DOI: https://doi.org/10.1109/ITSC.2016.7795766

Robot Operating System: A Modular Software Framework for Automated Driving

André-Marcel Hellmund, Sascha Wirges, Ömer Şahin Taş, Claudio Bandera, Niels Ole Salscheider
FZI Research Center for Information Technology, 76131 Karlsruhe, Germany
[hellmund,wirges,tas,bandera,salscheider]@fzi.de

***Abstract*— Automated vehicles are complex systems with a high degree of interdependencies between its components. This complexity sets increasing demands for the underlying software framework. This paper firstly analyzes the requirements for software frameworks. Afterwards an overview on existing software frameworks, that have been used for automated driving projects, is provided with an in-depth introduction into an emerging open-source software framework, the Robot Operating System (ROS). After discussing the main features, advantages and disadvantages of ROS, the communication overhead of ROS is analyzed quantitatively in various configurations showing its applicability for systems with a high data load.**

I. Introduction

Since the 1980s the research of automated vehicles has made tremendous progress [1] such that this field meanwhile has a thorough theoretic foundation for sensor processing, environment modeling, state estimation, tracking and vehicle control. The interest in automated and cooperative driving research has been boosted by the DARPA Challenges [2], [3] and the Grand Cooperative Driving Challenge [4], respectively. Every new competition thereby sets higher demands on the system complexity enforcing the development of highly adaptable functional system architectures.

From an abstract point of view, an automated vehicle can be considered as a *cognitive agent* and fundamentally consists of a perception module, wherein the input, e.g. images, GNSS positions or LIDAR scans, is used to derive the cognitive decisions of the system, which finally deliver actions, such as actuator commands or C2X messages (see Figure 1). Leaving this top-level view of the system, the internal functional system architecture has a high degree of interdependencies between many functional components [5]. This complex and interdependent structure of the system architecture imposes increasing requirements for the underlying software framework.

The intention of this paper is to shortly review existing software frameworks for automated vehicles with an in-depth introduction into one mature open-source software framework, the Robot Operating System (ROS), that is well established in the robotics community.

The paper is structured as follows: Section II introduces the requirements on software frameworks for automated driving. Section III summarizes two software frameworks that found practice in automated driving, while Section IV elaborately describes one software platform that is intensively used by major automotive manufacturers, suppliers, and research institutes. Section V performs a metric-based evaluation of the communication overhead of the system and Section VI finally concludes this paper and gives a short outlook on improvements for the presented software framework.

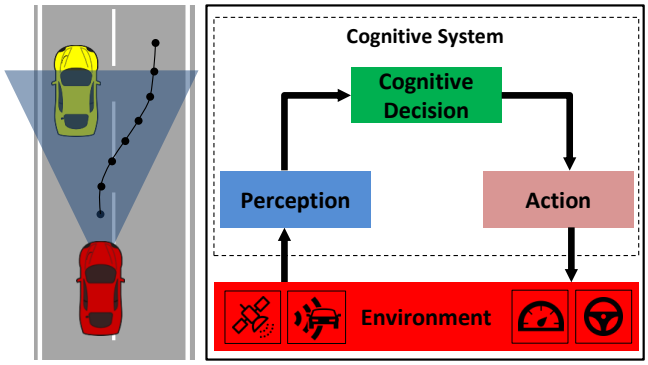

Fig. 1. **High-Level System Architecture for Automated Vehicles.**

II. Design Requirements

Automated driving states special requirements towards the design of Software Frameworks (SWFs). In this section, we introduce requirements that are specifically important to our team.

A. Modularity and Extensibility

Ideally, a SWF is divided into well-defined, independent modules of small size [6]. Due to their functional independence these modules can be tested and evaluated in isolation with low effort. This results in a better maintainability and enables the division of work in groups [7]. In order to achieve modularity, one needs to define common interfaces, which the user can use from a high level of abstraction. Given a modular SWF with well-defined interfaces, functionality can be extended and exchanged easily.

B. Performance

In real-time computing, tasks should be performed in a deterministic fashion with user-acceptable timings. *Hard real-time* SWFs guarantee that each computational response is made at a certain rate with a maximum deviation, i.e. jitter, around the expected response time. In a *soft real-time* SWF only the average response time is guaranteed to be within the defined time interval and jitter. In the case of SWFs for

automated driving at least soft real-time requirements should be fulfilled.

C. Simulation and Debugging

The offline simulation of automated systems is an important, however underestimated aspect in our community to fine tune and debug systems. A software framework should therefore offer a runtime environment and supporting tools to simulate a vehicle at different abstraction levels, from high-level perception down to low-level controlling.

