Communication and Architectural Patterns for Developing Distributed Systems

1st Sebastian Bindick
Group IT
Volkswagen AG
Wolfsburg, Germany
sebastian.bindick@volkswagen.de

2nd Hendrik Kausch
Chair of Software Engineering
RWTH Aachen University
Aachen, Germany
kausch@se-rwth.de

3rd Mathias Pfeiffer
Chair of Software Engineering
RWTH Aachen University
Aachen, Germany
mpfeiffer@se-rwth.de

4th Deni Raco
Chair of Software Engineering
RWTH Aachen University
Aachen, Germany
raco@se-rwth.de
*Corresponding author

5th Amelie Rath
Chair of Software Engineering
RWTH Aachen University
Aachen, Germany
amelie.rath@rwth-aachen.de

6th Bernhard Rumpe
Chair of Software Engineering
RWTH Aachen University
Aachen, Germany
rumpe@se-rwth.de

7th Andreas Schweiger
Embedded Real-Time Software Development
Airbus Defence and Space GmbH
Manching, Germany
andreas.schweiger@airbus.com

[BKP+25] S. Bindick, H. Kausch, M. Pfeiffer, D. Raco, A. Rath, B. Rumpe, A. Schweiger: Communication and Architectural Patterns for Developing Distributed Systems. In: 2025 8th International Conference on Software and System Engineering (ICoSSE), pp. 31-38, DOI 10.1109/ICoSSE65712.2025.00014, IEEE, Apr. 2025.

***Abstract*—This work explores at first various communication patterns, analyzing their strengths and trade-offs for distributed systems. Two architectural patterns are then evaluated with regard to their typical communication pattern and overall methodical approach. The contribution lies in offering insights into how different communication and architectural patterns are applicable for efficiently designing high quality distributed systems.**

***Index Terms*—Distributed System, Communication Pattern, Architectural Pattern, Tight Cohesion, Loose Coupling**

I. Introduction

A. Motivation for Distributed Systems

The need for distributed systems is motivated by a variety of perspectives, where some of which are presented in this section: For **structural reasons**, a local distribution of systems is observed if their subsystems are to be networked for them to fulfill their purpose. For example, in the German healthcare system, the IT systems of different stakeholders located at different locations are connected through the gematik telematics infrastructure. With the recently introduced electronic prescription form, the IT systems of a general practitioner, for example, are connected with those of a pharmacy to prescribe and dispense medication to patients, respectively. As an additional example, the automotive sector can be mentioned, in which via the mobile communications network a connection is established between vehicles and the backend of the manufacturer in order to offer value-added services such as remote checking of the fuel level using a smartphone app.

The increasing **complexity** of IT systems and the functionalities they offer require a corresponding mastery during the development and maintenance. Because of the limitations of human cognitive capacity, this can often no longer be achieved with a monolithic system due to the large size. If a system with a particularly high level of functionality is broken down into manageable subsystems, it can be developed through division of labor.

Due to the size of nowaday's systems, the development is usually carried out in **(locally distributed) teams**. Thus, teams or their members can do development work independently from each other. This means that the response to change requests can be made promptly within short release cycles. Patches can then be delivered as needed. Often, timely patches are required for fixing breaking changes, e.g., compatibility issues, compliance reasons, critical bugs, performance issues, or due to information security reasons. From the perspective of the development organization, agile methods are advantageous in such a context.

Isolated computing nodes with **resource limitations** are oftentimes not suitable for meeting the required performance. For instance, processors or memory modules in embedded systems are limited in their performance or capacity. It can be observed in aircraft or automobiles that individual onboard computers or control devices only perform a dedicated task or

German Federal Ministry for Economic Affairs and Climate Action, AMoBaCoD-Project (Grant No. 20X2201C).

implement a demarcated functional area.

When processing a system's huge amount of tasks, **load balancing** is needed to deal with peak loads. In this case, an overall system is provided which can cope with the maximum expected computing load. If necessary, the computing jobs are distributed among the existing (spare) nodes in order to achieve the required quality of service, i.e. without delay or achieving the agreed availability.

The desired service quality can potentially be affected by the failure of computing nodes. To counteract this deficit, redundancy mechanisms are used to ensure the **availability**. Unlike for load balancing scenarios, synchronization mechanisms are necessary to seamlessly replace failed systems with those that are kept in a waiting state. While for example stateless Web server requests can be handled and delegated to computing nodes with enough resources without the need for synchronization, taking over a previously interrupted task from another computing node requires the synchronization of the (temporarily) computed data to ensure determinism.

