Programming Languages For Programs For Stateful Distributed Systems

Jochem Broekhoff*

Abstract

In the age of the *Internet of Things* (IoT) emerging, distributed systems are becoming more and more mainstream and relevant, and so are novel programming languages. Traditionally distributed systems, like anything else, have been built from the ground up, tailored for their intended application. However, anything distributed is complex by nature, in stark contrast to what some novel programming languages aim to simplify. By means of a literature review, comparing the languages EdgeC, Distributed Oz, and OpenABL, insight is gathered about whether novel programming languages, intended for distributed applications, are actually meaningful in the relation to IoT-scale distributed systems, answering the main question "reliable execution of programs for distributed systems: are programming languages tailored for distributed applications beneficial in development of such programs?". Primarily Distributed Oz, but also EdgeC stand out in their capabilities. OpenABL on the other hand appears to be less flexible, due to its niche-limited agent-based model. Overall however, these new programming languages appear promising.

1 Introduction

Every day, more and more devices are getting connected, and the end is not yet in sight. The *Internet of Things* (IoT) is most prevalent in this area [1]. Often, these devices need to communicate among each other, in order to achieve some task that they would not be able to perform on their own. For example, an actuator device that can be triggered by another device (which produces events) that is physically separated, but connected through a network.

This is a trivial example, but more complex situations will arise when more devices come into play and depend on each other's state and events. This naturally gives rise to many questions, e.g. regarding synchronization, consistency and speed. Important to note is that for these systems network reachability is key to successful operation (because of the shared state they inherently operate on), and as such they are often not fully *partition-tolerant*, but merely to some extent (depending on many factors). In practice however, we see [2] that network failures are unavoidable, so distributed applications have to take this crucial factor into consideration.

Well-known is the fact that new programming languages are being developed all the time. Some of these are introduced by academics, others arise from enterprise activities and some are mostly community-driven. No new programming language is truly general; there always exist trade-offs because choices have to be made. Some of these languages are specifically designed for application in distributed systems. These are the kind of languages that are discussed in this article.

There are two notable industry-standard programming languages in this field: Erlang/OTP [3] and Elixir [4]. Elixir builds upon Erlang, but is more modernized and has a different feature set. Both languages have existed for numerous years, and are still very general in nature.

In this article, the focus lies on programming languages that are even less general, and more specific. Some of these languages are *domain-specific* languages (DSLs) [5, 6, 7, 8], which are developed with a very specific goal in mind. Namely, a specific domain of application, e.g. in the field of network routing appliances [9]. This is a popular field of study, and research aiding the development of such languages has been active in recent years too [10, 11, 12].

A middle ground between the two extremes of DSLs and general-purpose languages are the programming languages that should still be considered general, but that are developed with the mindset of sole application to distributed applications. General-purpose languages are considered to be languages of the likes of C^{++} and *Java*. From reviewing the literature, the languages EdgeC [13], Distributed Oz [14] and OpenABL [15] are considered in the remaining part of this article. These languages do not classify themselves as DSLs, but their designs are in fact tailored to software for distributed systems.

Considering the fact that from a high-level overview there are already three identifiable classes of programming languages, and that distributed applications are complex by nature, the question rises whether non-general-purpose programming languages may be beneficial. Besides this, focus is lied upon reliability of such programs, which could be defined as 'the program always makes the correct decision and will never make a decision based on too little information'. As briefly stated earlier, and explained in more detail in section 2,

*Delft University of Technology, faculty of EEMCS

this is a crucial for IoT-related applications. Thus the main question of this literature review becomes: *Reliable execution of programs for distributed systems: are programming languages tailored for distributed applications beneficial in development of such programs?*

2 The Problem

The problem described in the introduction is merely a trivial example. In this section, a more complex case is considered which is both in words and visually explained. For those wanting to get more of a feel for the problem and possible implementations: how this particular case might be implemented naively is discussed in section 3.

2.1 Key Concepts and Assumptions

Before describing the problem as a small story, some key concepts are important to know and so are the underlying assumptions. The following description list describes these different aspects.

physical device — A physical device that is capable of one or more of the following: mechanically switching state (e.g. on/off) or taking a snapshot measurement (e.g. relative humidity, amount of people counted on surveillance camera). In practice, such a device may delegate commands to other physical devices, but this must happen transparently and as such whether or not that happens in practice is not relevant.

edge device — A logical device that is located at an 'edge location', capable of interacting with an exclusive set of physical devices. Edge devices are strictly interconnected via a LAN and/or the Internet.