D. Fault Tolerance and Security

The SWF should be able to recover from unexpected failures during runtime, e.g. hardware- or software-related outages. Especially when communicating over channels with noise, jitter and latencies, the SWF should maintain proper execution. To achieve fault tolerance, monitoring concepts, e.g. a redundant voting logic, can be implemented. In addition to fault tolerance the architecture needs to secure the communication towards unauthorized data access or modification.

E. Usability and Support

The SWF's user interface should be easy to use for groups in research and development. Therefore, tools should be provided that simplify user interaction, monitoring and data visualization. It is especially important for non-experts to get documentation, examples or tutorials on how to use the software.

III. Automotive Software Frameworks

Throughout the past decade, various software frameworks emerged in the field of automated driving. The selection or design of a software framework is a crucial step for many research groups. In this section we introduce two well known and widely used software frameworks for automated driving.

A. Real-time Database for Cognitive Vehicles

For the last ten years, the *Real-time Database for Cognitive Automobiles (KogMo-RTDB)* was used as a data exchange provider within our team's research vehicles [8].

Within the centralized KogMo-RTDB architecture data objects can be published at a central place [9]. It provides a unified interface to insert, update and delete objects. Although KogMo-RTDB provides dynamic memory allocation for data objects, memory needs to be allocated statically within the objects. This can be cumbersome in situations where object sizes vary, e.g. measurement data acquired by laser scanning devices. Objects are kept within shared memory for a certain amount of time and can be serialized and written to storage. Within distributed systems, however, KogMo-RTDB is not intended to communicate via network protocols.

KogMo-RTDB is accompanied by tools to record and playback data objects. Unfortunately, the project is currently poorly maintained without a community of active open-source developers.

B. Automotive Data and Time-Triggered Framework

The proprietary *EB Assist Automotive Data and Time-Triggered Framework (ADTF)* is the mostly used development and testing environment for Advanced Driver Assistance Systems (ADASs) today [10], [11]. It is designed as a real-time system with distributed process chains and supports synchronous and asynchronous data processing. Communication between individual process chains can be realized in pipeline, event, or service-call based manner. ADTF is extendible by custom modules, while additional toolboxes such as device drivers or a MATLAB/Simulink interface can be purchased. Like KogMo-RTDB, ADTF also offers tools for recording and playback of data and hence provides rudimentary capabilities to simulate and debug the system offline. Because ADTF is a professionally licensed product, ADTF is mostly used by industrial companies, which is why the community around ADTF is unfortunately limited.

The next section introduces the software framework that is intensively used at our institute and by other international teams [12], [13], [14] for developing ADASs and automated driving functions.

IV. Robot Operating System

The Robot Operating System (ROS) [15], [16] is an open-source software framework supporting the development of complex, but modular systems in a distributed computing environment. While the core components of ROS are highly generic, the primary focus of ROS and its ecosystem is set to the development and research of robots. The performance critical parts of the framework are written in C++, but applications operating on top of the framework may currently be written in C++, Python or Lisp.[1]. The following sections introduce ROS technically, starting with its high-level software architecture down to a low-level view on how information is communicated in the system. After the introduction of the system and application architecture, the design requirements stated in Section II are analyzed by describing various features of ROS.

A. System Architecture

The ROS framework is a multi-server distributed computing environment allowing software applications (referred to as *nodes* or *nodelets* in the following) to communicate across server boundaries and thereby acting as *one* software system (Figure 2). One server in the ROS landscape is dedicated as *Master* which is responsible for application registration and execution as well as running the central parameter storage (*Parameter Server*) and the message logger (*Logging Service*). The satellite servers (*Slaves*) are connected to the *Master* via the local network using the TCP or UDP protocol and are in charge of running further applications to balance the load of the system.

[1]Experimental bindings for other languages are also available.

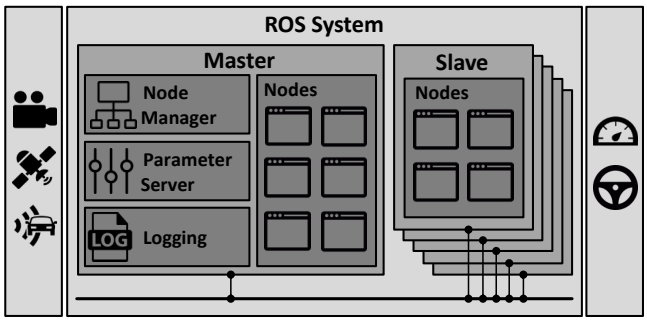

Fig. 2. **ROS System Architecture.** The multi-server ROS system acts as one integrated system for peripheral devices like sensors and actuators.

