Openpilot System Architecture Analysis

by: Team Horizon

Authors

Ali Sheikhi	alish@my.yorku.ca
Erika Grandy	egrandy@my.yorku.ca
Jaskirat Pabla	pabla1@my.yorku.ca
Mate Korognai	korognaimat@gmail.com
Nabi Khalid	khalid18@my.yorku.ca
Natalie Dean	natrdean@my.yorku.ca
Shaharyar Choudhry	Sharry09@my.yorku.ca
Suha Siddiqui	suhasid@my.yorku.ca

Abstract

This report investigates the concrete software architectural composition of the openpilot system. Specifically, the subsystem known as 'panda' will be investigated using the Understand software to analyze the source code and determine dependencies. This will allow the team to acquire the concrete architectural composition of panda's internal infrastructure and will help with understanding how it fits into the big picture of openpilot.

First, a decomposition of the system architecture was found using Understand. It highlighted the use of the implicit invocation style, both event-driven and publisher/subscriber. Following this, the subsystem panda is explored. In openpilot, panda is the adapter between the car and openpilot components. It translates CAN messages and makes them understandable to other components of the system. This subsystem is then analyzed using Understand and its architecture was likewise found to be implicit invocation.

Design patterns, concurrency, and lessons learned were also discussed in the report. The report looks into the panda subsystem and what its possible design patterns may be and finds that there were no design patterns used directly. However, panda itself could be said to be the adapter of the openpilot system. Furthermore, due to the Implicit Invocation architectural style, concurrency is present. Since the components are loosely coupled, they can happen simultaneously, allowing for concurrency. Lastly, the team learned that concrete architecture includes many more dependencies and things that the team did not expect.

Introduction and Overview

This report dives into the architecture of openpilot, an open-source project that pushes the boundaries of what's possible in autonomous driving technologies. Openpilot combines hardware and software to create a system that can control a vehicle with little to no human intervention. Our focus is on the panda subsystem, an essential part of this project that bridges the gap between the car's hardware and the openpilot software.

Openpilot's system architecture is made up of several subsystems, each with its unique role. These range from handling the car's physical components to making decisions based on data from sensors. The architecture is complex, with many parts working together. However, by categorizing these subsystems into groups like decision-making, inter-vehicle communication, and hardware components, we can get a clearer picture of how openpilot functions.

Panda is particularly interesting because it enables the communication between the car's sensors and actuators and the openpilot software. It's a small piece of hardware that connects to the car and allows the software to read data and send commands. This makes it a crucial part of the system for autonomous driving.

Despite its importance, our investigation into the panda subsystem showed that it uses simple design patterns. This was surprising, as design patterns are common in software development to solve recurring problems. Instead, panda's code is straightforward, focusing on efficiency and speed. This simplicity might be because half of panda's code is in C, which doesn't support the object-oriented design patterns found in languages like Python.

Panda also uses external interfaces like the Controller Area Network (CAN) to communicate with the car's systems. This is vital for controlling the vehicle and reading sensor data. Additionally, the subsystem employs cryptographic techniques to secure the data it processes, highlighting the developers' attention to security.

Concurrency is another area where panda shows its strength. The subsystem can handle multiple tasks at once, like reading data from USB ports and encrypting information. This is crucial for a system that needs to concurrently process a lot of data in real time to control a car safely.

Through analyzing openpilot and specifically the panda subsystem, we've learned a lot about the system's architecture and the choices its developers made. While openpilot uses a complex set of subsystems to achieve its goals, panda remains focused on its role as a bridge between the car's hardware and the openpilot software, without complicating its design with unnecessary patterns. This report not only provides insight into the architecture of an advanced autonomous driving system but also reflects on the practical aspects of developing such a system.

System Architecture

The architectural layout of the openpilot system is highly complex as well as spread out. There are various subsystems from the highest level of abstraction to the lowest. These subsystems include: common, body, system, third party, teleoprtc, panda, selfdrive, rednose, opendbc, cereal, tools and tinygrad. Each of these subsystems provides an essential component to the complete cohesion of the system. Figure 1 presents the dependency relations of the previously listed subsystems. Figure 1 is a little convoluted due to the low level of coupling that is present in the architecture of the system. To overcome this problem, each subsystem can be classified and reorganized to get a clearer view that is easier to visually grasp.

To understand the system, the subsystems can be categorized into distinct groups as follows: the decision-making, the inter-vehicle communication/connection, the hardware components, and the utility grouping. The decision-making module is made up of tinygrad, self-drive, and rednose. These components receive data, filter out any noise or unnecessary characteristics and then process that to accomplish its safety-critical functionality. Next, the inter-vehicle connection/communication is made up of panda, opendbc, and cereal. These components communicate between the various components and translate it into an understandable format for the car through various means. The hardware components are made up of the body, system and teleoprtc. These are responsible for communicating to the inter-vehicle system. Lastly, the utility group is made up of common, tools and third_party.