For the sake of keeping the scope limited, assume that, whenever some edge device can reach another edge device across the network, all physical devices attached to the edge device are available and functioning correctly.

action — One of *Read*, *Write* or *Update*. An action is a command sent to a physical device (*Read*, *Write*), or an event produced by a physical device (*Update*). *Read* and *Write* actions are initiated by the program. The program may listen for *Update* actions, but does need not do this. *Update*s that are not listened to by anybody are discarded.

cluster — The logical network of edge device, being interconnected via LAN and/or the Internet.

2.2 Problem Story

Assume there exist (at least) four edge devices, three of which exist together in a LAN. The other edge device is distantly separated and is only connected through the Internet, together forming a cluster. Each edge device has at least one physical device.

The task of this cluster of edge devices is now to evaluate logic atop of the state of their physical devices. Examples of such rules are, one simple and one more involved:

- If device x measures a value of at least 10, then set the value of device y to *on*.
- If the average value of device x (e.g. of the last 60 minutes) is above the current value of device y, then set the value of device z to the measured value from y.

The most important guarantee that the system must fulfill is reliable (consistent) execution. That means that each action must only be taken if there is 100% confidence in doing so, and therefore it should not be incorrect in hindsight. Not fulfilling this could be detrimental to mission-critical IoT applications. Other miscellaneous non-required wishes for such a system could be that introducing a new edge device is seamless and can happen dynamically.

The problem is visualized in figure 1. Here EDx represent the four edge devices. Physical devices are not drawn, but assume that each edge device EDx has some associated. Because ED0 through ED2 are connected to the same router, they participate in the same LAN. Edge locations A and B together form the cluster.

3 Naive Approaches

In order to get an idea of how the problem described in section 2 might be solved in practice, when only generic programming languages are used, three naive approaches will be described next. Of course, even with enough time and knowledge, an efficient and reliable implementation can be made, but this functions merely do demonstrate the point that it is not impossible to achieve the goal with the 'traditional', general-purpose programming languages. Note that these approaches may not all have been implemented in practice, and are partially derived from personal experience of the author.

Of each of the approaches, the downsides are mentioned, but also why one might implement a system that way. This may give an indication as to which problems arise when searching for a non-trivial solution. Please be warned that this is not the way to go for real-world applications.

3.1 Approach #1: on-demand synchronization

This approach functions around a central message broker to which all edge devices connect. Implementation details of such a broker is not relevant. The only requirement is that it guarantees delivery and supports

[1] Advanced Message Queueing Protocol
[2] Message Queueing Telemetry Transport

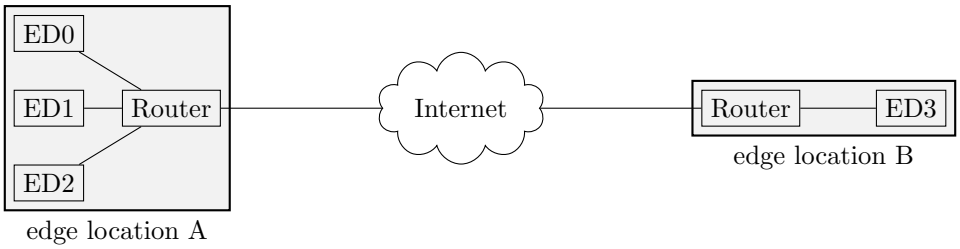

Figure 1: The problem visualized

message retaining. In practice AMQP[1], MQTT[2] and ZeroMQ are often deployed [16, 17, 18, 19].

If-Then-Else (IfE) logic, pure conditional programming, lends itself easily to this approach, because it is a very constrained yet highly structured way of expressing basic logic. However, IfE logic has no state, not even local; the only state exists implicitly in the devices.

A study closely related to this subject [20] describes a similar approach, with a proprietary implementation, based around the idea of sinks, sources and IfE-rules. They instead evaluate the logic not on the edge devices but on a centralized back-end. This shifts even more responsibility to the cloud and makes it practically impossible, with the exception of on-premises deployment, to use this implementation in isolation of the Internet.

Each edge device receives the full program and starts executing their responsible part. How it is determined which edge device is responsible for which part of the program does theoretically not matter, but in practice one may select the edge device which is able to execute as much of that part of the program without having to make requests to the message broker. Then at any time input is expected, be it an explicit *Read* request or a wait for *Update*, or output is to be written (*Write*), the responsible edge device is invoked via the broker. The broker will either respond with a retained message or a fresh response in case the target edge device processed the request.

