehicle with custom software. Most likely due to safety concerns and trade secrets, information

on system protocols is generally not formally disclosed. However some open-source groups and companies have decoded messaging protocols, democratizing knowledge about CAN networks. These groups may intend to use the information to manufacture devices that can expand on the capabilities of LKA and ACC so that cars can fully self-drive. Such devices are only possible based on the state of modern cars and the built-in rich sensor data. Off-the-shelf devices, such as those provided by Comma.ai, can also be used in other use-cases, like data collection from in-vehicle experiments.

2 Background

Cyber-Physical Systems have an intrinsic problem in regards to gathering reliable data. The physical component typically involves continuous time domains whereas the cyber component is discrete. The aspects of time regarding data collection can be critical to fully understand the physical phenomena. This is especially true in distributed systems where one system's physical process may affect the physical process of another system. Three important qualities of time can greatly affect the quality of CPS data: rate, timeliness, and synchronization.

1. The rate at which data are captured should be sufficiently dense to observe the phenomena for which one is looking to validate or control. This is important for performing post-processing like derivatives to understand the dynamics of data.
2. The timestamps of data should be consistent between distributed objects within control. This is important to understand timing characteristics like delays between different physical processes that recorded data in separate distributed systems [8] [18].
3. The timeliness of data capture must be certain to do integration or derivatives of data for observer-based analysis or control or timeliness in health data [21]. [3] [10].

In the context of libpanda, data are intended to be collected to understand the physical dynamics of vehicle-to-vehicle (V2V) interactions based on human driving dynamics, eventually leading to controller design. Data need to be sufficiently well-timed to design safe controllers. Well-timed data in V2V systems is certainly not a new idea where efforts have been done to ensure timely data [4]. In some cases, it has been found that 10Hz is required for the vehicle to be aware of cooperative agents in platooning scenarios [2]. In other scenarios, the vehicle needs a data-rate of 50Hz for safety control involving collision detection [12]. Poor response time and delays can cause system instabilities in V2V platooning [9].

One example system that is also an inspiration for this work is from an experiment held in 2017 involving 22 cars on a ring road to observe the phenomena of traffic waves [17]. This experiment was set up to record velocities and fuel consumption of all 22 cars, observing their inter-vehicle dynamics. One car named the CAT Vehicle has been outfitted with special control capabilities and high data-rate recording, along with an OBD-II interface recorder. The remaining 21 of the cars (not modified for control) were also outfitted with an OBD-II style interface based on the ELM-327, a common OBD-II interpreter [16] [11]. The OBD-II readers measured fuel consumption and velocity using a small computer in each vehicle. With this setup it seems that data collection and analysis would be trivial: however, the lack of confidence in data timing resulted in the need for further effort.

- The 22-car experiment data were saved from all the vehicles but significant post-processing was required to cross-correlate signal streams and to align the timestamps.
- Additional cameras were required to obtain high-frequency data on velocities, which were used to observe the emergent traffic waves.
- The CAT Vehicle provided ground-truth information on the high-frequency velocity estimates which could be used to align the OBD-II recorder on-board the CAT Vehicle with its velocities, and in turn, align the velocity observed by the camera system to the CAT Vehicle's onboard data collection. See Figure 1.
- Thus while it was possible to extrapolate information from all the vehicles in the experiment, the amount of time required for post-processing and custom projection into the experiment time-frame far exceeded the time taken during the experiment, making it unlikely to be used for larger-scale experiments that cannot fit onto a ring road.

The challenge, therefore, is the following: how to record meaningful data with existing off-the-shelf technology that would provide many of the needed correlations on its own? We describe in the following section what limitations still exist when it comes to being able to analyze the data streams at multiple timescales as required in CPS applications.

3 The Comma.ai Panda

Comma.ai sells multiple flavors of Panda with the intended use case of connecting with their EON lineup. The EON is a smartphone that has been modified by installing a Linux-based operating system that can run the custom self-driving software designed by Comma.ai. The phone however does not have the hardware capabilities to interface with vehicles, hence the use of the Panda to interface CAN and power buses on the car through the USB data and charging port on the phone. Through this USB interface, power can be back-fed to the phone for charging while the phone acts as a USB host for the CAN message reading and writing.