These subsystems are used heavily all across the system as they provide universal tools that are used by many other subsystems. The grouping can be seen in Figure 2. Each grouping communicates with one another, but the specific subsystem dependencies are simplified to simple group communications.

The architectural style that can be seen in both Figure 1 and Figure 2 is implicit invocation. As the source code, different repositories and low level of coupling highlight, this system is a loosely-coupled collection of components. Thus it would make sense that it uses the implicit invocation architectural style to accomplish its mission. The interaction between each subsystem is through the use of procedure calls, publish/subscribe, events, and event buses. As the system has to accommodate many functionalities, it makes sense for it to be built like this. For example, when certain events are detected by hardware communications modules, they are sent to the inter-vehicle communications module which is then processed using the decision-making module. During all of these processes, each module may use the utility grouping to accomplish its tasks. In short, the openpilot architectural style uses implicit invocation to allow for communication between subsystems through the use of events and procedure calls.

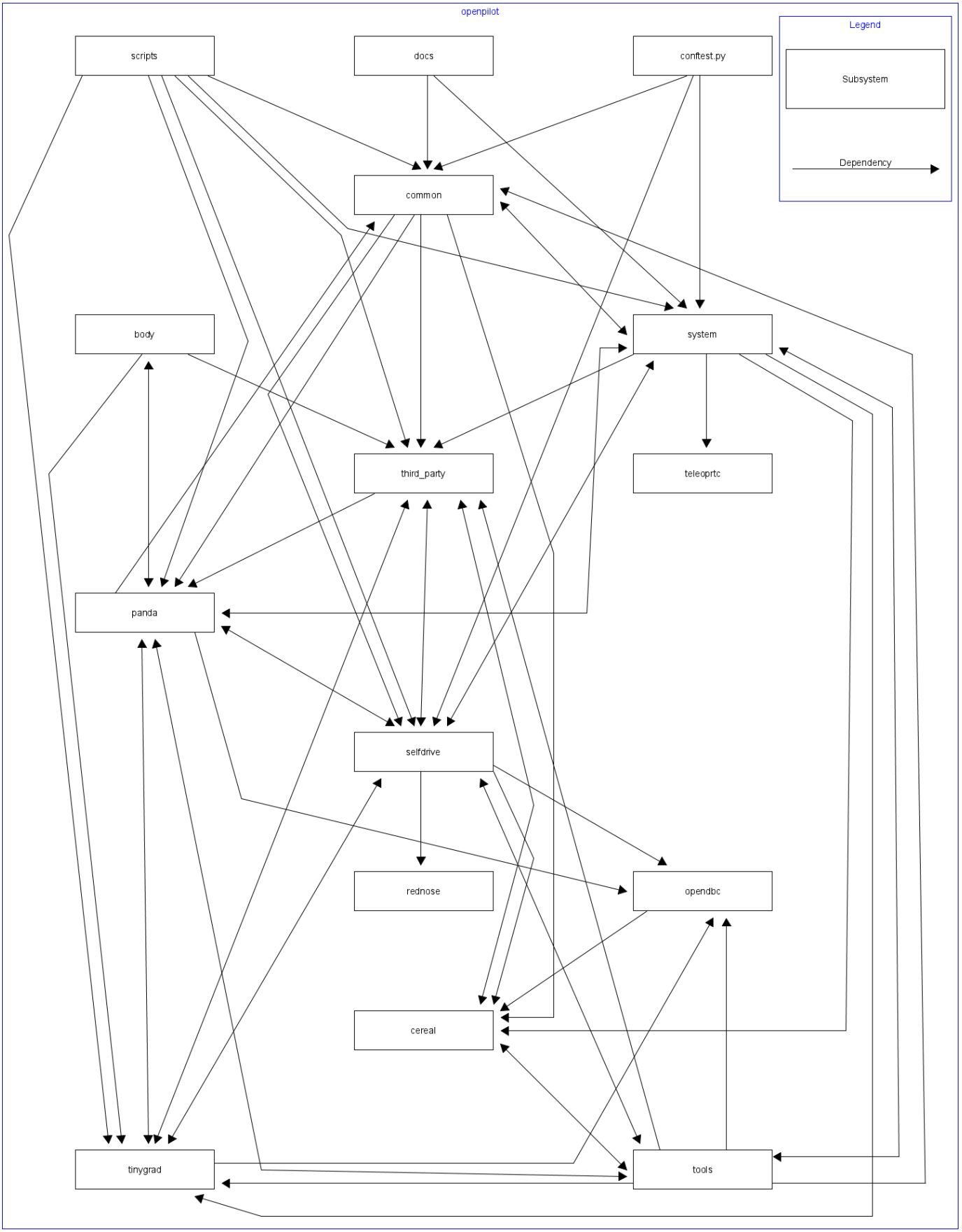

Figure 1: The highest level of subsystem dependency relation of the openpilot system