B. Contribution

As can be seen in section I-A the scenarios cover a broad spectrum of applications, and so do the required architecture implementations. Thus, there is no one-architecture-fits-all use cases, but for the development of a distributed system a proper architecture needs to be selected carefully to meet the respective requirements. One aspect for driving the selection of the right architecture are communication patterns. Thus, this paper takes into account their direction of communication, timing, structure, number of participants, degree of coupling, and communication strategy. This enables the categorization of architectural patterns. As an example, two architectural patterns are introduced and related to their typical usage of communication patterns.

C. Definition of Distributed Systems

An exact definition of the object under consideration is required for the purposes of this paper. According to [1, p. 373] we distinguish between a physical and a logical distribution:

- **Physically** distributed and mutually independent IT systems have resources exclusively available to them, such as memory, and exchange defined messages with other IT systems via communication systems.
- **Logically** distributed systems consist of a set of concurrent and mutually independent processes (programs), which exchange data (messages) via specified mechanisms for coordinating tasks. In doing so, a process (program) has exclusive access to the variables assigned to it.

When a user interacts with such a distributed system, the distribution transparency is expected [2, p. 11]. This means that the distributed nature of the system is hidden from its users. In addition, such systems are usually scalable [3].

D. Differentiation from Parallel Systems

For the scope of this article we differentiate between distributed and parallel systems. The distinction between the two approaches is based on their architecture and their goal of data processing: Parallel systems use several processors within a single computer or a closely integrated (usually homogeneous) cluster. They execute a program in parallel across all processors involved and use a common control flow (SPMD – single program multiple data). The objective is to simultaneously work on different parts of the same task in order to increase the processing speed [3].

A system can be both parallel and distributed by leveraging parallel processing and distributed architecture. However, the patterns considered in this work focus on just distributed systems, while architectural patterns and concepts for parallel systems are considered in [4], [5].

E. General Architectural Principles

With a clear understanding of distributed systems and their differentiation from parallel systems, we can now shift our focus to the general architectural principles that guide the design of distributed systems. The architectural principles of **tight cohesion** and **loose coupling**, which are described in the following, serve as the basis for the architectural patterns considered in this paper. According to the principle of tight cohesion [6], [7], in structuring units, semantically related calculations are carried out with semantically related data. For example, in object orientation, data and calculation rules are combined in one class. This is often accompanied by the principle of secrecy (data encapsulation, information hiding), with which the implementation of functionality is hidden from its users and only made accessible via a defined interface. These principles enable adequate structuring of the software and independence from the respective implementation.

The interfaces between the structuring units of tight cohesion are ideally characterized by loose coupling [6], [7]. Such a loose coupling is given if there are as few dependencies as possible between structuring units. This ensures that a change in one structuring unit causes as few or preferably no changes in another structuring unit that interacts with the first.

F. Necessity of Suitable Development Methodologies and Languages

In the domain of software development, complexity in systems engineering has long been addressed through modeling. To effectively manage this complexity, modeling languages must provide decomposition and refinement capabilities. Coupled with a well-defined methodology, this enables the efficient development of high-quality distributed systems [8], [9].

Modeling languages such as Unified Modeling Language (UML) [10] or Systems Modeling Language (SysML) [11] are suitable for presenting designs and requirements of a system to be developed. There is a learning curve and additional front-loaded costs associated with Model-Based

Systems Engineering (MBSE). The choice between MBSE and a document-based approach depends on the project context. Developing and maintaining complex models can be time-intensive and may not be warranted for smaller or less complex projects. However, the use of Model-Based Systems Engineering (MBSE) instead of document-based approaches offers a better overview with improved consistency of the model artifacts. In Model-Driven Systems Engineering (MDSE) models are additionally linked to the products (usually software) through generators. This holistic chain enables a more consistent maintenance between model and product. It also reduces the time effort by automating repetitive tasks, which can result in shorter development cycles.

Both UML and SysML are universally applicable. This flexibility often leads to incompatible models in practice due to ambiguities in their interpretation. Also, the analyzability of the models is reduced, since specific statements, for example about the behavior of a subsystem, are not possible. These ambiguities can be eliminated by suitable development methodologies such as SPES [12]. The methodologies provide guidelines for using modeling elements, for example, by restricting them to semantically well-defined constructs. Such methodologies can then be put into industrial practice through tool integrations like SpesML [1]. Formal correctness analysis of SysML models can also be realized in this way [13], which is an appropriate means of meeting ever-increasing operational and information security requirements.

Building on the formal specification language FOCUS [14], the article [15] introduces a semantic basis for developing modular and hierarchical distributed systems in a model-based way. This semantic basis describes that system components have an interface and a description of their behavior. The interface allows communication with other system participants. The behavior can be depicted either as a composition (additional hierarchy level), as a black box specification, or as an automata [16], [17]. Further development of distributed systems, through which correctness is guaranteed, is possible with the principle of *refinement* [18]. This principle allows, for example, the effective development of software product line architectures in the automotive industry [19], the design of energy-efficient cyber-physical systems [20], or the modeling of secure communication systems in aviation [13].