B. Application Architecture

Applications running in the ROS landscape are called *nodes* or *nodelets*. The distinction between nodes and nodelets is that each node is mapped to a dedicated operating system (OS) process, while nodelets can be grouped inside a single OS process allowing resource sharing between nodelets. The ROS application architecture, independent of the node and nodelet context[2], is shown in Figure 3. Inside the application, the production code is written in either C++, Python or Lisp. The interaction with ROS and other applications running in the system is performed through the ROS interaction layer. The ROS interaction layer thereby provides the following services among others:

- Uni-Directional, Asynchronous Message Exchange
- Bi-Direction, Synchronous Message Exchange
- Diagnostic Message Transmission
- Local and Global Parameter Handling
- Transitive Time-Based Coordinate Transformations
- Criticality-Based Message Logging

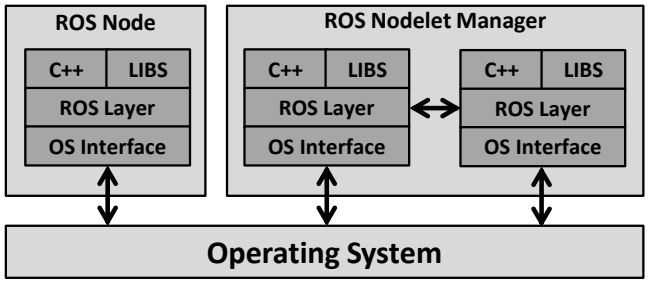

Fig. 3. **ROS Application Architecture.** A ROS node is mapped to a single OS process, while ROS nodelets reside as threads inside a single process called *ROS Nodelet Manager*. The communication between nodelets is accelerated by intra-ROS communications as indicated by the link between the ROS layers. Inter-process communication, e.g. across nodes, is transmitted through the operating system network layers.

C. Modularity and Extensibility

Message Exchange ROS supports two types of message exchanges: synchronous and asynchronous. Because synchronous messages are less commonly used, we will set the focus to asynchronous messages in this section.

[2]The term *node* will hereafter be used for nodes and nodelets.

Asynchronous messaging is implemented inside ROS following the publish-subscribe design pattern as shown in Figure 4. Therefore, a node generating information defines a publisher and registers this publisher to the ROS system, while a node requiring some information instantiates and registers a subscriber. The registered publisher and subscriber(s) are associated to each other by a unique *topic* name.

Message Format The information to be sent and received is declared in a ROS-specific data format (*ROS Message Format*) that supports common primitive data types like boolean flags, strings, integer and floating point numbers as well as user-defined types composed of the primitive types. To support the message exchange across language boundaries, the declared messages are compiled into native classes that allow direct modification of data members. During transmission and reception of data, the compiled classes are automatically serialized and deserialized by the ROS system. Consequently, the sole interface to the data from a programming perspective are the compiled classes. Note, the serialization and deserialization of data messages only occurs for inter-process communication, e.g. when two nodes exchange data. If nodelets communicate with each other, i.e. intra-process communication, the serialization overhead is omitted (see Figure 3), which is why the usage of nodelets is recommended for large data packets, e.g. images or LIDAR scans.

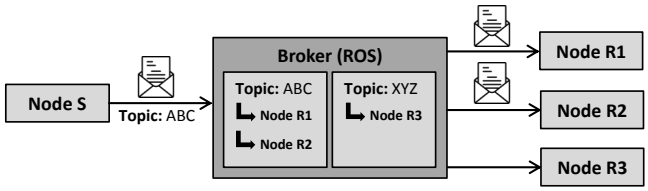

Fig. 4. **Publish-Subscribe Design Pattern.** The broker administers the list of subscribed nodes per topic and distributes incoming messages respectively.

Plug-and-Play Modularity When developing a new system, the decision on the finally employed algorithms is typically postponed until after the prototyping phase. To facilitate the instantaneous replacement of subcomponents, a concise and well established interface between components is required. Inside ROS, this interface is imposed by the *ROS Message Format* introduced above. Every node exactly adhering to the input and output message formats of a component, e.g. an image undistort component expects an image as input and produces an image as output, may be plugged into the system without modification.