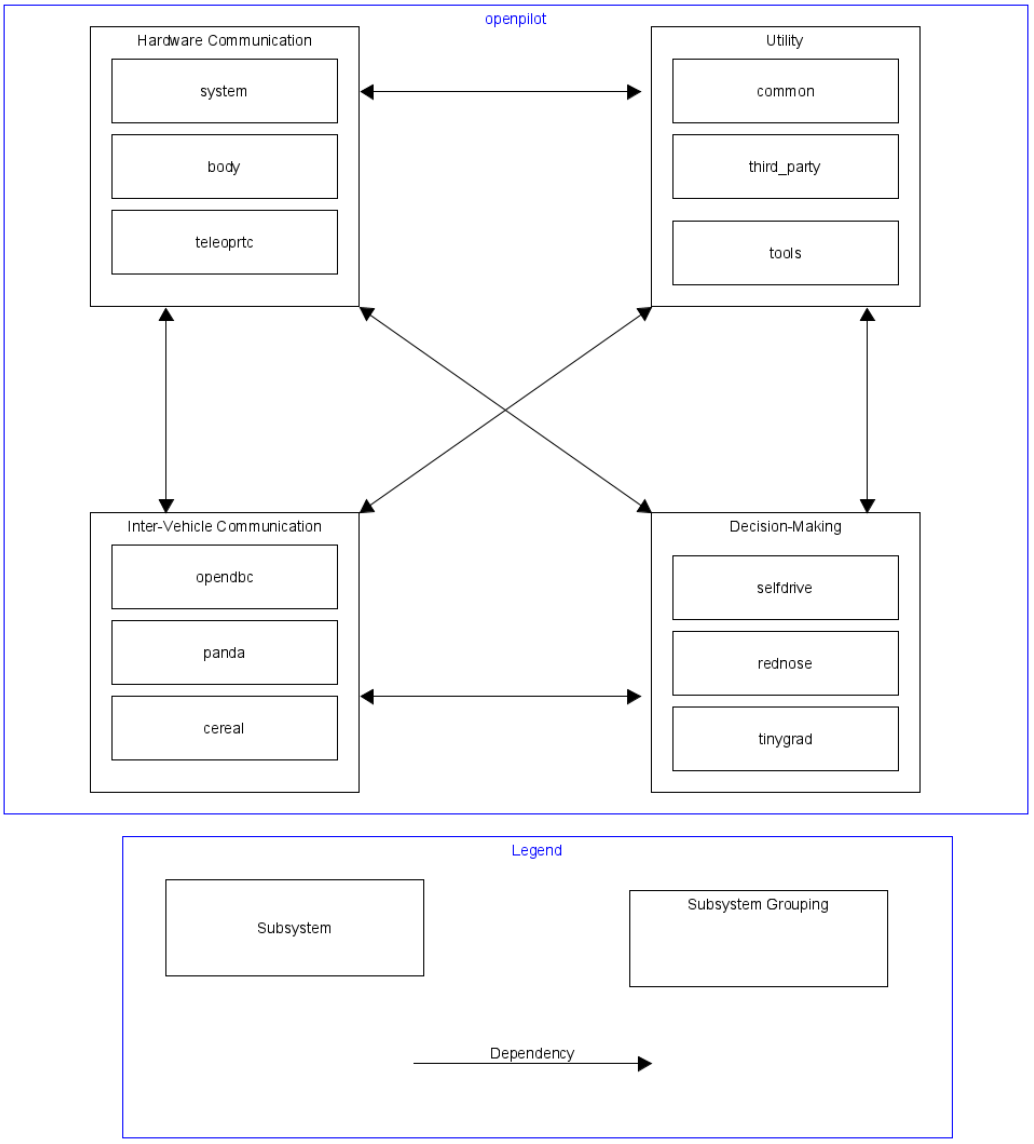

Figure 2: The highest level of subsystem categorization dependency relation of the openpilot system.

Panda

Our group chose to focus on the panda subsystem as the focal point of this report. The panda subsystem is a key component of openpilot as it enables communication between various hardware components and the software systems responsible for autonomous driving. Panda acts as a bridge between the vehicle's Controller Area Network (CAN) bus and the openpilot software system used for autonomous driving.

In terms of hardware, panda is a small electrical device that connects to your car via USB. Once connected, panda can communicate with various sensors and actuators to deliver a smooth autonomous driving experience. The sensors and actuators gather important information regarding the vehicle, allowing the openpilot system to make the correct decisions. These sensors and actuators can include cameras, GPS, steering actuators, braking actuators, etc.

In terms of software, panda integrates seamlessly with openpilot software, providing an interface for accessing vehicle data and sending vehicle commands. This integration allows for the openpilot software to leverage the data gathered by sensors and also send commands to actuators such as braking and steering.

Additionally, panda also goes beyond acting as an interface between hardware components and the software systems responsible for autonomous driving. Specifically, it includes features to monitor the health and status of different hardware components. It can also track data about the vehicle such as revolutions per minute, miles per gallon, battery life, etc. Panda also provides and enforces safety measures to ensure a safe environment for the user.