II. Ontologies and corresponding Communication Standards

The concept of loose coupling presented in section I-E increases the independence between communicating entities. However, this increases the necessity for specifying the messages to be exchanged. In particular, a common semantic basis for the content exchanged via the messages is required. Ontologies are suitable for this purpose because they describe a conceptual scheme for a particular domain in a technology-neutral way [21]. Entities are defined as categories with associated properties, which are connected via relations.

For a machine-based processing of ontologies, the W3C Web Ontology Language[2] (OWL) in the current version 2[3] is often used. This is based on Description Logic (DL, [22]), which provides the formal basis for the syntax and semantics. Such an ontology is stored in a machine-readable form in the Semantic Web document format. To simplify machine-based processing, the query language SPARQL[4] (SPARQL Protocol and RDF[5] Query Language) can be used.

An **Interface Description Language** (IDL) is a formal language that is used to describe the structure, methods, and data formats of an interface between software components [23]. These languages are used to ensure that different systems or services can communicate with each other, regardless of the underlying implementations or platforms. To ensure interoperability between different systems, various IDL standards have been established, some of which are listed below:

- **OpenAPI**[6]: A widely used standard for describing RESTful APIs. It enables the automatic generation of documentation, client, and server code and provides testing and validation tools.
- **WSDL (Web Services Description Language)**[7]: An XML-based standard for describing SOAP (Simple Object Access Protocol) web services that specifies the operations and data formats.
- **Protocol Buffers**[8]: Is a serial data format developed by Google that is used to efficiently describe and transfer structured data. It is used as the standard format for the gRPC framework (Remote Procedure Call), among other things.

III. Communication Patterns for Distributed Systems

With ontologies and communication standards in place to ensure a shared understanding of data and seamless interoperability, the focus now shifts to how distributed systems actually operate in practice. Communication patterns for distributed systems describe how different units of a distributed system interact with each other and exchange information. The selection of a suitable pattern depends on the specific use case and significantly influences the efficiency, reliability, and scalability of the entire system [24]. In a complex distributed system, several communication patterns are often combined to achieve the best possible performance. In order to make the different patterns more comparable, criteria for uniform classification are introduced first:

[1]https://spesml.github.io/, accessed on 18/09/2024.

[2]https://www.w3.org/OWL/, accessed on 27/09/2024.

[3]https://www.w3.org/TR/owl2-overview/, accessed on 27/09/2024.

[4]https://www.w3.org/TR/sparql11-overview/, accessed on 27/09/2024.

[5]Resource Description Framework forms the basis of OWL for the representation of information about resources in a graph, see also https://www.w3.org/RDF/, accessed on 27/09/2024.

[6]https://www.openapis.org, accessed on 01/10/2024.

[7]https://www.w3.org/TR/wsdl20/, accessed on 01/10/2024.

[8]https://protobuf.dev, accessed 10/01/2024

- The **direction of communication** refers to the direction of the flow between units of a distributed system. A distinction is made between **unidirectional** (communication takes place in one direction, from the sender to the receiver) and **bidirectional** (communication takes place in both directions between sender and receiver).
- The **temporal coordination** describes whether the communication is synchronous or asynchronous. **Synchronous** means that the sender waits for a response before it continues to work. Thereby, the control flow is blocking. In **asynchronous** communication, the sender sends a message and continues without waiting for a response. The control flow is not blocking.
- The **structure** describes the organization of the communication units. **Centralized** means that a central unit coordinates the communication, while in a **decentralized** structure the communication takes place without central control.
- **Number of participants**: Describes how many senders and receivers are involved in the communication. A distinction is made between **One-to-One**, **One-to-Many** and **Many-to-Many** communication.
- The **degree of coupling** describes the dependency between the systems or components involved. **Tightly coupled** means that the systems are directly dependent on each other, while **loosely coupled** means that the systems can act independently of each other.
- The **communication strategy** describes whether the sender or the receiver initiates the communication. A distinction is made between **push** (the sender initiates the communication and "pushes" the data to the receiver) and **pull** (the receiver initiates the communication to retrieve data from the sender).

In the following, commonly used communication patterns are explained and classified concerning the criteria mentioned above. In addition, fig. 1 compares the communication patterns based on the criteria.