This approach does not guarantee consistent execution, but that is not straightforwardly obvious and is quite subtle. Primarily the assignment of responsible edge device is problematic. A program consists of multiple parts which are not explicitly tied together, but only by means of a particular *Read*-*Write* (or *Update*) link. Some edge device E_1 may contain parts of the program that are fully executable locally, whereas a second edge device E_2 may contain parts for which it needs to communicate with E_1. If at some point E_2 needs to make a decision based on a value from E_1, a dirty read or write operation may occur if E_1 is actively performing local *Read* and *Write* actions.

3.2 Approach #2: strict replication

Perhaps the easiest solution to solving the problem described in the preceding section, is by means of strict program replication. On each of the edge devices, the same program will run, and all edge devices share the same state of execution, and they all agree on each step of the execution. The execution state consists of for example registers, the stack and the heap.

Only when either a *Read* or *Write* action is to be performed, the capable edge device will perform this action and publish the result to the cluster. Once the entire cluster agrees on the execution of the action, normal program execution may continue.

This approach is not dependent on the language in which the program is developed, as long as there exists some run-time mechanism that implements this approach. Because any general-purpose language suffices, some additional effort ought to be put into developing a library for that language that is used to express the R/W actions. The implementation of that can be delegated to the edge device run-time that participates in the cluster synchronization.

The three downsides of this approach are very significant. First of all, it has a rather high network traffic overhead, because at *each* step during the execution, all edge devices need to know of each other that they agree on the next state of the program. Even though this gives a very strong consistency guarantee, usually this scales in the order of $\mathcal{O}(n^2)$. Therefore, secondly, this model is inherently 100% unable to cope with network failures, as this makes the full synchronization impossible. Lastly, this model also suffers from any latency in the network, since the time it takes to synchronize the cluster on each step takes *at least* twice the largest round-trip time of any two edge devices.

3.3 Approach #3: leader-based single execution

Single execution using a leader-based approach is a slight variation on the preceding approach. This approach is inherently fault-tolerant to some degree, but theoretically also less efficient.

As in approach #2, all edge device participate in a cluster. However, instead of distributing the program to all edge devices and executing simultaneously, only one edge device is appointed to run the program. This is achieved by performing a leader election using a consensus algorithm like Raft [21] or Paxos [22]. The leader executes the program and communicates with the appropriate edge device where actions need to be performed.

Even though there is now a single executor, all other edge devices are always on the outlook for failures and when for example the leader is not reachable anymore. A new leader can be appointed and execution can continue from the last known safe execution point. Such a

point is known, because the leader continuously informs its followers of the progress.

4 Method

To conduct this literature review, publications were gathered by searching through the TU Delft library, about 'programming languages for distributed systems' in the broadest sense. During the search, the keyword 'aspect-oriented' was once used in conjunction with the primary search input, which lead to EdgeC. The results were not explicitly sorted by publication date, but on relevance judged by the library system. Then three publications [13, 14, 15] were filtered out that described an actual language, their implementation, and, most importantly, were strongly related to distributed systems.

Of each of these languages, the focus was determined and key ideas were analyzed. Then for each of them, a conclusion can be drawn to which extent it is capable of solving the described problem. Combined together, a final conclusion can be drawn.

5 Programming Languages For Distributed Systems

Having discussed a problem and some possible, albeit naive, approaches to solving this problem, a selection of programming languages for distributed applications is now discussed. These programming languages are, when compared to the sketched problem, quite general of shape.

Each of the following sections discusses one of the programming languages. The language's key points are first summarized, including explanations of terminology specific to that language, followed by an analysis which elements the language provides that are useful and which elements it lacks. Finally, the language's expressiveness and features are related to the problem from section 2. These observations are summarized in table 1.

5.1 EdgeC

EdgeC, derived as an anagram of 'event-driven distributed global-view consistent executions', is a novel language that allows for distributed application-specific reasoning, building this all around the concept of aspect-oriented programming [13]. The idea of applying the paradigm of aspect-oriented programming to distributed software has already been explored from time to time before [23, 24]. Syntax-wise it is inspired by Scala.

EdgeC by default internally builds on the fundamentals of the actor-based architecture. Key to the EdgeC language is the separation of concerns regarding concrete application logic and the distributed nature of a program. This means that a developer that is responsible for business/application logic can write in EdgeC almost without having to worry about the distributed nature.

Even though the running example used in the EdgeC paper considers a game server/client scenario, it does translate to the problem of driving conditional logic. Each edge device can be represented as a node, bearing some metadata. EdgeC especially excells in the ability to precisely indicate on which particular node some logic should be executed, or on which node state should be stored. Inherent to the DSL is a triggering system, which suits the *Update* action.