D. Performance

Real-Time Capabilities ROS in its current system architecture is not real-time capable due to missing time guarantees for node executions and priority-enforced message transmission. According to the future roadmap,

real-time capabilities for inter-process and inter-machine communication are planned for the next major release, ROS 2.0[3]. To nevertheless get a real-time capable ROS system, the currently claimed strategy proposes to deploy low-level circuits that meet the real-time constraints, e.g. to equip the vehicle with low-level, embedded and specialized hardware to control the vehicle laterally and longitudinally.

Node Pipelining For certain tasks in automated vehicles, computed output data at a constant, predefined rate is beneficial. For example, in vision-based localization with input images recorded at 15Hz, the position updates shall generally be available at 15Hz as well. A concept to achieve this despite possibly longer in total processing times is node pipelining. In node pipelining, one or multiple processing steps are split into smaller chunks that fulfill the desired timing constraints. Figure 5 shows a hypothetical and artificial process chain to estimate the motion of an automated vehicle. To meet the desired update frequency of the vehicle motion estimations, the individual steps are split into separate nodes and chained such that vehicle motion results are available at 15Hz with a dead time equal to the depths of the processing pipeline.

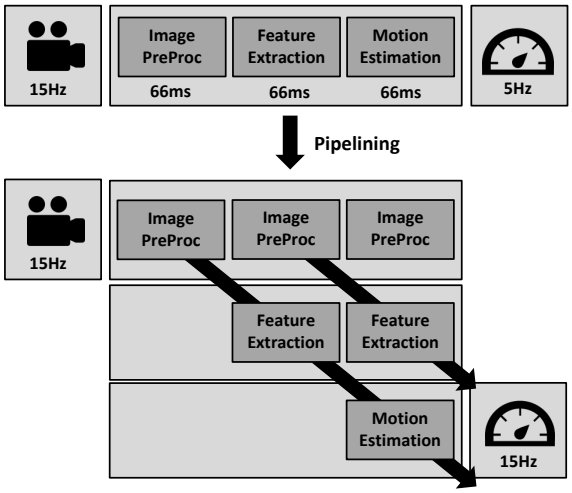

Fig. 5. **Principle of Node Pipelining.** The consecutive process steps residing inside a single node are split into multiple nodes chained together. Afterwards, an end result, e.g. a motion estimation, is available at full sensor speed.

E. Simulation and Debugging

Offline-Processing To simulate and analyze subsystems and to debug critical system errors, it is a fundamental capability of ROS to record a selected subset of topic data transmitted through the system. Figure 6 shows a simplified use case for a visual localization component. In the specific case, the input messages to the localization component are recorded to an offline storage. This offline storage saves the message timestamps as well as the message content for time-accurate replay of the recorded messages. Note, depending on the hardware equipment, especially network bandwidths and storage throughput, all topic messages may be recorded online while the vehicle is driving to analyze critical driving scenarios and decisions, e.g. for trajectory planning and controlling, offline.

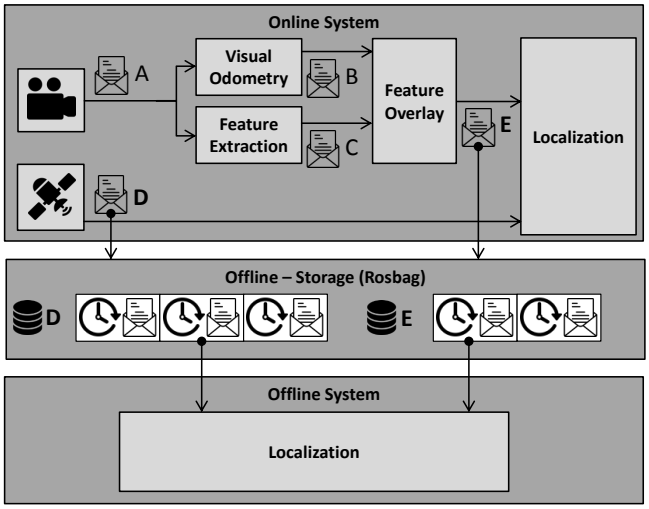

Fig. 6. **Offline Processing.** The message D and E are stored for later processing in a storage container while the system is running online. Each captured message is stored with a respective timestamp to replay the messages in a time-accurate manner. During offline processing, the messages D and E are transmitted back into the system to solely test the localization component.

Simulation For simulation, the modular messaging structure of ROS allows to interchange the source of incoming data. Thereby nodes can easily be tested on recorded data, as explained above, or even on simulated data. If user input is needed within the simulation, the tool *rviz*[4] provides a set of interactive markers that can be used for positioning and orienting obstacles, for example.