In distributed systems, communication patterns can be combined in various ways to suit the specific requirements of different architectural components. Certain parts of the system may rely on distinct patterns, while in some cases, multiple patterns can be employed within the same component to support different aspects of a single interaction. However, not all communication patterns can be easily combined. For example, merging Request-Response with Publish-Subscribe can be challenging due to their differing timing characteristics. Selecting compatible communication patterns requires careful consideration of each pattern's characteristics, and how well they align with the system's communication needs.

A. Request-Response

The request-response-pattern is a widely used bidirectional communication pattern in which a client (sender) sends a request to a server (receiver) and the server then sends back a response [25]. This pattern relies on pull-based communication, where the client initiates the interaction to retrieve data from the server. Communication usually takes place synchronously, i.e., the client waits for the server's response before continuing its work.

Areas of application include, among other things, web applications (HTTP-based communication between browsers (clients) and web servers), database queries (applications send queries to a database and receive the result of the query) or remote procedure calls (RPC), in which clients call remote procedures (functions) on a server.

The advantages of the request-response pattern are its simplicity and clarity, the good control over the process (the client can decide how to proceed based on the response), and the widespread support in many protocols and libraries.

Disadvantages are the high latency (the client is blocked until the response arrives), the dependency on the server due to the centralized structure (communication is interrupted if the server fails), the difficulties in scalability (the more clients send requests, the more difficult it becomes to cope with the load) and a tight coupling between client and server.

B. Publish-Subscribe

The publish-subscribe pattern is a flexible and asynchronous unidirectional communication pattern in which messages are sent (pushed) from a publisher to one or more subscribers without the publisher knowing the identity of the subscribers [26]. It is based on the idea that messages are grouped according to certain topics or "channels" and subscribers "subscribe" to these topics to receive the associated messages. Usually, there is a central intermediary or message broker (such as Kafka or RabbitMQ) that receives the messages and forwards them to the corresponding subscribers.

Application areas include content-distribution networks such as news or social media portals (information/messages are distributed to all subscribers), IoT applications (sensors send data to a central broker that forwards the data to interested subscribers), and log and monitoring systems (systems such as Elasticsearch, Logstash, Kibana, or Prometheus use publish-subscribe for the distribution of logs and monitoring data).

The advantages of the publish-subscribe-pattern are the loose coupling between publisher and subscriber (they do not need to know each other), the scalability (multiple subscribers can receive messages simultaneously without the publisher having to do any extra work), the asynchronous communication (no direct answers or feedback is necessary) and the high flexibility (new subscribers can easily be added without changing the publishers).

Disadvantages include the difficulty of troubleshooting (since publisher and subscribers are decoupled, it can be more difficult to find errors in the communication chain), the lack of guaranteed delivery (depending on the implementation, messages may be lost or not all subscribers may receive the message), possible delays (in systems with many messages and many subscribers, there may be delays when the broker distributes the messages), and the more complex

	Direction of communication	Timing	Structure	Number of participants	Degree of coupling	Communication strategy
Request-Response	Bidirectional	Synchronous	Central	One-to-One	Tight	Pull
Publish-Subscribe	Unidirectional	Asynchronous	Central	One-to-Many	Loose	Push
Message Queueing	Unidirectional	Asynchronous	Central	One-to-Many	Loose	Pull
Broadcast/ Multicast	Unidirectional	Asynchronous	Central/ Decentral	Many-to-Many	Loose	Push
Peer-to-Peer	Bidirectional	Asynchronous / Synchronous	Decentral	Many-to-Many	Tight	Push / Pull
Event-Driven	Unidirectional	Asynchronous	Central	One-to-Many	Loose	Push

Fig. 1. Classification of Communication Patterns.

synchronization (if several systems or subscribers react to the same data, it can be difficult to synchronize them correctly).

C. Message Queueing

Message queueing is a unidirectional communication pattern in which messages are temporarily stored in a queue before they are processed by the receivers [26], [23]. Here, the sender places a message in a queue without communicating directly with a receiver. The receiver collects the message from the queue (according to the pull principle) and processes it at a later time. The pattern is often used to decouple the communication between different system components and to enable asynchrony by buffering messages and sending and receiving them independently of each other. There is usually a central message queueing system (such as RabbitMQ, Apache Kafka, or Amazon SQS) that receives the messages and puts them in the queue.

Areas of application include load balancing (applications that have to distribute different workloads to several consumers), asynchronous processing (systems that have to process large quantities of messages, e.g. batch processing or job queues), and fault-tolerant systems (since messages are stored temporarily, even if the consumers are temporarily unavailable).

The advantages of the message-queueing-pattern are the high decoupling of sender and receiver (sender and receiver do not have to be active at the same time), scalability (consumers can be added or removed depending on the load in order to cope with the load without affecting the sender), and fault tolerance and reliability (because messages are buffered until they are processed, even if a receiver fails).

