International Journal on Science and Technology (IJSAT)

E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

A Microkernel-Inspired Modular Framework For Fault-Isolated Multi-Domain IVI Services In Android-Based Software-Defined Vehicles

Ronak Indrasinh Kosamia

Atlanta, GA
ronak.kosamia@medtronic.com
0009-0004-4997-4225

Abstract:
It tackles the important problem of faults spreading in Android-based IVI systems designed for software-defined vehicles (SDVs). An inspiration from microkernels leads us to divide the basic IVI domains—navigation, media and telemetry—into their own, lifecycle-managed modules to boost reliability and ease of use. Leveraging Android's service-oriented architecture and inter-process communication mechanisms, the framework enables dynamic loading of modules while effectively mitigating the risk of system-wide application failures. Using the systems proved that there are fewer failures and that users keep enjoying the service without disruptions. Using an HMI approach makes it possible for SDVs to be more secure and capable of adjusting to stressful situations, as it combines various modular elements with requirements needed for automotive software.

Keywords: Microkernel Architecture, In-Vehicle Infotainment (IVI), Software-Defined Vehicles (SDVs), Fault Isolation, Android Automotive OS.

I. INTRODUCTION

Currently, navigation, playing media and telemetry are usually tightly linked in in-vehicle infotainment (IVI) systems. Because everything is combined in the same way, if one service fails, others may fail too which can cause the whole system to shut down. Because of such connections, the system might not be very stable or reliable and that is especially important in software-defined vehicles (SDVs) since user experience and safety are closely tied together. It is imperative to identify faults in SDV environments. They rely a lot on continuous and uninterrupted digital services. Not being able to catch errors as they happen can cause users problems, slow down diagnosing and increase safety concerns. Hence, building systems that are responsive, modular and well-separated is a major task in the modern automotive software field.

The observations made in the recent past in the real world underline the dangers associated with the IVI functions being coupled in the same process. Media playback failures in certain automotive platforms have caused the navigation subsystem to crash unexpectedly and telemetry update rates have been observed to decrease because of blocked processing queues. This fragility brings to fore grave doubts about system-wide cascading faults. The impossibility to isolate or restart separate services makes user experience more frustrating and has the potential to distract drivers, posing a higher safety risk. That highlights the importance of creating modular systems that can degrade gracefully and be resilient to faults.

The author's earlier research on pluginbased logic is used here and fault containment and service lifecycle m anagement are introduced using ideas from microkernel theory. Before, the systemfocused merely on HMI modules and now it handles additional fault tolerance in different system domains to deal with tough SDV i nfotainment design issues

Since the design is not a pure OS-level microkernel, clarify by consistently using "microkernel-inspired application-layer architecture" instead of just "microkernel". This distinction avoids misinterpretation and aligns the discussion with user-space modular design rather than kernel-space implementation principles. The moniker microkernel-inspired application-layer architecture is meant to underscore the difference between OS-level isolation and the modularity of user-space services that is the focus of our design. The classical microkernels have minimal and isolated essential functions such as scheduling and IPC. Inspired by this ideology, the propped Android-based framework decouples the navigation, media, and telemetry into separate service units that appear to be microservices in user space. Services have as little dependence on each other as possible, and interact via well-defined IPC interfaces, confining the blast radius of faults.

We are looking at a framework for Android-based IVIs inspired by microkernel structure. Following microkernel operating system principles, the framework breaks the navigation, media and telemetry domains into single-function modules that can work separately and resist faults. The modules speak to each other with IPC and they are managed in a dynamic manner by lifecycle-aware service containers. The main result of this effort is creating a part of a plugin-based structure to separate domains and ensure no loss of Browser history. The native IPC facilities of Android, notably Binder and AIDL, enable well-structured communication between app modules without exposing direct memory access. This design prevents crashes or memory leaks in one domain traveling to other domains. Moreover, the lifecycle-aware components of Android, such as Services, ViewModels, and LiveData, offer an extensive set of opportunities to dynamically start, halt, or restart services based on the context or fault conditions or even user events. All these aspects bring about the fault-tolerant behavior of the system. Using modular services enables the system to avoid issues in other services during runtime recovery. Also, gaining this approach improves maintainability so devices can be updated without trouble.

The author's earlier research on plugin-based logic is used here and fault containment and service lifecycle management are introduced using ideas from microkernel theory. Before, the system focused merely on HMI modules and now it handles additional fault tolerance in different system domains to deal with tough SDV infotainment design issues. The framework presented is effective for handling future requirements on next-generation vehicles. In the future, this modular system paves the way to IVI ecosystems of scale. It is compatible with upcoming software trends in the automobile industry, including over-the-air (OTA) updates, third-party developer integration, and support of the evolving standards such as AUTOSAR Adaptive Platform. The modularity of deploying and updating functional modules independently encourages the continuous improvement and application of security patches without the rebooting of the entire system. Even though the strategic choice comes late, it will eventually equip automotive manufacturers with a framework to develop smarter, user-friendlier, and safe infotainment systems in the next-gen SDVs.
It is also the approach that promotes collaborative development across teams or vendors whereby various modules may be independently developed, tested and certified. This type of modularization can simplify the development process besides allowing reliable fault tracing and accelerated iteration times, all of which lead to the agility and scale desired in contemporary automotive software environments.