Overall, the panda subsystem serves as a critical component within the openpilot system, allowing for seamless integration between the vehicle's hardware components and openpilot software system necessary for a successful autonomous driving system.

Figure 3: The panda hardware

Subsystem Architecture

From the previous sections in which we extracted the concrete architecture of the openpilot system and explained the panda subsystem, the interactions it has with other subsystems can be extracted. Figure 4 highlights how it interacts with other components through the use of implicit invocation. Implicit invocation refers to both publish/subscribe and event bus-driven architecture. The use of events allows for the low coupling of the system as described previously so that each component can retain its open-source nature. If it was not designed in such a way, the system complexity would be much higher.

To understand and extract possible design patterns or architectural styles, one must look at the inner composition of the panda subsystem. Figure 5 shows the inner workings of panda. It makes heavy use of Python to accomplish its tasks. The main purpose of panda is to work with CAN messages from CAN buses and to accomplish this, a scripting language such as Python works well. It has to read data from usb ports, and as such the information it receives is serial. To capture this serial information, Python scripts can be set up to monitor ports and such. Each message goes through a pipeline of transformations that can be premade. Furthermore, panda also uses cryptographic hashing, encryption and signature to ensure information safety, which is what the crypto component is used for.

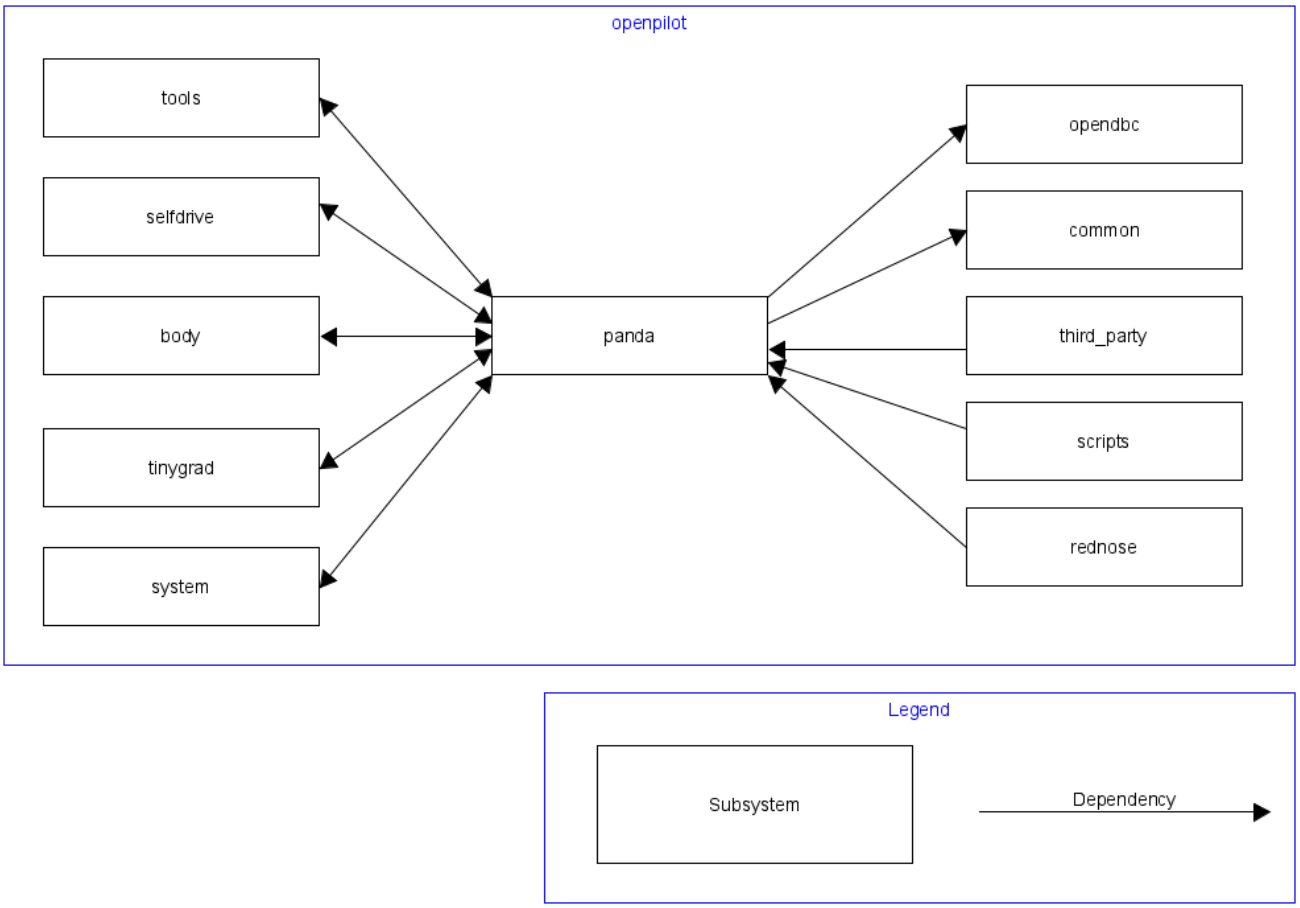

Figure 4: How panda fits into the openpilot system composition.