For more advanced, whole system simulations or even regression testing, ROS integrates seamlessly with Gazebo. Gazebo is a powerful 3D simulation engine with complete dynamic and kinematic physics [17]. Through a very extensible plugin system, it supports realistic sensor simulation including noise for a wide range of predefined sensors. Custom sensor models can be added through the Gazebo API. Nowadays Gazebo itself is a standalone software, but happens to use the same messaging interface that ROS is build upon, since it originates from the same developers. The ROS nodes under test are therefore connected through the already known ROS messaging interface.

F. Fault Tolerance and Security

System Diagnostics and Monitoring During the development and runtime of a complex system, it is elementary for engineers and users of the system to have an immediate overview of the health of the system, e.g. that all sensors are connected and are sending data into the system, that

[3]More information about ROS 2.0 design decisions is available at http://design.ros2.org/.

[4]*rviz* is 3D visualization tool shipped with ROS, see *Usability and Support* for further details.

all components compute the required data at the predefined rates, etc. To encourage engineers to add diagnostics for their components, ROS provides an easy to use and integrated infrastructure for sending diagnostic messages to a central diagnostic manager that displays the system health in a traffic light color scheme to immediately pinpoint critical components or if data integrity is at risk.

Lock Stepping ROS does not have specific features to inherently implement fault tolerance in the sense of lock stepping systems, however the modular architecture as well as the standardized interface description facilitate the development of lock stepping and voting systems to increase the reliability of the system.

G. Usability and Support

Coordinate Systems and Transformations Inherent to multi-sensor systems is the definition of multiple coordinate systems, e.g. camera, vehicle or global coordinate system, and the frequent need to transform data from one coordinate system into another one. To reduce hard-to-debug transformation bugs, ROS provides a time-based transformation service that keeps track of extrinsic transformations between coordinate systems tagged by a timestamp stating the transformation's validity. To support transitive and bi-directional transformation management, the published transformations are stored in a graph data structure for fast access. To associate message data with coordinate systems, most messages embed the coordinate system name into its payload by using a standardized ROS message header. Figure 7 shows a sample transformation tree with typically found coordinate systems in automated vehicles.

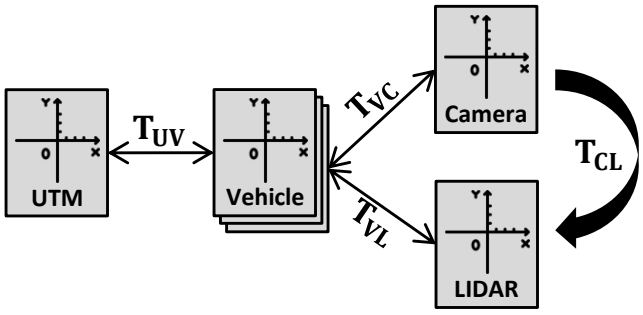

Fig. 7. **Transformation Graph.** The individual extrinsic transformations between coordinate systems as affine 3D matrices are stored in a time-based graph. The position of the vehicle for example is time-variant such that multiple transformations T_{UV} are stored. If the transformation is time-invariant, a static transformation is stored, e.g. T_{VC} between the vehicle and the mounted camera. Given the explicit transformations (thin arrows), implicit transformations, e.g. T_{CL} may be computed on-the-fly.

Multi-Rate Systems Almost all information in sensor-based systems like automated vehicles is typically exchanged at fixed rates, e.g. the maximum rates enforced by the sensors. As an example, camera images are acquired at an update rate of 15Hz, while GNSS position fixes are only available at 1Hz or 5Hz. Data synchronization is hence an important aspect, especially when fusing multiple sensor streams. To simplify the data synchronization in multi-rate systems, ROS provides synchronization primitives (*Message Filters*) based on timestamps, either embedded into the messages or captured during message transmission.

Visualization To provide user feedback and to perform high-level system monitoring, an in-depth visualization of sensor and derived data as well as internal states, e.g. vehicle odometry, are mandatory. For visualization purpose, ROS provides two highly extensible tools (*rqt* and *rviz*) that may be tailored to the systems requirements. Visualization use cases include intra via:

- Image Display
- 3D Plotting (Point Clouds)
- Primitive Markers (Lines, Squares, Ellipses, etc.)
- Vehicle State Data (Odometry)
- Satellite Image Tiles

V. Evaluation

As outlined in Section IV, one of the main advantages of the ROS framework is the communication back-end that facilitates asynchronous data exchange between nodes. Considering the performance of an overall system, the two dominating aspects are the algorithmic computation time and the communication overhead, e.g. latencies. The focus of this evaluation is set to the communication-related metrics of the ROS system, while the computation time of individual nodes is left out of this discussion.