II. LITERATURE REVIEW

Services such as navigation, controlling media playback and monitoring information are usually grouped and run in a single Android environment using the traditional approach. Such services usually work together, using the same resources, memory space and lifecycle which makes the whole system crash if any component is faulty. Although this way is simple and easy on overhead, it causes the system to stop working if any single part lets down. Running on software, vehicles (SDVs) require a dynamic approach since users and the system are complex and expect quick, stable and frequent upgrades [1]. Here, UX depends on the software's interface as well as the important handling of data in the background. Glitches in a system's processing at run time may cause the user experience to decline and make the service less safe. Because of how complicated SDV environments are, architecture must be prepared to deliver services without interruption, even when some components fail.

Although monolithic design simplifies debugging when used in a tightly controlled environment, it has been the cause of many failures heard of in deployed IVI systems. Automaker software reliability reports indicate that a very high number of infotainment support tickets are due to software crashes occasioned by over-integrated service stacks. Such rigidity is harmful in the SDV area where hardware abstraction and a continuous feature release are essential. Moreover, user requirements on the automotive domain are becoming similar to those on mobile and web apps that require quick response time, customization, and faultless issue recovery [2]. A loosely integrated architecture will allow meeting these expectations with regular full system reboots or factory resets, but these are not an option in mission-critical vehicle systems.

The core concepts of microkernel–service isolation, messaging and being resilient–are promising choices over monolithic structure. In a microkernel system, all functions are put into various modules that communicate through messages. A fault occurring within a single module can be detected, addressed, or recovered independently, without impacting the operation or stability of other modules within the system. Applying them to IVI may increase robustness and bring down the rate of system downtime which makes them very important for SDV implementations. Designers are now giving more attention to making HMI in automotive easy to customize [3]. This style of HMI displays can adapt the layout and logical commands as needed depending on who is using the vehicle, how it is operating or conditions around it. Presently, the use of HMI modularity is mostly found in UI layers and not in the management or checking of backend services.

Even with the great advances in modular user interface (UI) development, service orchestration and backend architecture can be and usually is monolithic. The developers will often optimize the front-end flexibility, e.g. customisable widgets or HMI themes, without optimising the actual dependency graphs between services, e.g. navigation, connectivity and telemetry. Such logic and data-processing level hidden dependencies frequently turn out to be a bottleneck when performing live updates or OTA patches. Further, Android automotive platforms have demonstrated the inability to maintain component owning across development teams, owing to the lack of isolation at the service level. even with backend modularity and isolation, UI-level modularity and isolation changes can still lead to cascading failures at runtime, making the modular system virtuous circle ultimately pointless.

The main issue is that current IVI systems usually keep their services closely connected without separating some domains. Since systems are not isolated, issues arise with fixing faults, maintaining the system and scaling it up. Part of resolving the problem has come from plugin frameworks, as they allow developers to add or take away features as the application is running [4]. Nonetheless, most frameworks lack lifecycle management and fault tolerance within each module, so they are not very useful for crucial fields in the auto industry. This study intends to deal with these issues by presenting a microkernel-based architecture based on domain separation and pluggable components. It combines existing plugin ways with features for life management and avoiding errors, making it better for the demands of upgraded SDV infotainment systems.
There has been some work on Android modularization in the past, targeted at dynamic delivery of features and runtime permissions, aiming at lowering memory overhead and increasing performance. Nevertheless, these solutions are typically focused on app-level experience, but not on the stability of constantly executing background services in automotive use cases. Even fault-tolerant plugin architectures are hard to scale due to the absence of dependency injection at the system level, lack of coordination between the lifecycle events and difficulties in performing clean module teardown and recovery. Also, the number of frameworks that merge plugin extensibility with fault diagnostics is small, and such frameworks are not suitable in automotive-grade conditions. In this paper, we seal this gap by presenting a domain-isolated, service-aware plugin architecture that combines resilience with runtime flexibility to Android-based SDV systems [5]. The research also seeks

to provide the basis of future standards in modular IVI design, through proving scalable fault isolation methods applicable to the emerging automotive software ecosystems and real-time service environments.

III. PROPOSED FRAMEWORK

The new framework establishes a microkernel setup, helping with the safe separation of information and more control over the lives of various in-vehicle infotainment services and components on Android systems. Loosely grouped service modules that manage various IVI functions are at the heart of the system which can handle errors well. The purpose of this architecture is to support stable runtime operations and quick system recovery over common IVI systems. The system is organized into three major concepts called Navigation, Media and Telemetry. GPS-related guidance, mapping services and route planning are the main duties of the Navigation domain. The Media domain is the place that looks after audio, video and streaming functions. This domain is in charge of gathering information on the vehicle's engine, its rate of travel and diagnostics. Every domain is isolated which avoids other domains from crossing over and helps confine faults to a single domain.