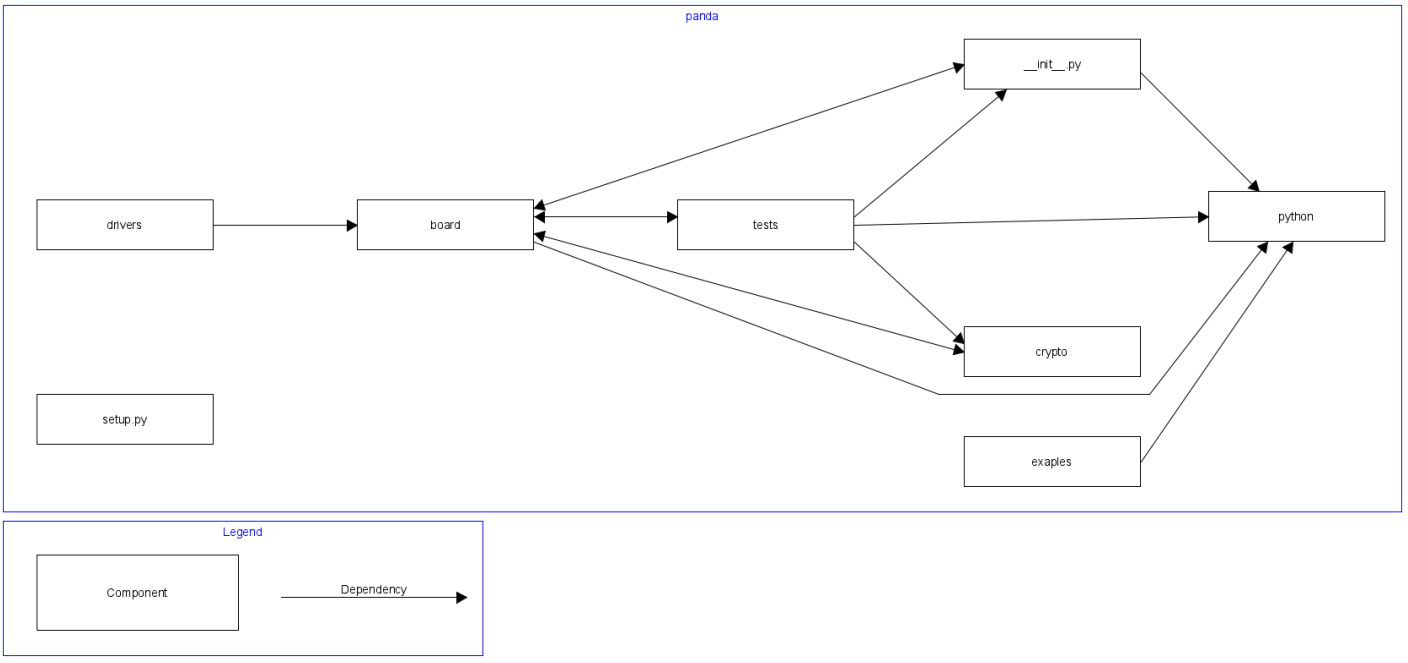

Figure 5: The internal composition of the panda subsystem

Subsystem Design Patterns

Panda does not use any noteworthy design patterns. Five design patterns were looked into as being potentially implemented in panda. The adapter design pattern was considered in the potential case that interfaces need to be adapted to work with different panda devices or vehicles. The observer design pattern was considered in the potential case that it is used for updating when new sensor data is received by panda. Iterator was considered in the potential case that any custom collections are used and require an iterator. The template design pattern was considered in the potential case that it was used to structure algorithms as skeletons with specific operations defined in subclasses. State was considered in the potential case that it was used to reduce conditionals by structuring and managing state machines with objects.

In viewing panda's structure it was found that about half of the code was written in C while the other half was written in Python. C is not an object-oriented programming language so it is less likely for object-oriented design patterns (OODPs) to be present in C code due to the language's lack of support for their implementation. As expected, no clear uses of OODPs in panda's C code could be found. An example of where one of the design patterns could have potentially been used but was found to not be is with panda's state machines. Panda implements state machines in multiple areas of the board firmware's C code which could have potentially used the state design pattern. However, a more traditional method of representing states as unsigned integers and managing them with switch statements was found to be used instead. This is likely due to the lack of support C provides for implementing OODPs, which would have made implementing the design pattern difficult, and the need for the board firmware to be very fast and efficient, which attempted object implementation in C could affect negatively.

In examining panda's Python code and class associations, it was found that inheritance was used minimally. The only instances in which inheritance is used are in creating Custom exceptions, extending abstract handle classes for communicating with the panda device to make handle classes for CAN, USB, and SPI communication, and extending panda and pandaDFU classes for interfacing with the panda device to create the pandaJungle and pandaJungleDFU classes respectively for interfacing with the pandaJungle debugging device. There is no instance where inheritance is involved in implementing any of the five considered design patterns, or any inheritance-dependent design pattern elements, so it was concluded that the Python code also has no uses of design patterns and the panda subsystem overall did not implement any design patterns.