The EON is intended to be a complete package for users to add self-driving functionality to their cars. Fortunately, the open-source nature of the entire hardware and software

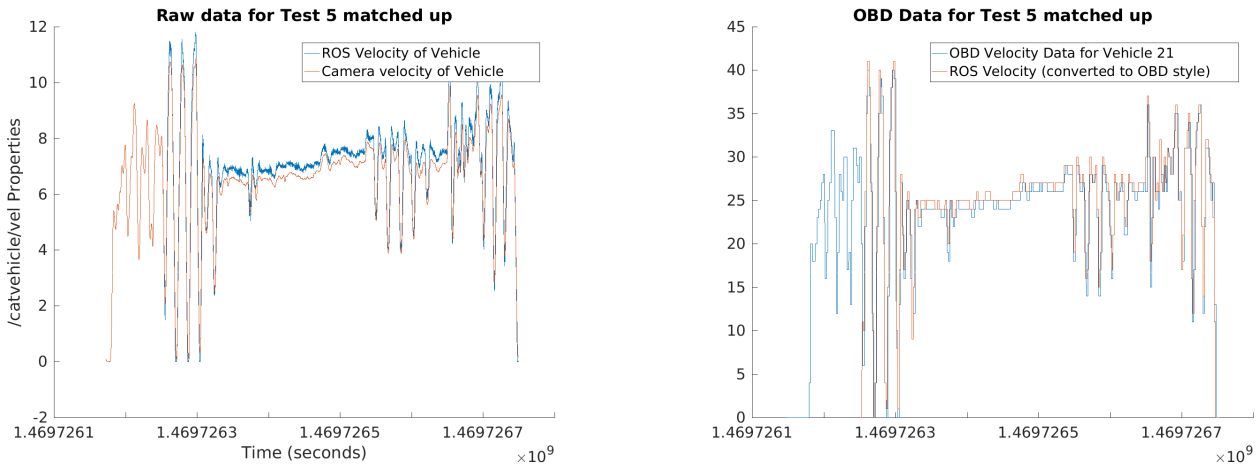

Figure 1. Left: Velocity estimates gathered from high-frequency observational cameras provided velocity estimates that could be compared to ground-truth information from the CAT Vehicle, which validated those measurements (note: the velocity of the CAT Vehicle is proportionally faster than the camera estimate, as it drove in a slightly wider arc than the other vehicles for experiment safety). Right: OBD-II data was obtained at a much lower frequency, and required significant post-processing to account for time synchronization. In order to obtain results for just a few hours of driving of the seminal experiment, weeks of data processing had to be performed, based on shared velocity estimates from the ground truth data as observed by the cameras and the CAT Vehicle.

stack leads to the ability to use components for other use-cases. This includes the Pandas, which can be leveraged for interfacing vehicles with custom control algorithms and/or data gathering software. There are three types of Panda available: White, Grey, and Black. All of these devices have the same feature sets regarding communication with CAN busses but have different connectors and other features. At the time of writing, the Grey Panda is being phased out of production since it features a similar function set to the Black Panda.

1. White Panda: Can connect over WiFi as well as USB, has no GPS. Uses an OBD-II connector and must be coupled with a Comma.ai Giraffe for connecting to CAN buses other than the vehicle's OBD-II port through parts of the vehicle's wiring harness.
2. Black Panda: A smaller device that connects to both the OBD-II port as well as a type (based on vehicle model/year) of Comma.ai's Car Harness. Has no WiFi and instead has GPS.
3. Grey Panda: Similar to the White Panda in requiring a Giraffe for car wire harness connection, but also similar to the Black Panda in having no WiFi in favor of a GPS.

All three devices use the same fundamentals for connection to the CAN bus, from the USB handling to the functional electrical connections in the car. For our purposes and eventual use cases the WiFi functionality was not needed due to the safety-critical nature of using custom controls. Also, the GPS functionality adds another useful data point and adds GPS time-synchronization. Therefore even though libpanda may still be able to use the CAN-bus functionality on the Panda White, only the Grey and Black Pandas are directly supported. Also, even though the Grey Panda is technically functional, only the Black Panda will be discussed due to future reduced product availability.