Unfortunately, this authors of EdgeC do not consider fault-tolerance at all. However, the EdgeC paper does dedicate a subsection to consistency. Most importantly, by design, the language compiler will analyze the program and critical points are determined (*consistency analysis*). Even MQTT[3], popular in the world of IoT, is mentioned and considered suitable for inter-edge device communication; the EdgeC system implements a mechanism that guarantees reordering. Secondly, the language optimizes for the consistency guarantee to some degree, as to not perform as bad as typical consistency protocol implementations. The authors consider this of high importance, especially for the applications of edge computing. Lastly, but separate from the consistency, EdgeC supports replication to some degree, which can be indicated explicitly, but is managed implicitly.

5.2 Distributed Oz

Oz, and historically one of the first real-world programming languages for distributed systems, Distributed Oz [14], are concurrent object-oriented programming languages. They enables the programmer to express many facets of software for distributed systems in a uniform and logical way. Distributed Oz is semantically similar to Oz, but it extends the syntax to provide constructs which can be used to express facets of a distributed program with failure detection. Even though Oz is concurrent, that does not say anything about suitability for distributed systems. Distributed Oz still makes use of the concurrent model to exploit parallelism on a machine, but adds functionality to make it suitable for distributed systems.

Internally Oz is built around a concurrent constraint computation model, this most notably means that execution order is not linear. The authors therefore also suggest that Oz can be seen as a successor to Prolog.

In contrast to EdgeC, Distributed Oz dedicates extensive effort to fault-tolerance and failure detection. The authors of Distributed Oz do not describe any means of automatic adaptation to a failure scenario. They only account for failures that either exist permanently, causing the program, or parts of the program, to stop execution prematurely, and failures that may fully recover, such as temporary network unavailability. However, this may in practice actually not be a problem as only the part of the program that is interconnected

[3] Message Queuing Telemetry Transport

	EdgeC [13]	**Distributed Oz** [14]	**OpenABL** [15]
architecture basis	aspects and actors	concurrent constraints	agents
compilation targets	Java bytecode	Oz	multi-backend source-to-source
primary development goal	edge computing; separating concerns	separate functionality from distribution structure	distributed agent modelling
considers fault-tolerance?	partially, through consistency guarantees	yes, first-class support	no; irrelevant due to the immutable architecture
other highlights	ability to express where code runs	elaborate algorithm analyses	immutable state, transition-based
suitable for the problem	good; reasonably fault-tolerant	very well	insufficient; requires too many drastic adaptations

Table 1: Different programming languages for distributed applications summarized

with a failing *site* fails. If site B fails, and is connected to site A, site A does not fail, only a subset of its nodes that were actually connected to B fail and propagate the failure to nodes they are in turn connected to.

The paper describes some of the algorithms used to implement features extensively. Each aspect that Distributed Oz introduces on top of Oz is covered and illustrated with examples. The distributed execution model is what the authors describe as a *distribution graph*. The way *manager nodes* are key to the operation of a program may appear to be a significant weakness of Distributed Oz. As the *access structure* only consists of the minimal required amount of nodes, this will not be a problem in practice. Since only the necessary nodes are connected, if any node, even the managing node, fails, the entire program would need to enter an invalid state anyways.

5.3 OpenABL

OpenABL is "a domain-specific language designed for portable, high-performance, parallel agent modeling" [15]. The language is designed to model agent-based simulations, which is significantly different from the more general-purpose mindset of EdgeC and Distributed Oz. An 'agent' is not defined to be anything special in particular, but it is usually used to represent some 'thing' that can perform actions by transitioning from one state to the other. The paper mentions traditional examples of predator-prey models and Conway's *Game of Life*.

This agent-based model does not straightforwardly translate to the described problem. Due to the atomic step nature of OpenABL centered around a master node, no intricate consistency management is required, or it is offloaded to the run-time platform. There appears to be no way to express the explicit relation between edge devices and physical devices in OpenABL. Both can be modeled as a separate agent type, but it is impossible to express computation constraints. Although, it is likely not hard to modify the run-time to account for this.

However, OpenABL explicitly focuses on locality based on Euclidean coordinates. The agents for each edge device are created at an integer position with at least two units of space in between. Each physical device is mapped to an agent that is positioned at most one unit away from the edge device agent to which it belongs. Now the step function for each device shall only interact with agents that are in proximity of a radius of one distance unit. This should, but is not guaranteed to, cause the OpenABL run-time to ensure that operations on physical devices and edge devices happen mostly on the same worker node.

Even though the problem of this article can be adapted to the OpenABL execution model, it is not possible to do that with the default OpenABL implementation. There are two key aspects of the OpenABL implementation that need to be changed significantly: First of all the agent state must be hookable for a different run-time system that can perform the physical actions. Secondly, OpenABL works on the assumption that the model is simulated in finite time followed by persisting the final agent state. However, for our purposes, the run-time would need to simulate the agents indefinitely.