Although the panda subsystem does not implement any design patterns itself, the subsystem could be seen as a component of a higher-level adapter pattern in openpilot. This concept stems from how part of panda's role within openpilot is to essentially adapt sensor information for use with various actuators in various components and brands of vehicles.

External Interfaces

The panda subsystem acts as a bridge between the openpilot system and the vehicle's sensors. Thus, panda requires the use of several key interfaces to achieve its data transfer. The main interface used is the Controller Area Network (CAN) along with CAN Flexible Data (CAN FD) messages. Panda reads and writes to the vehicle's sensors and actuators through CAN messages which are the main form of data transmission in vehicles. These messages enable real-time control and monitoring of the vehicle, covering a wide range of functionalities from engine management to safety systems. CAN operates using only two

wires to enable multiple Electronic Control Units (ECU) to converse. Each ECU is allocated a unique ID that remains constant, serving as a reliable identifier for the source of messages. Within the CAN framework, message priority is indicated by the ID number. Lower IDs signal higher importance and higher IDs suggest lower priority. For example, the system ensures that messages from the Powertrain Control Module signalling engine issues take precedence over less important messages like temperature readings.

The next external interface used by panda is the opendbc Files. These are essential for decoding CAN messages, serving as the translation layer that allows openpilot to understand and interact with a vehicle's CAN network. Each vehicle model may have multiple CAN buses, each represented by its own DBC file within the openpilot system. Panda uses these files to correctly interpret messages from the vehicle's network. Some messages include information about the vehicle's speed, brake status and wheel angle.

The boardd component within openpilot utilizes the cereal communication protocol to bridge panda with other modules in the system. This protocol ensures structured and reliable data exchange between the vehicle's hardware components (through panda) and the decision-making algorithms within openpilot, facilitating a coherent flow of information necessary for advanced driver assistance functionalities.

After analyzing the source code files of the panda subsystem, it is evident that panda employs cryptographic hashing, encryption, and signature verification to ensure the integrity and security of the data it processes. This layer of security is crucial for preventing unauthorized access and manipulation of vehicle controls, which could result in safety implications. To further explore how panda hashes its messages, the RSA, SHA and signing. C source files can be found in the Crypto folder within the panda folder.

Concurrency

The subsystem’s design incorporates implicit invocation mechanisms, including publish/subscribe and event-driven bus architectures. This design choice enables certain components to operate concurrently, enhancing the system’s efficiency. Specifically, the subsystem is adept at processing serial data from a USB port in real time, allowing it to function simultaneously with other modules like encryption and CAN translation. The encryption module is particularly versatile, capable of encrypting data blocks using RSA public key encryption, securing them with SHA hashing, and appending digital signatures. Remarkably, this module operates as an independent Python script, facilitating concurrent execution with other system processes.

Furthermore, the subsystem demonstrates its concurrency prowess by transmitting data serially through ports, seamlessly integrating with the ongoing activities of other processes. Panda, the subsystem in question, also exhibits concurrent behaviour. However, its operations are not entirely independent. Given its role as an intermediary, it must execute a sequence of interdependent components to accurately translate and relay CAN messages. This sequential dependency renders panda only partially concurrent, with simultaneous actions limited to data transmission, reception, translation, and encryption.

Use Cases

As defined above, the panda device is a piece of hardware interface which connects a computer (Android device) to a car over USB. In Figure 6, the data transmission between the Android device and the panda device is shown, taking the camera, GPS, and infrared comma sensors as input for the Android device. The main purpose of panda device is to provide a communication gateway between the CAN bus and the openpilot software.

Each component in Figure 6 [1] provides a use for the panda device. Firstly, body is a component which would use panda as an intermediary between the sensors and actuators. That part of the code would be responsible for dealing with the control of hardware on the car. The three nodes of tinygrad, selfdrive, and rednose are responsible for decision-making. For example,tinygrad is important and is used for machine learning models. Also, the rednose node would be used for data processing for the most part as it is meant to filter the data and reduce noise. Furthermore, there are other nodes which are responsible for inter-vehicle communication such as panda, boardd, opendbc, and cereal. For example, boardd facilitates communication between the panda module and the communication module cereal. Also, the opendbc is responsible for serving as the backbone for understanding and interpreting the CAN bus traffic of a vehicle. The opendbc would be used for synthesizing the information such as decoding from CAN format. Additionally, cereal would be used for communication library/standard (sends packets of data between sensor processes, Ex. a sensor process would use cereal to send IMU data to calibration and localization processes). Below is a diagram of a sample use case which demonstrates the purpose of rednose in relation to the panda interface (Figure 7).