Figure 2 Shows how a Panda connects between a computer (Raspberry Pi) and the vehicle harness and OBD-II port. The Pandas are incapable of connecting to a vehicle's wire harness without the use of either the Giraffe (White/Grey) or Comma.ai's Car Harness (Black). Each of these is tailored for different vehicle makes since they typically use different electrical connectors on their CAN-based modules. As a small note regarding the usage of the word "fake" in the figure: though a USB-C and RJ-45 cable, are used to connect Comma.ai devices they do not abide by USB nor Ethernet electrical protocols and are simply means of connecting custom electrical signals. Also, the Panda Black is intended to act as a client USB device while providing power to the EON, back-feeding power to the host. **To safely connect this to a normal computer, the power lines must be severed to avoid hardware damage.**

4 Panda Hardware/Firmware

The Panda is capable of reading from the CAN bus, as well as sending its own messages. One of the use cases for the Panda is to operate as an alternative implementation of Adaptive Cruise Control (ACC). There is a problem, however, if a vehicle's native devices are sending their commands as well: then

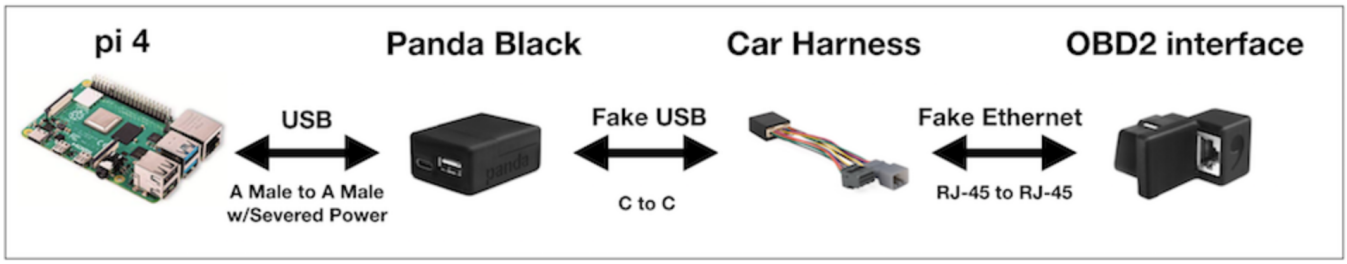

Figure 2. Connections of the Black Panda.

there will be message conflicts and potential race conditions if the Panda also sends commands. This has the potential to cause errors in vehicle subsystems. To circumvent this issue the Panda has been designed to be installed in between the CAN bus and particular modules in the vehicle that sends the commands of interest (acceleration/steering commands). This lets the Panda block commands from being sent to prevent conflicts and to pass through all other commands. Thus the Panda is capable of reading, injecting, and blocking messages. In pure data collection scenarios, all messages will be passed through.

This method of CAN message interception is defined by firmware in Panda's micro-controller. In other words, this is not something that can be customized dynamically by host software using the Panda as a USB device. It is possible to create new firmware and is something necessary if new supported vehicles come out with different command structures or change message IDs. Currently, the intended use of the Panda is to be used only on vehicles supported by Comma.ai. Since the Panda is intended to be general-purpose for any of the supported vehicles, the firmware is designed to be configurable by the host device. If the host knows the installed vehicle model and makes then the host is responsible for informing the Panda of the connected vehicle type. The Comma.ai EON goes through a fingerprinting process to determine the vehicle type by observing the CAN message IDs and lengths. The setting for the Panda vehicle type is called "safety mode" since it also does low-level safety checking to disengage autonomous driving based on the driver's vehicle interaction. Setting the Panda Safety mode, therefore, accomplishes a few different tasks based on the vehicle's state.

- Listens for the cruise control state by reading the driver's inputs for ACC operation.
- Intercepts cruise control messages. If the Panda detects a host computer is connected and wants to send commands, then messages corresponding to ACC and LKA commands are blocked—permitting the computer to inject these commands.
- Checks the values that the car reports for steering torque and pedal presses. If pedals are pressed, then control is relinquished to the operator.
- Times out for safety. If host control is active and any command is sent slower than 1Hz, then control is also relinquished due to a safety timeout.

In the case of pure data collection, the Panda safety mode does not need to be set. Regardless of the safety mode setting, all CAN information is shared with the host device. One more detail to note is that vehicle control has no impact on data collection, meaning that all data can be collected even if messages are blocked.