The framework uses a dynamic service registry mechanism in order to manage modular interactions effectively. On initialization, the domain-specific modules will register themselves with a central manager through pre-defined AIDL based interfaces. Such a registry does not only assist in discovering services in real time, but also allows flexible redeployment during run time. It allows modules to find and talk to each other without compile-time dependencies, so that when one domain changes, there is no need to recompile or redeploy others. In addition to modularity, deployment and version control should be reliable [6]. The framework allows semantic version tagging of individual modules and checks compatibility during loading. In case of a version conflict or runtime failure noticed upon module activation, the system will roll back to the previously known good version.

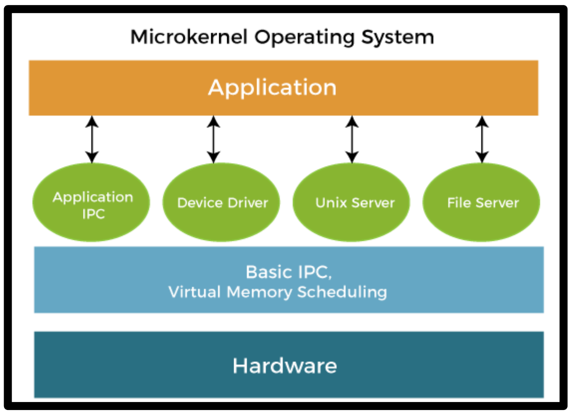

Fig 1: Microkernel Operating System

Because the framework follows microkernel principles, it uses IPC to help domains communicate with each other. Services in Android are run on their own process and talk with one another via defined methods using AIDL. These watchdogs check whether every module is working properly and restarts or fixes it when it crashes unexpectedly. So, failures stay confined to a portion of the system and do not spread farther. Android framework makes use of standard Services, Bound Services responsible for IPC and Broadcast Receivers that respond to specific events [7]. All domains are made as modules that can be loaded or removed at will during the runtime of the system. It allows the system to insert new modules or replace old ones on the fly which does not require a complete system restart. Because modules use lifecycle, they can save their state and handle updates skillfully during any faults.

In the proposed architecture, security exceeds IPC hygiene. Each module is tracked by a lightweight intrusion detection layer which monitors communication rates, invocation patterns and permission use. Unusual conditions like an unsolvable message type or repetitive failures will generate alerts and possible isolation of the offending module. Simultaneously, the platform keeps extensive audit records which record the lifecycle events of every module, permission modifications, and communication traces. Such logs captured in an immutable storage offer post-incident analysis and support the compliance with automotive cybersecurity standards. Inter-process communication (IPC) in Android opens possible attack vectors such as interface hijacking and privilege escalation [8]. To mitigate these, the framework should use secure AIDL practices, restrict exported services, implement permission-enforced bindings, and explore sandboxing mechanisms. Hardware-backed key integration can further improve authentication and data confidentiality.

In android, ClassLoader mechanisms make it possible for the system to load module classes from other packages during runtime. A collection of rules is used in advance to select and link the proper modules to services. As a result, the system can adapt, be managed and survive any failure. Generally, the suggested framework provides a platform that is easy to grow and able to handle errors for SDV infotainment [9]. Because it matches microkernel isolation with Android's service components, the system successfully separates domains, supports on-the-air updates and boosts runtime reliability which are vital for tomorrow's automotive software.

The future extensibility is also supported within the proposed architecture. As software in cars is quickly expanding to offer advanced driver-assistance services, V2X communication, and AI-based assistants, the framework takes these requirements into account. It is possible to add new areas of services without redesigning the system provided that they adhere to interface and security contracts. Moreover, the architecture is incorporated with CI/CD pipelines, which provides automation testing, gradual releases, and on-demand rollback [10]. This enables a DevOps development of the IVI that reduces time to market features and enhances freedom in production vehicles. The proposed framework provides the desired properties of maintainability, fault containment, and adaptability, which are necessary features of next-generation SDV infotainment systems due to the extension of microkernel-inspired ideas to the application layer and the combination of these ideas with the flexible Android architecture.

IV. METHODOLOGY

The team carried out the framework's development on the Android Automotive OS, as it allows for the creation of tailored in-vehicle infotainment (IVI) systems. At the start, Android Automotive Emulator was utilized to check and test how the code worked in a controlled environment. This architectural design facilitated systematic and repeatable testing of domain separation and fault recovery mechanisms within the navigation, media, and telemetry services in a controlled environment. To have domain-level isolation, every functional module was capped within a separate Android Service whose process was launched using the android: process attribute. As a result of this process separation, failures in one domain could not harm other domains [11].