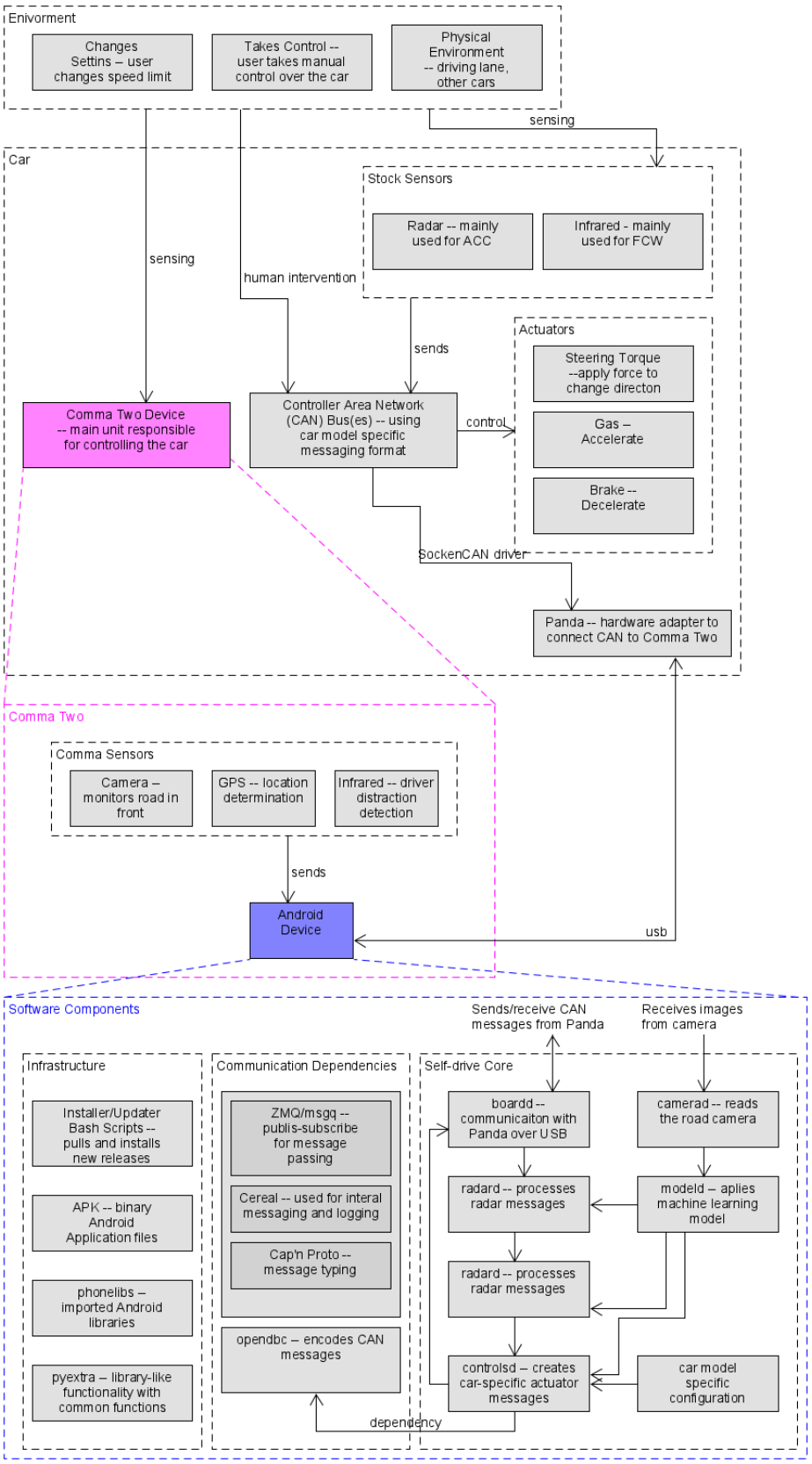

Figure 6: DESOSA architecture diagram demonstrating information flow [1]