5 USB Interfacing

Libusb is a library commonly used to interface with USB hardware devices [6] and is suitable for Panda communication. There exist variants in both C/C++ and Python. The Panda does exhibit a mode to emulate an ELM327 - a commonly used OBD-II to USB integrated circuit and thus ELM327 data recording software can be utilized. However to fully use the capabilities of the Panda regarding GPS and CAN message interception, particular messages need to be sent by a custom hardware driver. Libusb lets a developer send commands needed by the Panda to set particular safety modes and read from the GPS.

The general USB protocol features four different methods of data transfer, however, only some are used by the Panda:

1. Interrupt: Used to signal immediate data transmissions. Unimplemented by the Panda.
2. Isochronous: Used for high data rates. Unimplemented by the Panda
3. Bulk: Used exclusively for CAN bus communication by the Panda
4. Control: Used to configure the Panda (e.g. safety mode) as well as read GPS information

6 Python USB interfacing

Examples provided by Comma.ai involve the implementation of libusb in Python to read data from the panda. To check the feasibility of the comma.ai Panda, a minimal example in Python was created for data-collection scenarios. This example continuously polls the Panda device CAN bus buffers and writes the data in a CSV format to file. This is based on libusb bulk transfer calls.

Using this code on Raspberry Pi 4 the Python process exhibited 92% CPU usage. Figure 3 shows initial data collection from a single CAN message with an expected rate of 50Hz. By plotting the differences of time between packets over time, evidence of missed packets can be seen. Unfortunately, these results implied that custom controllers would have no trust in the safe operation of the vehicle if radar data was missed. Even in data collection scenarios, loss of data could mean a loss of important data interpretation. The performance of the Python solution can be improved by the use of a higher performance computer like a laptop however this is not a scalable solution. Since this was an issue of software performance there was clear motivation to explore enhancements so that data can be collected in large-scale experiments. This led to the exploration of optimization through both multithreading and a faster performing language.

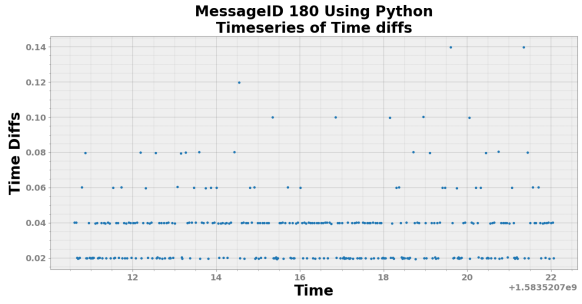

Figure 3. Time differences of CAN Bus Messages with ID 180 using Python-USB interface. At an expected regular rate of 50Hz, the resulting differences around quantized increments of 0.02 seconds away from 0.02s suggest packet loss.

7 Libpanda

Libpanda is designed to be a C++ library intended for a variety of use cases. As implied by the name, libpanda is a library for interacting with Comma.ai Panda devices through USB protocol abstraction, providing data of interest from the CAN bus and GPS as well as device control. Libpanda can be used for data collection or vehicle control and be integrated into utilities, service scripts, and the Robotics Operating System (ROS). Since libpanda stemmed from the lack of performance of Python running on a Raspberry Pi, this library is tested and tuned based on these platforms.

7.1 Threading

The single-threaded Python implementation resulting in 92% CPU utilization from the top on a 4-core Raspberry Pi 4. This suggests that only one thread is using an entire core and that more processing power may be available if needed. This is likely the reason that CAN packets were dropped at a 40% rate. Restructuring the polling methodologies of libusb could result in multi-core usage and reduce packet drop rate. To facilitate multithreading, a library called libmogi was partially used for its pthread-based abstraction for C++ [14]. This library was chosen purely by the developer's familiarity with the library. Within the C++ namespace of Mogi exists class Thread.

Mogi::Threads are abstract classes that use terminology from UML Statecharts, specifically regarding the virtual functions of entryAction(), doAction(), and exitAction(). Upon starting a thread, entryAction() is called if any thread initialization is needed. doAction() is then called consistently until the thread is told to be stopped. Once the thread has been stopped and doAction completes, exitAction() is called for any cleanup if needed.