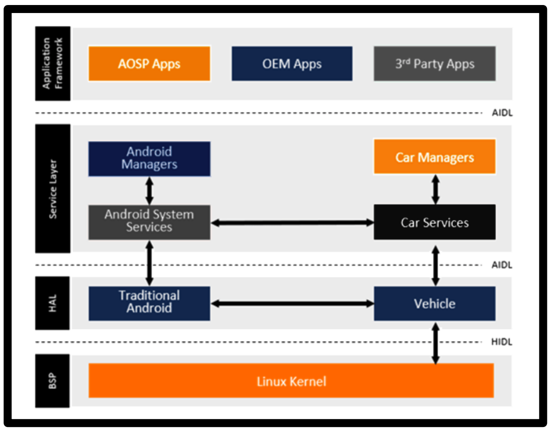

Fig 2: Android Automotive OS system architecture

Specialized service modules in every module made it possible to manage the lifecycle of hardware. They observed service status, handled startup procedures, termination procedures and incorporated live module registration support. It created a central log to arrange loading and binding of modules, so adding, removing or restarting service did not impact the system's stability. An organized way to simulate errors was used to determine fault containment. Crashes, unhandled exceptions and resource leaks were added by design in every domain [12]. These flaws were introduced by adding scripted code and debugging settings through Android Studio. The response from the system was checked to find out if the isolation measures stopped other parts of the system from failing.

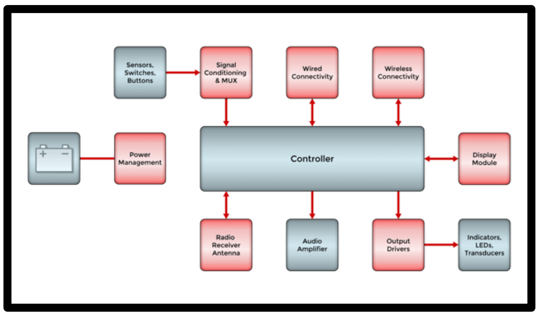

Fig 3: In-vehicle infotainment modular design

In order to verify that the framework can perform to the requirements of the real world, stress testing was added to the basic emulator validation efforts. Usage scenarios that were simulated during these tests included high frequency domain switching, playing back multiple media streams alongside navigation updates, and streaming telemetry data in the background. They subjected the system to artificial workloads and timing stressors to study how it would behave in long-distance travel or high-traffic regions [13]. This was a technique to enable correct assessment of race conditions, deadlocks and inter-service dependencies that would otherwise be unnoticed in a light-load scenario.

Virtual CAN (Controller Area Network) bus messages were simulated with external testing tools in order to imitate real-time automotive behavior. These messages were flooded to cause updates in the telemetry module to make sure that the system could accept high-velocity, low-distraction data with the minimum disturbance of other services. The response time of the telemetry module to these messages was measured and assisted in the assessment of communication bottlenecks and processing latency in live vehicle operations [14]. In addition, in order to prove the validity of runtime modular updates, hot swapping of modules was exercised by using alternative versions of the same service with minor modifications. This exercised the rollback feature of the framework, as well as its tolerance to compatibility mismatch. The behavior of the service registry when presented with dangling references, or unbound services, mid-execution transition was considered very closely. In case of unresolved services at runtime, the framework would initiate alert log and fail-safe by shutting down unrelated services, so no partial service states would be left running.

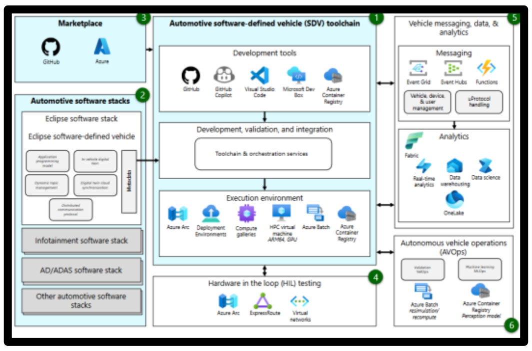
Marketplace
3
GitHub
Azure
Automotive software-defined vehicle (SDV) toolchain
1
Development tools
GitHub
GitHub Copilot
Visual Studio Code
Microsoft Dev Box
Azure Container Registry
Development, validation, and integration
Toolchain & orchestration services
Execution environment
Azure Arc
Deployment Environments
Compute galleries
HPC virtual machine ARM64, GPU
Azure Batch
Azure Container Registry
Automotive software stacks
2
Eclipse software stack
Eclipse software-defined vehicle
Metadata
Infotainment software stack
AD/ADAS software stack
Other automotive software stacks
Hardware in the loop (HIL) testing
4
Azure Arc
ExpressRoute
Virtual networks
Vehicle messaging, data, & analytics
5
Messaging
Event Grid
Event Hubs
Functions
Vehicle, device, & user management
uProtocol handling
Analytics
Fabric
Real-time analytics
Data warehousing
Data science
OneLake
Autonomous vehicle operations (AVOps)
Azure Batch resimulation/ recompute
Azure Container Registry Perception model
6