Disadvantages are the more complex architecture (the introduction of queues leads to a more complex system architecture, especially in error handling and queue management), latency (delays in processing, especially if the queue is long or consumers are overloaded), and memory and resource management (queues require additional memory and must be configured correctly to manage resources efficiently and not overflow).

D. Broadcast and Multicast

Broadcast and multicast are unidirectional and asynchronous communication patterns in which messages are sent (based on the push principle) from one sender to several receivers [27]. The main difference between the two lies in the target group of the messages: With broadcast, messages are sent to all participants in the network, while multicast only sends messages to a defined group of receivers. Furthermore, broadcast uses a decentralized structure because messages are sent to all nodes in the network without a central control over the receivers. Multicast usually follows a more centralized approach, with a central server managing multicast groups.

Publish-Subscribe can be seen as a specific form of multicast; however, it differs by relying on a broker as an intermediary that manages subscriptions and selectively distributes messages to interested receivers. This approach allows for additional features, such as filtering and persistence, providing more control over the distribution and handling of messages.

Application areas for the broadcast and the multicast include network management (e.g. DHCP requests to all the devices in the network), real-time transmissions for example in streaming applications or in the IoT area (e.g., to send sensor data to a group of recipients).

The advantages of broadcast and multicast are efficiency, scalability, and centralized management. In particular, multicast is an efficient method, since a message is only sent once but is received by several receivers at the same time. This reduces the bandwidth consumption compared to individual point-to-point communication. Multicast supports a large number of receivers without overloading the sender because typically, the network infrastructure takes care of distributing the messages. The sender does not have to establish individual

connections to the receivers, which simplifies the management of communication.

Disadvantages of broadcast and multicast encompass the lack of feedback (neither broadcast nor multicast provide feedback from the receivers to the sender by default, resulting in no guarantee that the messages have been successfully received). Additionally, they can lead to network overload (broadcast, in particular, can overload the network because all nodes must process the messages, even if they do not require them).

E. Peer-to-Peer

Peer-to-peer (P2P) is a communication pattern in which the participants in a network (known as peers) communicate directly with each other and no central authority is required manage the communication [23]. Each peer operates in a decentralized manner and can function bidirectionally as both a sender (client) and a receiver (server), capable of both receiving and sending messages. P2P networks can implement both push and pull models. A peer can actively send data to other peers (push), or peers can request data from other peers (pull). The direct communication between peers leads to a close coupling between the individual participants. The communication can be synchronous or asynchronous, depending on the implementation.

Areas of application for P2P networks include file-sharing systems (e.g. BitTorrent), blockchain technologies (e.g. Bitcoin), VoIP services (e.g. Skype), or distributed computing power (e.g. SETI@home).

The advantages of the peer-to-peer pattern are scalability (P2P architectures can grow with the number of participants without straining central resources), fault tolerance (if a peer fails, another peer can take over the communication), and cost efficiency (the costs for central servers and infrastructure are avoided because the resources are distributed between the peers).

The disadvantages of P2P are performance (the quality of the network can fluctuate since resource allocation depends heavily on the performance and availability of individual peers) and security risks (P2P-systems are more susceptible to security risks such as malware propagation). Another disadvantage is the more difficult management (the lack of a centralized management makes monitoring and regulating the system more difficult).

F. Event-Driven Communication

Event-Driven Communication (EDC) is a pattern in which an unidirectional communication between units in a network is determined by events [28], [26]. Events can represent changes in a system, such as state changes or user input that are communicated to other components. The components in an EDC system react to these events, which are generated by other components, by performing certain actions or triggering further events. In the Event-Driven Communication pattern, an event producer generates an event that is transmitted via an event channel (according to the push principle) and typically serves as a central component. Event consumers react to these events when they arrive and process them. Abstracting from a central component, each component can be event producer and receiver. But in this case, the components must know the interfaces of other components or system parts they wish to communicate wish. Using a central component, other components must only know the central one and only the central component knows the interface of all components. EDC systems are usually asynchronous and loosely coupled because the various components work independently of each other and do not need to communicate directly with each other. In contrast, an example of a non-event-driven distributed system is a batch processing system, where tasks are executed in bulk across multiple nodes at scheduled times or intervals, rather than reacting to real-time events. In this setup, data is collected and distributed among different nodes, but processing occurs only at predefined intervals.

Areas of application for EDC include E-commerce systems (order tracking is often triggered by events such as “order created”, “payment confirmed” or “product delivered”), Internet of Things (sensors generate events such as temperature changes or motion detection, which are then processed by other devices or systems), and financial systems (stock market prices, transactions, and market changes can be processed as events in real-time).

The advantages of EDC are the loose coupling (components are not directly dependent on each other, but only react to events), and the scalability (new components can be added without changing the existing ones).