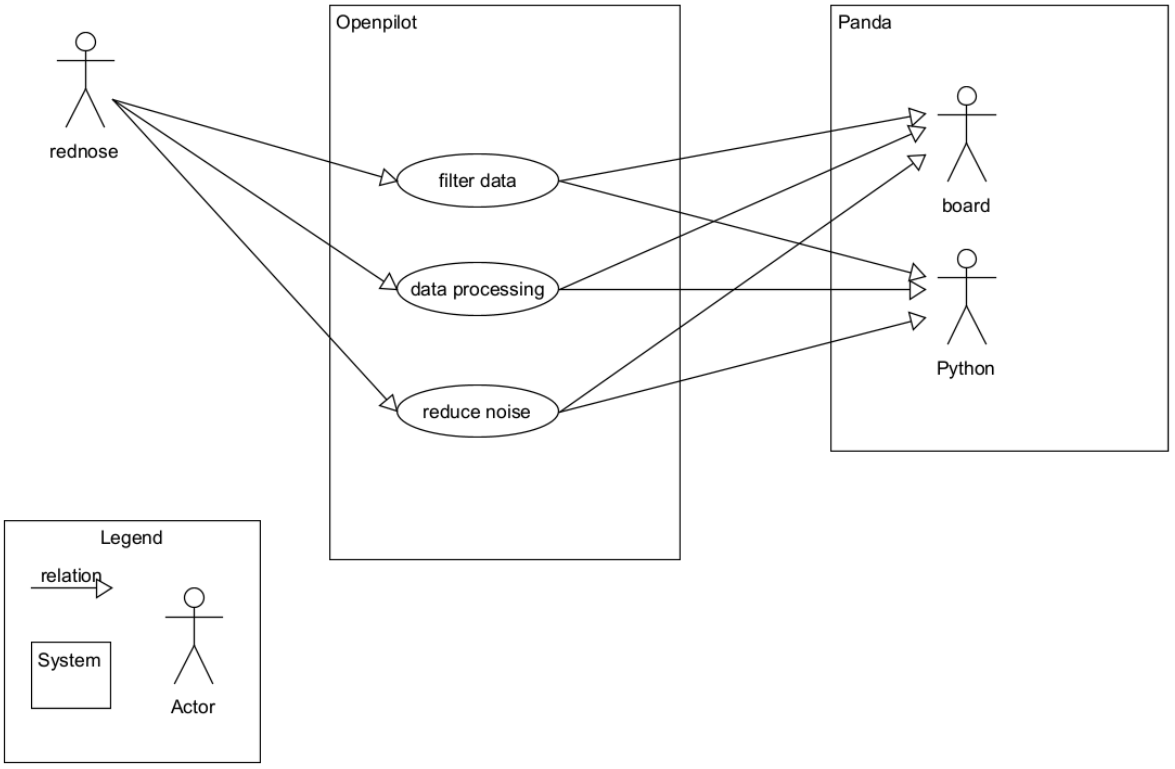

Figure 7: Depicting the use cases of red nose in perspective of panda

---

Lessons Learned

Investigation into the concrete architecture of openpilot allowed for a variety of new lessons learned about openpilot, as well as software design and architecture. In terms of openpilot, this deep-dive allowed for an increased understanding of openpilot and the different components involved. As compared to previously, when making a mockup of the conceptual architecture given documentation, looking into the openpilot source code has allowed for an increased understanding as to how the developers have chosen to split up the overall system. This deeper understanding, and concrete view of component groupings and relationships, has also allowed for a greater overall understanding of each individual component and its main tasks.

Regarding panda and openpilot specifically, insight was gained into the use of architecture styles, including seeing concrete examples. An example of this included seeing multiple uses of implicit invocation, and seeing how it can be used to provide loose coupling. There was also insight gained into the lack of design patterns used - at least the ones familiar to us. A big part of this was the use of C language and lack of inheritance, which is required for most design patterns. Additionally, viewing the source code allowed for new understanding regarding specific choices made by the developers, such as how they incorporate hashing and encryption as a security measure, or make use of IDs as part of message prioritization algorithms.

In a more general sense, the investigation into openpilot's concrete architecture also resulted in long term lessons learned. A major lesson involved the use of the Understand software, and how it can be applied to open-source software to examine structure. Although useful - Understand supplies only the current file directory hierarchy of the project, and

additional work is needed to manually group into subsystems. Regardless, learning how to use the software has definitely been a lesson learned, especially in gaining experience in group system structure by subsystem behaviour.

Derivation Process/Alternatives

The derivation process for acquiring the architecture of both the overall openpilot system and panda are very similar. First, the entire github repository of the system was cloned locally, and then Understand was run on the system to get its basic directory-level structure. Following this, the dependencies were gathered from the graphs and our group discussed our findings. We talked about it and then decided to group components of the openpilot system, which can be seen in the section 'System Architecture' under Figure 2. We made the groupings based on functionalities, so their grouping was based on behaviour. Following this, the specifics of panda were investigated. For panda, the subsystem structure was analyzed.

No alternatives were found during our investigation. We found a heavy use of implicit invocation because the components of the system were loosely coupled. However, other than that, no other concrete architectures were found. In terms of design patterns, none of those were found in panda. Panda may not have used design patterns, but in the system of openpilot, panda can be said to be the adapter of the system. It is a design pattern where it allows for communication between two non-matching members in an efficient way.

Conclusion

Overall, the concrete architecture that the team found is complex. It has a lot more hidden components and dependencies that we could not gather from documentation alone. There were many discrepancies with our guesses for the system, but that will be discussed in assignment 3. For this report, we only looked at the concrete architecture and nothing else. This allowed us to delve deeper into openpilot and panda to get a solid understanding of how it all works. It allowed us to name the architectural styles present as well as to point out any design patterns that we noticed. In conclusion, there were a lot of unexpected findings during the making of assignment 2, but it made the team understand how a real system such as openpilot can become a cohesive whole that works properly.

References

[1] “From vision to architecture: How to use openpilot and live,” From Vision To Architecture: How to use openpilot and live - DESOSA 2020, https://desosa.nl/projects/openpilot/2020/03/11/from-vision-to-architecture (accessed March 13, 2024).