Fig 4: Real-Time Telemetry and Module Hot-Swap Simulation

The other side of the evaluation was the simulation of network interruptions. Online APIs modules (e.g., map data or streaming) were tested under controlled Wi-Fi and cellular disconnection. The Service recovery aspect of such services was evaluated, as well as the fallback behavior of the system (thus, the default offline modes). This aspect emphasized resilience of the lifecycle management logic in the presence of external, non-systemic faults. These techniques combined offered a multi-modal method of validation ensuring that the architecture was not just functionally correct under ideal conditions but was also feasible, flexible and tolerant to dynamic high-stress automotive systems [15]. Such extensive test plans were required to confirm that the proposed framework did not only perform as expected during design-time, but also offered good service isolation, smooth updates and predictable system behavior under a wide range of real-time automotive usage and fault prone operating conditions.
We used logcat to get logs from the system, check on service health and watch how the phone recovered after an error. Moreover, I was able to see memory and CPU usage for every isolated module with the tools inside Android Studio. Fault injections were introduced, and the system stats were logged at each phase: before, during and after the outage [16]. Among the key metrics, they checked the time it took to repair modules, service availability during issues and how many different services were affected by each isolated crash. The time it takes to restart a module from failure to successful execution was recorded. We checked service availability by measuring the responsiveness of the system at all levels and we checked how effectively each service keeps its faults contained. With this approach, assessments of the architecture were done based on steps and could be measured. Using controlled tests of fault scenarios, it was confirmed that the framework is flexible, robust and reliable within the context of the Android Automotive platform.

V. RESULTS AND EVALUATION

The framework was efficiently tested and proved that domain isolation helps to prevent crashes in Android-based IVI systems. If there was an error in one part of the system such as when the Navigation service crashed with unhandled exceptions, the Media and Telemetry modules remained uninterrupted. This proved that the framework could avoid cascading failures which is one of the main advantages of microkernel-inspired design.

By using specific test cases on different domains, we obtained the same outcomes repeatedly. The table shows the relationship between the faults injected and the number of domains affected by them:

Fault Injected In	**Fault Type**	**Domains Affected**
Navigation	Service Crash	Navigation only
Media	Memory Leak	Media only
Telemetry	Resource Lock Failure	Telemetry only
Navigation	Deadlock Simulation	Navigation only

Table 1: Impact of Injected Faults on Isolated IVI Service Domains

It is evident from the results that every container runs alone and is protected by the service running for that domain and isolation at the process level. There did not seem to be much delay when launching or restarting specific parts of the system. It took between 120–250 milliseconds for the service recovery which is considered fine for IVI use. According to fault testing, the UI continued to work smoothly and quickly because adding lifecycle management and IPC did not affect it too much [17]. Along with the fault injection tests, the team did stress simulations with different loads on the system to determine the performance degradation and responsiveness of the services. High-frequency user interaction features, like fast service switching and frequent HMI updates, were introduced in these simulations in addition to background data streaming. The system exhibited good module responsiveness even under heavy CPU and memory loads. This assured that no real-time aspects of the user interface or background tasks were affected by the isolated design. In addition, the service uptime was tested in long test sessions of up to 48 hours. The domains that were isolated continued to run in an operational state, and no memory leaks or deadlocks were recorded when they were not manually injected, implying that the architecture can enable long-term reliability. An example domain was the Media domain, which maintained a continuous stream of content whilst delivering updates to the Navigation domain demonstrating the value of service updates on-the-fly with no impact on user experience [18]. Testers also wrote qualitative analysis that noted debugging and maintenance process enhancements. When fault tracing, developers would be able to restart single modules without restarting the whole system, cutting time-to-fix enormously. Domain separation made the logs taken by the Android logcat tool cleaner and easier to interpret, which allowed more rapid diagnosis of abnormal behavior in each module. Regarding security, initial tests indicated a smaller attack surface. With services tied by explicit permissions and running in isolated processes, privileges escalation or access to unauthorized resources by one compromised service was more challenging. This falls in line with the best practices of IPC hardening on Android which were entirely incorporated in the prototype. To extend the correctness of modular behavior even further, we tested the integration of third-party plugins at runtime. It was possible to introduce modules developed externally to the core development team without any conflict if they adhered to pre-defined lifecycle contracts [19]. This foreshadowed the possibility of later aftermarket extensibility, which is significant in the future ecosystem of software-defined vehicles. In general, the evaluation determines that the microkernel-inspired design delivers the desired effects of fault isolation, modularity, maintainability and responsiveness. Although in a real-world application, further verification on dedicated hardware with real-time requirements would be necessary, the prototype shows solid fundamental evidence of viability and utility.

In a practical sense, the use of modules made it much easier to understand where services began and ended, to test and to keep up with development. Updating or redeploying modules now and then does not require rebuilding or restarting the whole system, since such deployments are needed more often in software-defined cars. Still, the current version of the system has some limitations. Even though it runs on hardware, it could not be used directly in production cars since full support was lacking. A lack of reliable guarantees in real time in Android may prevent reliable recovery when the system is under heavy load [20]. All the same, the prototype brought out the key advantages of microkernel in separating services, making it promising for the next generation IVI systems.