We evaluate the performance of the ROS system by setting up an examplary vision-based processing pipeline, e.g. for visual localization, in a live system as depicted in Figure 8.

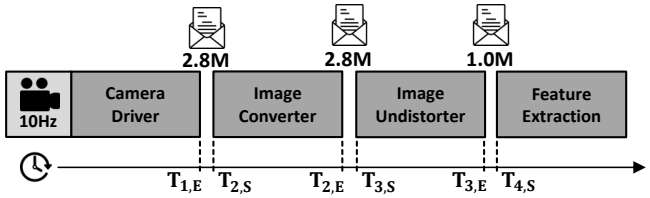

Fig. 8. **Evaluation System.** A camera image is retrieved by the camera driver at 10Hz. Afterwards the image is converted into an 8-bit grayscale image that is undistorted. At the last step, features are extracted from the images, e.g. for visual localization of a vehicle. $T_{i,S}$ and $T_{i,E}$ represent the measured start and finish times of node i.

The pipeline contains three stages, namely image conversion, image undistortion and feature extraction. The amount of data transmitted between the nodes is different as indicated in Figure 8. In order to generate a realistic system load, four Point Grey Flea3 with a resolution of 2.8 MP are connected via Gigabit Ethernet to a server running Ubuntu 14.04. The server is equipped with 12 CPU cores[5] and 64 GB of main memory. The connected cameras are triggered externally with a predefined, fixed update frequency of 10 Hz.

The evaluation of the live system is centered around two metrics: The communication-induced latency and the jitter

[5]Intel Hyper-Threading is enabled during the measurement runs.

of the configured processing pipeline. The communication overhead is measured by recording wall clock times between message transmission at node i and message reception at node $i+1$. The measured times are based on the operating system clocks with an accuracy in the nanosecond range.

In total, we conduct the time-based evaluation for 6 different system loads. Firstly we distinguish between nodes and nodelets to showcase the benefit of intra-process communication without message serialization. Within each group, we evaluate the latency and jitter for three cases: single camera setup, 4 camera setup and 4 camera setup with additional CPU load[6] produced by a CPU stress tool.

A. Latencies

The communication latency is the time overhead caused by the ROS interaction and OS communication layer to transmit a message from one node to the other. In accordance with Figure 8, the latency is measured by:

$$L_{i,i+1} = T_{i+1,E} - T_{i,S} \tag{1}$$

Table I lists the average latencies $\bar{L}_{i,i+1}$ for all three communication paths distinguished by nodes and nodelets. The numbers are averaged over 1000 samples.

Scenario	**Nodes**			**Nodelets**		
Times in [ms]	$\bar{L}_{1,2}$	$\bar{L}_{2,3}$	$\bar{L}_{3,4}$	$\bar{L}_{1,2}$	$\bar{L}_{2,3}$	$\bar{L}_{3,4}$
1 cam	5.274	4.743	0.917	0.139	0.065	0.150
4 cams	3.843	4.571	1.914	0.138	0.074	0.144
4 cams + load	5.666	3.337	43.691	0.115	0.088	0.213

TABLE I
LATENCY EVALUATION.

The numbers show that the latency for larger data packets is in the lower millisecond range if nodes are configured. If the system deploys nodelets however, the transmission latency is reduced down to the sub-millisecond range. The second consequence with regard to nodelets is that the latency remains rather stable even if the system is under heavy load while the transmission times get unpredictable in the node case, e.g. considering $\bar{L}_{3,4}$. Note, the high numbers for $\bar{L}_{3,4}$ are obviously not the net overhead of the transmission, but are caused by scheduling effects due to the CPU stress tool. A more thorough analysis of scheduling effects could be valuable here, but the emphasis in this evaluation is mostly set on the comparison between nodes and nodelets.

B. Jitter

The jitter of periodic signals describes the deviation of the signal arrival time from perfectly aligned, periodic arrival times as shown in Figure 9. For this evaluation, the jitter is computed as the standard deviation of the measured latencies:

$$\sigma_{i,i+1} = \sqrt{\frac{1}{N}\sum_{k=1}^{N}(L_{i,i+1} - \bar{L}_{i,i+1})^2} \tag{2}$$

[6]8 CPUs are kept busy at 100% during the measurements.