A concrete implementation of Mogi::Thread then can make use of thread control functions start(), pause(), resume(), stop(), lock(), and unlock(). Usage of lock() and unlock() are based on pthread mutual exclusion to prevent race conditions. Calling start() will start the thread, causing entryAction() to be called once followed by infinite calls to doAction(). Calling pause() will temporarily prevent further calls of doAction(), until resume() is called. Calling stop() will stop calls of doAction() then call exitAction() before stopping the thread.

The design of Mogi::Thread is simple and sufficient for usage in the design for libpanda. This threading methodology is used for handling USB as well as parsing CAN and GPS data for data gathering.

7.2 Observer Design Pattern

The observer design pattern is used for sending immediate notification messages to objects from an associated parent object [15]. A class defined with an observer style pattern holds a list of observer objects, invoking methods within the observers so that the observers may handle the message as needed. This is effectively an object-oriented method of callback functions. Due to the safety-critical nature of a CPS, the use case of libpanda, immediate notifications help mitigate issues of latency. Without a callback methodology, a thread would need to constantly poll another object until data is ready which would increase the CPU usage of each thread. Observer patterns are used for USB events to notify the handler of GPS and CAN handler objects which are also notifiers that can distribute GPS and CAN message to further observers.

By employing C++, a std::vector<> is used to contain a list of observers allowing for multiple observers to be attached to a notifier. Classes that wish to be notified need to inherit from an observer parent class. Notifications occur through a virtual method defined in the parent class. This means that base classes can be designed to be highly cohesive. For example, one class can be designed for GPS data logging while another can be designed for synchronizing the system clock and be within the same vector of observers. Currently, there does not exist any priority mechanism, and the order

in which observers are added is the same order that they will be executed on each notification. A caveat to using this notification scheme is that a person designing a base class needs to be fast with the data so that other processes are not blocked. This can be mitigated by implementing a queue type of method and have a separate thread for data consumption based on the queue.

7.3 Asynchronous USB Handling

The prior implementations of writing to CSV files in Python involved the procedure of requesting data, then reading data, then writing formatted data to the file before repeating. This effectively causes a potential bottleneck in requesting new data while waiting to save data. To mitigate this issue an asynchronous method was integrated to allow for simultaneous libusb requests and CSV logging. To check against synchronous methods, libpanda is designed to be configurable as synchronous or asynchronous. A class named Panda::Usb has been designed to manage the methods of asynchronous or synchronous calls.

The Python implementation was purely synchronous through the libusb calls of libusb_bulk_transfer() and the companion method libusb_control_transfer(). This is a blocking call that can be terminated based on a set timeout. In order to perform an asynchronous bulk transfer, basic libusb control and bulk transfer functions must be avoided. For implementation in libpanda, a modified version of libusb_bulk_transfer() and libusb_control_transfer() has been created: asyncBulkTransfer() and asyncControlTransfer(). These functions are based on the source of libusb_bulk_transfer and libusb_control_transfer, respectively, however, they have been truncated to avoid waiting for the submitted transfer to be finished. Libusb is expected to perform a callback function with user data on completion, so a static member function is provided and is expected to receive a Panda::Usb object for handling the libusb events.

Panda::Usb inherits from Mogi::Thread such that doAction() regularly performs libusb_handle_events_timeout(), which handles the submitted transfers and invokes callback functions as needed. The callback function for bulk and control transfers provided to asyncBulkTransfer() and asyncControlTransfer() checks the status of the submitted transfer. If libusb reports that the data is valid and complete, then the list of transfer observers is notified.

In order to allow both synchronous and asynchronous methodologies without a large restructuring, calls to libusb and the asynchronous transfer methods are abstracted into methods called requestUartData() and requestCanData(). Either of these is called based on a configurable option in Panda::Usb. The asynchronous pipeline has been described above regarding the callback methodology with observers. The synchronous method undergoes a similar pipeline, where requestUartData() and a requestCanData() do not return any data since data is is passed through observers. The synchronous configuring simply notifies the observers immediately after performing a libusb_bulk_transfer() or libusb_control_transfer(). This reduces the observer framework into a set of procedural calls that occur during a call of requestCanData() or requestUartData().

A final note on the threading implementation and synchronous calls is that threading is only needed for the asynchronous configuration of Panda::Usb. This is because synchronous calls already invoke libusb_handle_events() and therefore do not need to be handled externally. In doAction(), the thread will be automatically paused if the mode is set to be synchronous. This method of implementation results in using identical pipelines for both asynchronous and synchronous configurations without needing refactoring on a per-use-case basis.