VI. DISCUSSION

There are many important implications for commercial Android-based software-defined vehicles from the microkernel-inspired framework. Partly because it separates parts of the system such as navigation, media and telemetry, IVI increases its tolerance to errors and ensures better safety and experience. Compartmentalizing issues in safety-related vehicles prevents the whole system from failing, thus lowering the possibility of distraction or poor vehicle management. Modularity also makes it possible to update modules online which means each element can be updated or reverted without disturbing the whole device or service [21].

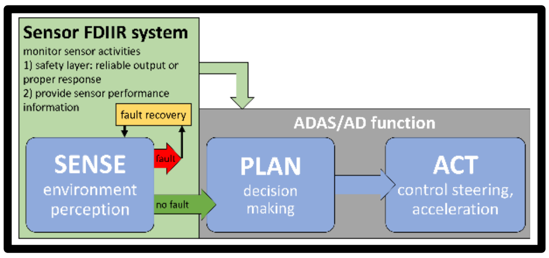
Sensor FDIIR system
monitor sensor activities
1) safety layer: reliable output or proper response
2) provide sensor performance information
fault recovery
ADAS/AD function
SENSE
environment perception
fault
no fault
PLAN
decision making
ACT
control steering, acceleration

Fig 5: Fault isolation in software-defined vehicles

Despite any faults, the framework ensures that infotainment services are not interrupted and stay consistent for the user. Unlike the traditional approach where a single problem often ends in crisis, closed-IVI systems usually offer great user experiences and reliability. Still, it is difficult to use quantum computers in practical situations. Security risks are raised because more communication is happening via IPC which can lead to attacks that require powerful authentication and encryption to deal with. Meeting safety standards, for example with ISO 26262, is challenging, since the framework must show error-free and reliable ways to address faults inside the strict requirements set by those rules [22]. To align with ISO 26262 standards for functional safety, the framework must introduce deterministic behavior under failure conditions, rigorous module validation, formal fault recovery mechanisms, and traceable safety evidence. Lifecycle tracking and static analysis can further support certification by verifying isolation guarantees and fault-handling logic.

The other major advantage of this microkernel- inspired structure is that it leads to increased development agility and vendor cooperation. Third party integration in a traditional IVI system can be quite involving and lengthy because of tightly coupled services that may need to be released together. In comparison, this modular architecture enables OEMs and software providers to develop domain-specific services such as a media vendor providing only update to the audio module without affecting the rest of the system. This isolation speeds up time-to-market of features and enables a competitive plugin ecosystem of SDVs [23].

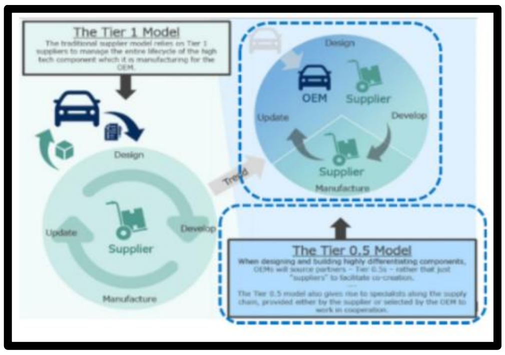
The Tier 1 Model
The traditional supplier model relies on Tier 1 suppliers to manage the entire lifecycle of the high tech component which it is manufacturing for the OEM.
Design
Update
Develop
Supplier
Manufacture
Trend
OEM
Supplier
Design
Update
Develop
Supplier
Manufacture
The Tier 0.5 Model
When designing and building highly differentiating components, OEMs will source partners – Tier 0.5s – rather that just "suppliers" to facilitate co-creation.
The Tier 0.5 model also gives rise to specialists along the supply chain, provided either by the supplier or selected by the OEM to work in cooperation.

Fig 6: Modular SDV Design: Collaboration, Testing & Sustainability

In addition, testability and diagnostics of individual modules are much better. Because services run in isolated run time contexts, developers are able to do unit and integration tests on components in isolation. This isolation not only simplifies debugging, but also enables selective logging and performance analysis, with less of the overhead normally found with full-system testing. Without searching through system-wide logs, it is simpler to attribute performance regressions, memory leaks, and system instability to the bad module [24]. In long-term lifecycle view modular architectures aid sustainability in software maintenance. Automotive systems can be required to be supported beyond 10 years and the capability to replace specific services is useful to mitigate obsolescence, to apply security patches and to use new standards without requiring a complete re-certification. This benefit follows the increasing trend toward green software engineering, where waste minimization in terms of resources with the help of efficient code updates is a welcome characteristic. Concurrent load reliability is a serious problem although these advantages are made. When several modules are in operation at the same time, e.g. real-time telemetry streaming alongside media playback and navigation, shared hardware resources including memory and the CPU become contended. Fault isolation enables graceful degradation of a system but performance bottlenecks can still arise. Mechanisms of adaptive resource scheduling and predictive load balancing might be needed in the future versions of the framework to achieve the consistent performance across domains [25]. Finally, machine learning-based fault prediction can also be integrated into the lifecycle manager to additional reinforce system robustness. Through historical usage pattern analysis and runtime behavior analysis the system can pro-actively restart or patch modules before they can fail in a critical manner. Such active approach would move the system toward fault-predictive instead of fault-tolerant, which is a major step in the evolution of SDV.