The disadvantages of EDC are the high complexity (due to the fact that the control flow is not linear and a lot of components work independently of each other), the lack of transaction security (because of the asynchronous communication, it can be more difficult to ensure consistent transaction security across multiple components), and the difficulty of troubleshooting (since the sequence of events is often not deterministic).

IV. Architectural Patterns for Distributed Systems

This section introduces two architectural patterns for designing distributed systems and categorizes them in regards to their typical communication patterns or combinations thereof and the methodical approach.

A. Microservices

Microservices are an architectural pattern that divides a software system into individual, mutually independent services with a clearly definable technical scope (the bounded context) [29]. Each service focuses on a specific, usually small functional task, that can be implemented by a dedicated development team of a maximum of ten people.

Microservices are based on the principle of modularization [30]. The basic idea is to divide a system into modules with limited complexity and clearly defined interfaces. In contrast to traditional modularization techniques, microservices are

structured so that each module runs in its own independent process and communicates through a language-neutral network interface. Each service should have its own data storage, independent business logic, and, if necessary, a user interface. The technology stack (e.g. programming language, database technology, etc.) can be selected individually for each microservice in order to tailor it optimally to the respective use case.

A key advantage of microservices is their independent deployability. Development teams can put individual services into production largely independently of each other, as they are executed in isolation from each other and are only loosely coupled via the network. In addition, coordination efforts between teams are significantly reduced when implementing large software systems, resulting in a faster time-to-market for new features.

However, a microservices architecture also brings challenges with it, particularly in terms of infrastructure. Orchestrating and managing a large number of distributed services that communicate in the network is far more complex than with a monolithic approach, in which all functionalities are bundled into a single large service. Moreover, end-to-end testing of the entire system and ensuring fault tolerance pose particular requirements.

In microservices-based architectures, different communication patterns are often combined with each other to optimally fulfill the requirements of the system. Typical deployed communication patterns are request-response (for synchronous communication between services), publish-subscribe (for asynchronous communication and event-driven communication), and message queueing (for decoupling of services and processing of messages in the queue).

B. MontiBelle Pattern

The development of distributed systems based on the MontiBelle methodology is aligned with the requirements of avionics and is presented in [31]. The methodology not only supports the methodical and model-based specification of distributed systems but also the mapping of the semantics of the system in the theorem prover Isabelle [32], [33]. This then enables a formal correctness verification that accompanies the development process [34]. The integration of domain-specific modeling languages such as SysML or MontiArc is possible and presented in [35], [36] using an avionics case study. In principle, however, the MontiBelle approach is language-agnostic with regard to the selected architecture modeling language [37]. The MontiBelle methodology builds upon the SPES development methodology [12] and utilizes FOCUS citeBS01 as a formal foundation and semantic domain for distributed systems. Its application in other domains therefore is possible in principle and has been demonstrated, for instance, for the automotive industry [38].

A system developed using the MontiBelle methodology is typically characterized by event-driven communication. However, alternative types of communication can also be modeled and defined. The communication procedure can either be modeled directly in the behavior of the components or added to communication channels modularly as separate processing components. Fundamentally, this also enables the creation of diversified systems whose sub-systems can communicate in different ways. Leveraging the modularity and compositionality of the MontiBelle methodology to provide a model library for the different communication patterns described in section III offers practitioners simplified design choices and combination possibilities.

As advantages, the formal basis FOCUS gives the modeled system clear semantics by mapping the models to Focus as its semantic domain [39] and thus enables strong verification and validation for safety-critical systems. Furthermore, starting from an abstract underspecified system architecture of High-Level Requirements (HLRs) allows allocating system parts or components to different development teams. The development teams refine their respective development object while maintaining compliance to their HLRs. This leads to a loose coupling between system components while maintaining high cohesion in strongly connected systemparts over potentially multiple hierarchical system levels. Integration problems do not occurr, since compositionality of refinement, i.e., refining components seperately and then composing them together leads to the refinement of the whole system, is ensured by the use of FOCUS as a formal basis. Leveraging a model-driven approach [40], using a SysMLv2 Profile for formal specification of distributed systems (MontiBelleML [35]), the methodology MontiBelle [41], and a Formal Integrated Development Environment (F-IDE) that encodes MontiBelleML models into Isabelle syntax [42], the MontiBelle pattern allows a hardware, programming language, and tool independent functional design for the distributed system.

As a disadvantage, the need for strictly formal models of the architecture might lead to a longer time-to-market. Following the MontiBelle pattern, starting with abstract formal HLRs, refining and decomposing then into the distributed architecture, and afterwards modeling concrete, close to implementation Low-Level Requirements (LLRs), is an overhead, but leads to formal verification of safety-critical properties. Furthermore, building variants of systems and their formal verification is then faster as most verification and modeling effort was already done and need no, or only few, changes.