7.4 CAN Message Handling

A class named Panda::Can is integrated to manage the handling of CAN message reading and parsing. This class uses the Panda::Usb to invoke regular bulk read requests at the proper USB endpoints. The panda does not directly transmit raw data from the CAN bus, instead, it decodes a CAN frame into data of interest like the message ID, bus, time, and data. Panda::Usb takes this stream of data and parses it into a Panda::CanFrame for convenience. Panda::Can works with Panda::Usb by subscribing itself as a CAN data observer. Upon each new message notification, Panda::Can further applies the observer pattern to notify observers of new Panda::CanFrames. This structure is intended for future integration of libpanda in middleware like ROS.

Each CAN message fits into 16 bytes of data. Through experimentation, it was found that bulk read requests will only be successful if the read request is at a minimum of 64 total bytes, or a minimum of 4 total CAN packets. It is possible to read more packets in a single read but this introduces a lack of data timeliness.

Since the Panda does not provide a feature to check if data is ready to be read, it is up to the host device to regularly attempt data reads. These regular intervals may vary for different vehicles depending on the message rates on the bus. For a Toyota RAV4, we found regular polling intervals of 500us to be sufficient.

Panda::Can also features methods of convenience for data logging. End-user code can set logfile names to enable logging both raw data directly from the bulk transfers to file, as well as a CSV-formatted variant for easier plotting. Part of the CSV-formatted data also includes system time instead of the CAN bus time to understand the true time of arrival of data. To avoid similar issues to that of the Python variant by blocking further CAN read, data is sent to a queue to prevent blocking the Panda::Usb threads and ensure timeliness of data while formatting data for files.

Table 1. Python vs libpanda

	Python	libpanda
CPU Usage	92%	35%
10s File Size	1.3 MB	2.2 MB

Table 2. Comparisons between running top and measuring file size over the same time period in the same Toyota RAV4

7.5 GPS Message Handling

GPS is handled in a very similar manner to CAN messages, through a class named Panda::Gps. The design is nearly identical with subtle differences due to requiring USB control transfers instead of bulk transfers. The GPS module in the Panda is only capable of sending data at a maximum of 10Hz so the polling interval has been set at 100Hz to ensure data timeliness while not causing a high thread CPU load. The GPS module reports NMEA formatted strings, so just like Panda::Can messages are parsed for convenience using a library name NMEAParser [19]. Fully parsed NMEA strings are then sent out as notifications to any GPS observer if needed. As before, data can also be logged as raw NMEA strings or as a CSV-formatted file containing system time, GPS time, the GPS module state, quality, pose, motion, and satellite information.

An additional class named Panda::SetSystemTimeObserver can be attached to a Panda::Gps as an observer. This is a convenience class intended to synchronize the system's time based on GPS time. The original intent of this class is to set a Raspberry Pi's clock on startup since a battery-based Real-Time Clock (RTC) is not included. This can be useful in experiments where multiple cars are needed in the same experiment and data needs to be compared between multiple systems, just like in the 22-car experiment as shown in Figure 1.

8 Comparisons

Table 2 shows the results of a test performed in the same vehicle that remained running during each test. The vehicle is Toyota RAV4 equipped with a RADAR for ACC. The computer for data collection is a Raspberry Pi 4. The first test shows the CPU usage measured using the "top" command. The implementation of libpanda shows a significant improvement over the Python code, running at 35% versus 92% CPU usage. Libpanda in this case was also recording GPS data while the Python version had not yet integrated this feature. GPS data is only configurable to a maximum of 10Hz so it may not impact CPU much, regardless asking more from the Python code may result in more dropped packets.

During the initial development of libpanda, it should be noted that the use of threads and the observer design pattern were not integrated for initial tests. We were unsure if the packet dropping issues were a result of the Panda or the Python code, so a quick C++ port of the Python code was first built to check for the feasibility of the language change. The first test of C++ code, which was functionally identical to the Python version, exhibited 50% CPU usage with no packet loss. This was a clear sign that moving to C++ was effective and what triggered the design efforts of libpanda.