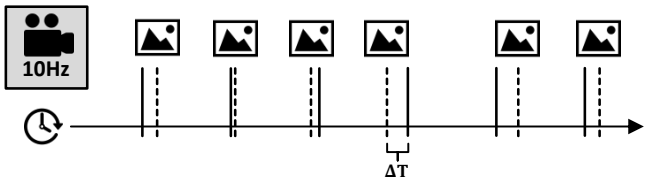

Fig. 9. **Jitter of Periodic Signals.** The dashed vertical lines indicate the perfect signal arrival times, while the solid lines visualize the real arrival times. The deviation of both arrival times is the jitter of the signal ΔT.

Table II lists the jitter $\sigma_{i,i+1}$ for all three communication paths distinguished by nodes and nodelets.

Scenario	**Nodes**			**Nodelets**		
Times in [ms]	$\sigma_{1,2}$	$\sigma_{2,3}$	$\sigma_{3,4}$	$\sigma_{1,2}$	$\sigma_{2,3}$	$\sigma_{3,4}$
1 cam	0.671	0.382	0.091	0.024	0.013	0.098
4 cams	0.632	0.852	1.817	0.087	0.123	0.196
4 cams + load	1.774	3.294	27.180	0.151	0.242	0.432

TABLE II
JITTER EVALUATION.

The numbers confirm the conclusion from the latency analysis that reasonable jitter values are expectable for the nodelet configuration even under high load such that ROS is capable to handle high data load scenarios in automated vehicles.

C. Multi-Server Communication

As introduced in Section IV, a ROS system may be operated across server boundaries with data transmitted via the local network. Table III lists the transmission latencies and the jitter according to Equations 1 and 2 for a simplified variant of the setup in Figure 8. The second node in the pipeline is relocated to a different server such that image data is transmitted via the local network. We distinguish two different scenarios: (1) raw image transmission and (2) compressed image transmission using the JPEG format. The numbers show that transmitting large payloads via the network results in high transmission latencies, however the latency may significantly be reduced in terms of increased CPU consumption by compressing the data before the transmission.

Scenario	**Raw**		**Compressed**	
Times in [ms]	$\bar{L}_{1,2}$	$\sigma_{1,2}$	$\bar{L}_{1,2}$	$\sigma_{1,2}$
1 cam	26.086	5.716	18.805	2.920
4 cams	87.097	4.500	17.137	4.500

TABLE III
LATENCY AND JITTER EVALUATION FOR INTER-SERVER COMMUNICATION.

D. Software Framework Comparison

Given the introduced software frameworks ROS, Kogmo-RTDB and ADTF, Table IV summarizes the capabilities of each framework qualitatively from our perspective with regard to the design and feature requirements in Section II.

Thereby, we evaluate the features that are available out-of-the-box without further development needs. Each category is classified from very good (++) via moderate (o) down to hardly available (--).

Requirement	ROS	RTDB	ADTF
Modularity	++	++	++
Extensibility	++	+	+
Real-Time Performance	-	+	++
Simulation	++	o	o
Debugging	++	+	++
Fault Tolerance	o	-	+
Security	-	-	o
Usability	++	o	+
Support	++	--	+

TABLE IV
QUALITATIVE COMPARISON OF SOFTWARE FRAMEWORKS.

VI. CONCLUSIONS AND OUTLOOK

In this paper we introduced high-level requirements on software frameworks for automated vehicles. That is, we demand a framework to be modular and extendible, to have a low-overhead, to support fault-tolerant development and finally to provide a rich ecosystem of supporting tools. Afterwards we named existing software frameworks, Kogmo-RTDB and ADTF, that have been used in the past for demonstration purpose in research and industry. As the main part, we presented the Robot Operating System as an open-source software framework that our automated vehicles are operated on. Using ROS, we have successfully finished multiple demonstrations involving high data workloads with multiple cameras and controlling a vehicle to follow an offline recorded GNSS trajectory. Recently, we participated in the second Grand Cooperating Driving Challenge (GCDC) and successfully finished the competition as the second best team. Our modular and extendible software framework played an important role in adapting the system to short-term modifications and thereby obtaining the very good end results.

Finally, we evaluated the ROS overhead and analyzed the latency and jitter of periodic data signals flowing through the ROS system. While the configuration with nodes and separate processes for each component provides limited and partly unpredictable results, using nodelets for each component results in highly reasonable timings.

The results have proven that ROS shows its strength in developing applications for automated vehicles, especially for prototyping purpose. On the contrary, our experience has shown that the main flaw lies within the area of real time performance and time guarantees for message transmission. Additionally, inappropriate sizing of the multi-server landscape with data loads close to the limits of the network bandwidth may result in silent package losses. The next main version of ROS, version 2.0, however is supposed to address those issues and provide a system closer to the status quo of real-time processing and hence towards its more wide-spread application in automated vehicles.