V. CONCLUSION AND OUTLOOK

In this paper different communication patterns of distributed systems were compared. The results regarding their direction of communication, timing, structure, number of participants, degree of coupling, and communication strategy was summarized in fig. 1. Afterwards, two architectural patterns were introduced. Both patterns offer insight into the methodical development of distributed systems. The MontiBelle pattern is a model-driven approach and formally specifies distributed systems early on in the design

phase. This enables formal verification of safety critical properties, especially relevant for certification of safety critical system in the avionics industry. The development of Microservice architectures allows through its loose coupling independent development of its components. In comparison, the Microservice approach is less formal early one and lacks formal verification capabilities for the early design phase, but may result in a faster time-to-market.

REFERENCES

[1] K. R. Apt, F. S. Boer, and E.-R. Olderog, *Distributed Programs*. London: Springer London, 2009, pp. 373–406.

[2] M. van Steen and A. S. Tanenbaum, *Distributed Systems*. Maarten van Steen, 2024.

[3] G. Bengel, C. Baun, M. Kunze, and K.-U. Stucky, *Masterkurs Parallele und Verteilte Systeme*. Springer, 2015.

[4] R. Robey and Y. Zamora, *Parallel and High Performance Computing*. Manning Publications, 2021.

[5] T. Weinzierl, *Principles of Parallel Scientific Computing: A First Guide to Numerical Concepts and Programming Methods*. Springer, 2022.

[6] W. P. Stevens., G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, p. 115–139, 1974.

[7] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. New York, N.Y., USA: Yourdon Press, 1975.

[8] F. Fieber, M. Huhn, and B. Rumpe, "Modellqualität als Indikator für Softwarequalität: eine Taxonomie," *Informatik-Spektrum*, vol. 31, no. 5, pp. 408–424, Oktober 2008.

[9] H. Kausch, M. Pfeiffer, D. Raco, B. Rumpe, and A. Schweiger, "Enhancing System Model Quality: Evaluation of the SysML-driven Approach in Avionics," *Journal of Aerospace Information Systems(JAIS)*, vol. 22, no. 1, 2025.

[10] B. Rumpe, *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

[11] Object Management Group, "SysML Specification Version 1.0 (2006-05-03)," August 2006.

[12] Pohl, Klaus and Broy, Manfred and Daembkes, Heinrich and Hönninger, Harald, *Advanced Model-Based Engineering of Embedded Systems*. Springer International Publishing, 2016, pp. 3–9.

[13] H. Kausch, M. Pfeiffer, D. Raco, B. Rumpe, and A. Schweiger, "Correct and Sustainable Development Using Model-based Engineering and Formal Methods," in *2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC)*. IEEE, September 2022.

[14] M. Broy and K. Stølen, *Specification and development of interactive systems: Focus on streams, interfaces, and Refinement*. New York: Springer, 2001.

[15] M. Broy and B. Rumpe, "Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung," *Informatik-Spektrum*, vol. 30, no. 1, pp. 3–18, Februar 2007.

[16] B. Paech and B. Rumpe, "A new Concept of Refinement used for Behaviour Modelling with Automata," in *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, ser. LNCS 873. Springer, 1994, pp. 154–174.

[17] R. Grosu, C. Klein, B. Rumpe, and M. Broy, "State Transition Diagrams," TU Munich, Tech. Rep., 1996.

[18] J. Philipps and B. Rumpe, "Refinement of Information Flow Architectures," in *ICFEM'97 Proceedings*, M. Hinchey, Ed. Hiroshima, Japan: IEEE CS Press, 1997.

[19] A. Haber, H. Rendel, B. Rumpe, and I. Schaefer, "Evolving Delta-oriented Software Product Line Architectures," in *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, ser. LNCS 7539. Springer, 2012, pp. 183–208.

[20] T. Kurpick, M. Look, C. Pinkernell, and B. Rumpe, "Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings," in *Modelling of the Physical World Workshop (MOTPW'12)*. ACM, October 2012, pp. 2:1–2:6.

[21] P. Simons, "Ontology meets ontologies: Philosophers as healers," *Metascience*, vol. 18, no. 3, pp. 469–473, 2009.

[22] F. Baader, D. McGuinness, D. Nardi, and P. Patel-Schneider, *The Description Logic Handbook*. Cambridge, United Kingdom: Cambridge University Press, 2007.

[23] M. V. S. A. S. Tanenbaum, *Distributed Systems: Principles and Paradigms*. Pearson Education, 2007.