Evidence of dropped packets was shown earlier in Figure 3, however, this phenomenon is also observable through observing file size growth over time. By running each version of data collection for a set amount of time, the expected file size from each session should be very close. Table 2 also shows the output file sizes of libpanda and Python after running each for approximately 10 seconds. This data suggest that Python was only able to record 59% of the data relative to libpanda, a significant improvement. Though improved, this is not evidence for not dropping packets.

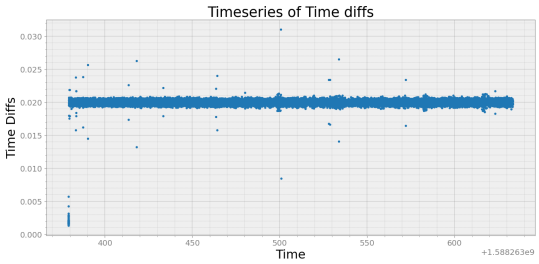

Figure 4. Time diffs of an expected 50Hz message using libpanda showing a significant improvement over the Python implementation shown in figure 3

Evidence for an improvement in the reduction of packet loss can be seen in Figure 4. This signal is the same 50Hz message ID recorded in Figure 3. As can be seen, the messages are close to a time difference of 0.020 seconds compared to the jumps of up to 0.120 seconds from the Python version. This implies that during this recording session, no packets were missed since a single missed packet would exceed a time difference of 0.040s which is not seen in this figure. There are however a couple of items that should be noted regarding this data. First, the current implementation of libpanda results in a set of buffered data on the Panda at the start of a recording session that gets read nearly all at once. This is the anomaly seen at the very beginning with data arriving with a time difference of 0.002-0.006 seconds. Implementing a better initialization procedure would eliminate this. The second note is in regard to the very clear outliers that stray away from an expected 0.020s time difference. The majority of these outliers are symmetric outlier about the average time difference of 0.020s. Given regular message intervals, this suggests that at times when a message is read too late the next message will be read much earlier. These outliers

could be the result of many things, either from the implementation of libpanda, a process on the Raspberry Pi causing delays in libpanda execution time, the minimum of 4 packets per bulk transfer, or the CAN bus on the vehicle being busy with other messages. Approximately 95% of the data is very close to the expectation, falling between 0.019s-0.021s, or 47.6Hz-52.6Hz.

9 Conclusion

Libpanda has been shown to address the initial issues with the simple Python-based code. Along with addressing packet loss and CPU utilization libpanda also has included methods of recording GPS data in addition to CAN data and has introduced methods to synchronize clocks in multiple vehicle experiments. This provides data that requires less post-processing to align data, ensuring proper data interpretation. By addressing these issues, usage of the Comma.ai Panda is possible with low-cost hardware like the Raspberry Pi for data collection, enabling scaling experiments to many vehicles with less of an impact on the budget.

Further work can be done to further optimize the performance and cost of the system. The Panda device currently only allows for minimum block sizes of four messages to be sent, introducing small delays in the first messages stored in the queue. The panda also must be regularly polled inefficiently to check if data is available. Firmware changes could alleviate these shortcomings, however, in their current state, they are very workable. The current solution involves a set of off-the-shelf hardware that is very flexible, eventually cost could be reduced further by making a hardware solution that accomplishes the functions of both the Raspberry Pi and Panda for data collection.

10 Acknowledgement

This work is supported by the National Science Foundation under awards CNS-1544395. This material is based upon work supported by the U.S. Department of Energy's Office of Energy Efficiency and Renewable Energy (EERE) under the Vehicle Technologies Office award number CIDDE-EE0008872. The views expressed herein do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

The authors would like the thank George Gunter for the initial research in CAN-bus devices, leading to the discovery of the Panda devices. We would also like the thank Matthew Nice for his feedback in using libpanda for data collection to further refine libpanda. We also thank Safwan Elmadani for integrating libpanda into ROS to help improve libpanda as an API.

References

[1] M Bertoluzzo, Paolo Bolognesi, Ottorino Bruno, G Buja, Alberto Landi, and A Zuccollo. 2004. Drive-by-wire systems for ground vehicles. In *2004 IEEE International Symposium on Industrial Electronics*, Vol. 1. IEEE, 711–716.

[2] Mate Boban and Pedro M d'Orey. 2016. Exploring the practical limits of cooperative awareness in vehicular communications. *IEEE Transactions on Vehicular Technology* 65, 6 (2016), 3904–3916.