REFERENCES

[1] K. Bengler, K. Dietmayer, B. Färber, M. Maurer, C. Stiller, and H. Winner, "Three decades of driver assistance systems: Review and future perspectives," *IEEE Intell. Transp. Syst. Mag.*, vol. 6, no. 4, pp. 6–22, 2014.

[2] S. Thrun, M. Montemerlo, H. Dahlkamp, and Others, "Stanley : The Robot that Won the DARPA Grand Challenge," *Journal of Field Robotics*, vol. 23, no. April, pp. 661–692, 2006.

[3] S. Kammel, J. Ziegler, B. Pitzer, M. Werling, T. Gindele, D. Jagzent, J. Schröder, M. Thuy, M. Goebl, F. von Hundelshausen, and Others, "Team AnnieWAY's autonomous system for the 2007 DARPA Urban Challenge," *Journal of Field Robotics*, vol. 25, no. 9, pp. 615–639, 2008.

[4] A. Geiger, M. Lauer, F. Moosmann, B. Ranft, H. Rapp, C. Stiller, and J. Ziegler, "Team AnnieWAY's Entry to the 2011 Grand Cooperative Driving Challenge," in *IEEE Transactions on Intelligent Transportation Systems*, vol. 13, no. 3, 2012, pp. 1008–1017.

[5] O. S. Tas, F. Kuhnt, J. M. Zöllner, and C. Stiller, "Functional System Architectures towards Fully Automated Driving," in *IEEE Proc. Intell. Veh. Symp.*, 2016.

[6] K. Jo, J. Kim, D. Kim, C. Jang, and M. Sunwoo, "Development of Autonomous Car–Part I: distributed system architecture and development process," *IEEE Trans. Ind. Electronics*, vol. 61, no. 12, pp. 7131–7140, 2014.

[7] ——, "Development of Autonomous Car–Part II: A case study on the implementation of an autonomous driving system based on distributed architecture," *IEEE Trans. Ind. Electronics*, vol. 62, no. 8, pp. 5119–5132, 2015.

[8] M. Werling, M. Goebl, O. Pink, and C. Stiller, "A Hardware and Software Framework for Cognitive Automobiles," in *IEEE Intelligent Vehicles Symposium*. IEEE, 2008, pp. 1080–1085.

[9] M. Goebl and G. Färber, "Interfaces for integrating cognitive functions into Intelligent Vehicles," *IEEE Intelligent Vehicles Symposium, Proceedings*, pp. 1093–1100, 2008.

[10] J. Messner, "EB Assist ADTF Automotive Data and Time Triggered Framework," Tech. Rep., 2015.

[11] K. Hoffmeister, "Automated Driving - Necessary Infrastructure Shift," *ATZ elektronik*, vol. 1, pp. 42–47, 2016.

[12] M. Aeberhard, T. Kühbeck, B. Seidl, M. Friedl, J. Thomas, and O. Scheickl, "Automated Driving with ROS at BMW," ROSCon 2015 Hamburg, Germany. [Retrieved: May 10, 2016]. [Online]. Available: http://roscon.ros.org/2015/presentations/ROSCon-Automated-Driving.pdf

[13] L. C. Fernandes, J. R. Souza, G. Pessin, P. Y. Shinzato, D. Sales, C. Mendes, M. Prado, R. Klaser, A. C. Magalhães, A. Hata *et al.*, "Carina intelligent robotic car: Architectural design and applications," *Journal of Systems Architecture*, vol. 60, no. 4, pp. 372–392, 2014.

[14] D. Martin, F. Garcia, B. Musleh, D. Olmeda, G. Peláez, P. Marin, A. Ponz, C. Rodriguez, A. Al-Kaff, A. de la Escalera, and Others, "IVVI 2.0: An intelligent vehicle based on computational perception," *Expert Systems with Applications*, vol. 41, no. 17, pp. 7927–7944, 2014.

[15] W. Garage, "Robot Operating System (ROS)," 2012.

[16] M. Quigley, K. Conley, B. Gerkey, J. FAust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Mg, "ROS: an open-source Robot Operating System," *ICRA Workshop on Open-Source Software*, vol. 3, no. 3.2, p. 5, 2009.

[17] M. R. Zofka, S. Klemm, F. Kuhnt, T. Schamm, and J. M. Zöllner, "Testing and Validating High Level Components for Automated Driving: Simulation Framework for Traffic Scenarios," in *IEEE Proc. Intell. Veh. Symp.*, 2016.