[24] S. Bindick, "Effiziente kommunikationsverfahren in microservices-architekturen," *IT Spektrum*, vol. 2022, no. 6, pp. 37–41, 2022.

[25] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, vol. 2, pp. 39–59, 1984.

[26] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison Wesley, 2003.

[27] D. J. W. A. S. Tanenbaum, N. Feamster, *Computer Networks (6th ed.)*. Pearson, 2021.

[28] M. Fowler, "What do you mean by "event-driven"?" *Martin Fowlers Website*, 2017, https://martinfowler.com/articles/201701-event-driven.html, Zugegriffen 2024-10-01.

[29] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2021.

[30] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[31] H. Kausch, M. Pfeiffer, D. Raco, B. Rumpe, and A. Schweiger, "Model-driven Development for Functional Correctness of Avionics Systems: A Verification Framework for SysML Specifications," *CEAS Aeronautical Journal*, vol. 16, no. 1, 2025.

[32] L. C. Paulson, *Isabelle: A generic theorem prover*. Springer, 1994.

[33] J. C. Bürger, H. Kausch, D. Raco, J. O. Ringert, B. Rumpe, S. Stüber, and M. Wiartalla, *Towards an Isabelle Theory for distributed, interactive systems - the untimed case*, ser. Aachener Informatik Berichte, Software Engineering, Band 45. Shaker Verlag, March 2020.

[34] H. Kausch, M. Pfeiffer, D. Raco, and B. Rumpe, "An Approach for Logic-based Knowledge Representation and Automated Reasoning over Underspecification and Refinement in Safety-Critical Cyber-Physical Systems," in *Combined Proceedings of the Workshops at Software Engineering 2020*, R. Hebig and R. Heinrich, Eds., vol. 2581. CEUR Workshop Proceedings, February 2020.

[35] H. Kausch, J. Michael, M. Pfeiffer, D. Raco, B. Rumpe, and A. Schweiger, "Model-Based Development and Logical AI for Secure and Safe Avionics Systems: A Verification Framework for SysML Behavior Specifications," in *Aerospace Europe Conference 2021 (AEC 2021)*. Council of European Aerospace Societies (CEAS), November 2021.

[36] H. Kausch, M. Pfeiffer, D. Raco, A. Rath, B. Rumpe, and A. Schweiger, "A Theory for Event-Driven Specifications Using Focus and MontiArc on the Example of a Data Link Uplink Feed System," in *Software Engineering 2023 Workshops*, I. Groher and T. Vogel, Eds. Gesellschaft für Informatik e.V., February 2023, pp. 169–188.

[37] H. Kausch, M. Pfeiffer, D. Raco, and B. Rumpe, "Model-Based Design of Correct Safety-Critical Systems using Dataflow Languages on the Example of SysML Architecture and Behavior Diagrams," in *Proceedings of the Software Engineering 2021 Satellite Events*, S. Götz, L. Linsbauer, I. Schaefer, and A. Wortmann, Eds., vol. 2814. CEUR, February 2021.

[38] S. Kriebel, D. Raco, B. Rumpe, and S. Stüber, "Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible?" in *Proceedings of the Workshops of the Software Engineering Conference. Workshop on Avionics Systems and Software Engineering (AvioSE'19)*, vol. 2308. CEUR Workshop Proceedings, February 2019, pp. 87–94.

[39] M. V. Cengarle, H. Grönniger, and B. Rumpe, "Variability within Modeling Language Definitions," in *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, ser. LNCS 5795. Springer, 2009, pp. 670–684.

[40] H. Kausch, K. Koppes, L. Netz, P. O'Brien, M. Pfeiffer, D. Raco, M. Radny, A. Rath, R. Richstein, and B. Rumpe, "Applied Model-based Co-Development For Zero-Emisson Flight Systems based on SysML," in *Proceedings of the Deutscher Luft und Raumfahrt Kongress*. Deutsche Gesellschaft für Luft- und Raumfahrt (DGLR), October 2024, p. 10.

[41] H. Kausch, M. Pfeiffer, D. Raco, B. Rumpe, and A. Schweiger, "Enhancing System-model Quality: Evaluation of the MontiBelle Approach with the Avionics Case Study on a Data Link Uplink Feed System," in *Avionics Systems and Software Engineering Workshop of the Software Engineering 2024 - Companion Proceedings (AvioSE)*. Gesellschaft für Informatik e.V., Februrary 2024, pp. 119–138.

[42] H. Kausch, M. Pfeiffer, D. Raco, and B. Rumpe, "MontiBelle - Toolbox for a Model-Based Development and Verification of Distributed Critical Systems for Compliance with Functional Safety," in *AIAA Scitech 2020 Forum*. American Institute of Aeronautics and Astronautics, January 2020.