[3] Simin Cai, Barbara Gallina, Dag Nyström, and Cristina Seceleanu. 2016. Towards the verification of temporal data consistency in Real-Time Data Management. In *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*. IEEE, 1–6.

[4] Byeong-Moon Cho, Min-Seong Jang, and Kyung-Joon Park. 2020. Channel-aware congestion control in vehicular cyber-physical systems. *IEEE Access* 8 (2020), 73193–73203.

[5] Andreas Eidehall. 2007. *Tracking and threat assessment for automotive collision avoidance*. Ph.D. Dissertation. Institutionen för systemteknik.

[6] J Erdfelt and D Drake. 2019. Libusb homepage. *Online, http://www. libusb. org* (2019).

[7] Mohammad Farsi, Karl Ratcliff, and Manuel Barbosa. 1999. An overview of controller area network. *Computing & Control Engineering Journal* 10, 3 (1999), 113–120.

[8] Marisol García-Valls and Roberto Baldoni. 2015. Adaptive middleware design for CPS: Considerations on the OS, resource managers, and the network run-time. In *Proceedings of the 14th International Workshop on Adaptive and Reflective Middleware*. 1–6.

[9] George Gunter, Derek Gloudemans, Raphael E Stern, Sean McQuade, Rahul Bhadani, Matt Bunting, Maria Laura Delle Monache, Roman Lysecky, Benjamin Seibold, Jonathan Sprinkle, et al. 2020. Are commercially implemented adaptive cruise control systems string stable? *IEEE Transactions on Intelligent Transportation Systems* (2020).

[10] Kyoung-Don Kang and Sang H Son. 2008. Real-time data services for cyber physical systems. In *2008 The 28th International Conference on Distributed Computing Systems Workshops*. IEEE, 483–488.

[11] Reza Malekian, Ntefeng Ruth Moloisane, Lakshmi Nair, Bodhaswar T Maharaj, and Uche AK Chude-Okonkwo. 2016. Design and implementation of a wireless OBD II fleet management system. *IEEE Sensors Journal* 17, 4 (2016), 1154–1164.

[12] Chetan Belagal Math, Ahmet Ozgur, Sonia Heemstra de Groot, and Hong Li. 2015. Data Rate based Congestion Control in V2V communication for traffic safety applications. In *2015 IEEE Symposium on Communications and Vehicular Technology in the Benelux (SCVT)*. IEEE, 1–6.

[13] Keith McCord. 2011. *Automotive Diagnostic Systems: Understanding OBD I and OBD II*. CarTech Inc.

[14] Mogillc. [n. d.]. mogillc/nico. ([n. d.]). https://github.com/mogillc/nico

[15] Wolfgang Pree. 1995. Design patterns for object-oriented software development. (1995).

[16] Alex Xandra Albert Sim and Benhard Sitohang. 2014. OBD-II standard car engine diagnostic software development. In *2014 International Conference on Data and Software Engineering (ICODSE)*. IEEE, 1–5.

[17] Raphael E Stern, Shumo Cui, Maria Laura Delle Monache, Rahul Bhadani, Matt Bunting, Miles Churchill, Nathaniel Hamilton, Hannah Pohlmann, Fangyu Wu, Benedetto Piccoli, et al. 2018. Dissipation of stop-and-go waves via control of autonomous vehicles: Field experiments. *Transportation Research Part C: Emerging Technologies* 89 (2018), 205–221.

[18] Ying Tan, Steve Goddard, and Lance C Perez. 2008. A prototype architecture for cyber-physical systems. *ACM Sigbed Review* 5, 1 (2008), 1–2.

[19] Monte Variakojis. [n. d.]. NMEA Parser Library. ([n. d.]). https://visualgps.github.io/NMEAParser/index.html

[20] Josef Wenger. 2005. Automotive radar-status and perspectives. In *IEEE Compound Semiconductor Integrated Circuit Symposium, 2005. CSIC'05.* IEEE, 4–pp.

[21] Yin Zhang, Meikang Qiu, Chun-Wei Tsai, Mohammad Mehedi Hassan, and Atif Alamri. 2015. Health-CPS: Healthcare cyber-physical system

assisted by cloud and big data. *IEEE Systems Journal* 11, 1 (2015), 88–95.