With this approach, the system is easier to maintain and can be scaled out because each module can be developed and managed separately. Although it is easy to integrate monolithic systems, their constrained structure makes it hard to fix errors or make changes. For this reason, a microkernel design can help IVIs become more flexible and prepared for new SDV needs. All in all, this study points out that IVI systems built with fault-tolerance could change the automotive software field, but first, they must overcome important security and certification-related challenges.

VII. CONCLUSION AND FUTURE WORK

It proposed a microkernel-based design meant to improve fault tolerance and separation of services for IVIs supported by software-defined cars. The framework deals with the major issues of linked failures and complexity by breaking down navigation, media and telemetry features into independent domains. The flexibility of the system comes from using Android's Services, AIDL and process separations which make it possible to manage modules, their lifecycles and likely faults with service watchdogs and IPC.

This framework distinguishes itself by applying microkernel-inspired principles to in-vehicle infotainment (IVI) systems, enabling the development of modular components that facilitate maintainability during system evolution and ensure robust operational support. Conventional use of plugins isn't like this, because it doesn't have fault isolation at the service level and minor failures may cause trouble for other parts of the system.
This framework demonstrates a structured method for achieving runtime fault tolerance and modular lifecycle management in Android-based SDV IVI systems." Emphasize contributions clearly and conclude by stating its potential readiness for future industry deployment with continued enhancement in security and compliance. By doing this, safety is increased, and users enjoy a better experience which is more important as environments for SDVs get more complex. There are several paths being planned to boost the functionality of the framework. Measures like using containers and hardware-assisted virtualization are being studied to make sure security and fault containment are handled better than with separation between processes alone. Because cybersecurity threats in vehicles are growing, new secure communication channels will be created for use between modules. Moreover, using cloud-enhanced recovery strategies is foreseen which helps keep the system safe from failure and speeds up time until the problem is solved.

Besides the isolation of faults and the enhanced runtime reliability that will have to be demonstrated, the proposed architecture also creates the possibility of advanced monitoring and system intelligence. Including telemetry analytics within the fault management subsystem may enable the vehicle to predict failures before they happen through machine learning or rule-based anomaly detection. Through correlation of usage patterns, system loads and past failure data, proactive mitigation techniques can be also designed and implemented on the fly without involving users. Furthermore, with the trend of in-vehicle systems becoming capable of supporting over-the-air (OTA) updates, the modular nature of the proposed framework may be taken even further by providing automated version management and rollback capabilities per-service. This allows assuring that the upgrade of the navigation or media realms, say, does not jeopardize the integrity or availability of other services. This, combined with a formal verification pipeline to update modules can impose safety constraints even after the system has evolved in use.

The next promising direction of improvement is that of integrating automotive-grade container orchestration platforms, including lightweight Docker alternatives, which would offer better isolation guarantees and support runtime resource assignment. Such implementation can assist in the optimization of the central processing unit (CPU) and memory utilization by module according to the contextual requirements (e.g., giving navigation priority when the vehicle is in motion and media when the vehicle is idle). Considering real-time constraints, a mixed-criticality scheduling approach can also lead to responsiveness and safety benefits, especially when dealing with events of different urgencies occurring simultaneously. This will necessitate the refinement of the existing Android service model, which should be more indicative of temporal needs, as found in safety-critical automotive systems.

Lastly, it will be required to work with the partners in the automotive industry and standards organizations to test the suggested framework in the real conditions of regulations. A longitudinal testing of real vehicular hardware and comparison with the conventional monolithic IVI systems can produce empirical performance and safety standards that would warrant production implementation. The following steps will not only result in a further refinement of this architecture, but will also assist in getting it traction towards adoption in commercial SDVs.

To sum up, the framework supports developing modular and strong IVI designs that respond to changes in automotive software trends. Further improvements on isolation, security and how the system works with cloud services will play a major role in making the prototype ready for mass production.

REFERENCES:

1. Gai, P. and Violante, M., 2016, May. Automotive embedded software architecture in the multi-core age. In *2016 21th IEEE European test symposium (ETS)* (pp. 1-8). IEEE.
2. Ji, D., Zhang, Q., Zhao, S., Shi, Z. and Guan, Y., 2019, August. Microtee: designing tee os based on the microkernel architecture. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)* (pp. 26-33). IEEE.
3. Chen, T., Li, H., Niu, J., Ren, T. and Xu, G., 2019, August. Embedded Partitioning Real-Time Operating System Based on Microkernel. In *2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)* (pp. 205-210). IEEE.
4. Yang, W., Liu, W., Wei, X., Lv, X., Qi, Y., Sun, B. and Liu, Y., 2019, May. Micro-kernel OS architecture and its ecosystem construction for ubiquitous electric power IoT. In *2019 IEEE International Conference on Energy Internet (ICEI)* (pp. 179-184). IEEE.
5. Lampka, K. and Lackorzynski, A., 2019, November. Using hypervisor technology for safe and secure deployment of high-performance multicore platforms in future vehicles. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)* (pp. 783-786). IEEE.
6. Liu, B., Wu, C. and Guo, H., 2021, June. A Survey of Operating System Microkernel. In *2021 International Conference on Intelligent Computing, Automation and Applications (ICAA)* (pp. 743-748). IEEE.
7. Mounir, M., AbdelSalam, M., Safar, M. and Salem, A., 2019, December. Hardware-Assisted Virtualization for Heterogeneous Automotive Applications. In *2019 14th International Conference on Computer Engineering and Systems (ICCES)* (pp. 195-200). IEEE.
8. Silva, M., Cerdeira, D., Pinto, S. and Gomes, T., 2019. Operating systems for internet of things low-end devices: Analysis and benchmarking. *IEEE Internet of Things Journal*, *6*(6), pp.10375-10383.
9. Sivakumar, P., Devi, R.S., Lakshmi, A.N., VinothKumar, B. and Vinod, B., 2020, February. Automotive grade Linux software architecture for automotive infotainment systems. In *2020 International Conference on Inventive Computation Technologies (ICICT)* (pp. 391-395). IEEE.
10. Kovacevic, B., Kovacevic, M., Maruna, T. and Papp, I., 2017. A java application programming interface for in-vehicle infotainment devices. *IEEE Transactions on Consumer Electronics*, *63*(1), pp.68-76.
11. Tierno, A., Santos, M.M., Arruda, B.A. and da Rosa, J.N., 2016, November. Open issues for automotive software testing. In *2016 12th IEEE International Conference on Industry Applications (INDUSCON)* (pp. 1-8). IEEE.
12. Vdovic, H., Babic, J. and Podobnik, V., 2019. Automotive software in connected and autonomous electric vehicles: A review. *IEEE access*, *7*, pp.166365-166379.

13. Bjelica, M.Z. and Lukac, Z., 2019. Central vehicle computer design: Software taking over. *IEEE Consumer Electronics Magazine*, *8*(6), pp.84-90.
14. Sadio, O., Ngom, I. and Lishou, C., 2019. A novel sensing as a service model based on SSN ontology and android automotive. *IEEE Sensors Journal*, *19*(16), pp.7015-7026.
15. Scalas, M. and Giacinto, G., 2019, October. Automotive cybersecurity: Foundations for next-generation vehicles. In *2019 2nd International Conference on new Trends in Computing Sciences (ICTCS)* (pp. 1-6). IEEE.
16. Elkhail, A.A., Refat, R.U.D., Habre, R., Hafeez, A., Bacha, A. and Malik, H., 2021. Vehicle security: A survey of security issues and vulnerabilities, malware attacks and defenses. *IEEE Access*, *9*, pp.162401-162437.
17. Chen, J., Zhou, H., Zhang, N., Xu, W., Yu, Q., Gui, L. and Shen, X., 2017. Service-oriented dynamic connection management for software-defined internet of vehicles. *IEEE Transactions on Intelligent Transportation Systems*, *18*(10), pp.2826-2837.
18. Iorio, M., Reineri, M., Risso, F., Sisto, R. and Valenza, F., 2020. Securing SOME/IP for in-vehicle service protection. *IEEE Transactions on Vehicular Technology*, *69*(11), pp.13450-13466.
19. Kugele, S., Hettler, D. and Shafaei, S., 2018, November. Elastic service provision for intelligent vehicle functions. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)* (pp. 3183-3190). IEEE.
20. Silva, L., Magaia, N., Sousa, B., Kobusińska, A., Casimiro, A., Mavromoustakis, C.X., Mastorakis, G. and De Albuquerque, V.H.C., 2021. Computing paradigms in emerging vehicular environments: A review. *IEEE/CAA Journal of Automatica Sinica*, *8*(3), pp.491-511.
21. Luo, G., Zhou, H., Cheng, N., Yuan, Q., Li, J., Yang, F. and Shen, X., 2019. Software-defined cooperative data sharing in edge computing assisted 5G-VANET. *IEEE Transactions on Mobile Computing*, *20*(3), pp.1212-1229.
22. Bandur, V., Selim, G., Pantelic, V. and Lawford, M., 2021. Making the case for centralized automotive E/E architectures. *IEEE Transactions on Vehicular Technology*, *70*(2), pp.1230-1245.
23. Wulf, C., Willig, M. and Göhringer, D., 2021, August. A survey on hypervisor-based virtualization of embedded reconfigurable systems. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)* (pp. 249-256). IEEE.
24. Syed, T.A., Siddique, M.S., Nadeem, A., Alzahrani, A., Jan, S. and Khattak, M.A.K., 2020. A novel blockchain-based framework for vehicle life cycle tracking: An end-to-end solution. *IEEE Access*, *8*, pp.111042-111063.
25. Arakaki, R., Hayashi, V.T. and Ruggiero, W.V., 2020, June. Available and Fault Tolerant IoT System: Applying Quality Engineering Method. In *2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)* (pp. 1-6). IEEE.