All these changes together are quite dramatic, and result in a significantly different variant of OpenABL than its original authors intended. Even though it appears feasible to implement these necessary tweaks, it is still invasive and appears hacky. The OpenABL language therefore is more applicable to other problems.

6 Conclusion

This literature review aimed to research how programming languages tailored for distributed applications might be beneficial in reliable (consistent) execution of programs for distributed systems. Each of the languages Distributed Oz [14], EdgeC [13] and OpenABL [15] has been explored, their key ideas were discussed and a connection to the problem of driving IoT device interaction logic in a cluster formation.

Now it is not the question which language is 'best', but from the three chosen languages, Distributed Oz ap-

pears to be the most powerful and well-explained and easiest to relate to the problem, standing out because of its dedication to fault-tolerance. EdgeC is mostly on the same line, but lacks constructs for explicit fault-tolerance. OpenABL on the other hand appears to be not really fit for this purpose, purely because significant modifications to the existing implementation would be required in order to morph the problem onto the agent model that OpenABL is intended to be used for. These modifications go so far that it is fair to say that they are merely hacks which are not desirable.

All three of the languages enable programmers to express their problem in a significantly more concise way, because of the new constructs they introduce. This reduces both visual and mental complexity of the software, as it partially abstracts away implementation details of the underlying hardware and network. Yet, the programmer still has, albeit somewhat limited, control over the specific aspects of the distributed nature. Especially compared to traditional naive approaches that a non-knowledgeable developer might take using a general-purpose language, these novel programming languages appear promising and should be seen excellently suitable for the task.

References

[1] Daniele Miorandi et al. "Internet of things: Vision, applications and research challenges". In: *Ad Hoc Networks* 10.7 (2012), pp. 1497–1516. ISSN: 1570-8705. DOI: 10.1016/j.adhoc.2012.02.016. URL: https://www.sciencedirect.com/science/article/pii/S1570870512000674.

[2] Peter Bailis and Kyle Kingsbury. "The Network is Reliable". In: *Commun. ACM* 57.9 (Sept. 2014), pp. 48–55. ISSN: 0001-0782. DOI: 10.1145/2643130.

[3] *Erlang Programming Language*. URL: https://www.erlang.org/.

[4] *The Elixir programming language*. URL: https://elixir-lang.org/.

[5] Tony Clark and Balbir S. Barn. "Domain Engineering for Software Tools". In: *Domain Engineering: Product Lines, Languages, and Conceptual Models*. Ed. by Iris Reinhartz-Berger et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 187–209. ISBN: 978-3-642-36654-3. DOI: 10.1007/978-3-642-36654-3_8.

[6] Øystein Haugen. "Domain-Specific Languages and Standardization: Friends or Foes?" In: *Domain Engineering: Product Lines, Languages, and Conceptual Models*. Ed. by Iris Reinhartz-Berger et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 159–186. ISBN: 978-3-642-36654-3. DOI: 10.1007/978-3-642-36654-3_7.

[7] D. R. Barstow. "Domain-Specific Automatic Programming". In: *IEEE Transactions on Software Engineering* SE-11.11 (1985), pp. 1321–1336. DOI: 10.1109/TSE.1985.231881.

[8] G. Muller et al. "A domain-specific language approach to programmable networks". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 33.3 (2003), pp. 370–381. DOI: 10.1109/TSMCC.2003.817364.

[9] Sukhveer Kaur, Krishan Kumar, and Naveen Aggarwal. "A review on P4-Programmable data planes: Architecture, research efforts, and future directions". In: *Computer Communications* 170 (2021), pp. 109–129. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2021.01.027. URL: https://www.sciencedirect.com/science/article/pii/S0140366421000487.

[10] Lennart C.L. Kats and Eelco Visser. "The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10. Reno/Tahoe, Nevada, USA: Association for Computing Machinery, 2010, pp. 444–463. ISBN: 9781450302036. DOI: 10.1145/1869459.1869497.

[11] *MPS: The Domain-Specific Language Creator by JetBrains*. URL: https://www.jetbrains.com/mps/.

[12] Sven Efftinge and Miro Spoenemann. *Xtext - Language Engineering Made Easy!* 2008. URL: https://www.eclipse.org/Xtext.

[13] Ivan Kuraj and Armando Solar-Lezama. "Aspect-Oriented Language for Reactive Distributed Applications at the Edge". In: *Proceedings of the Third ACM International Workshop on Edge Systems, Analytics and Networking*. EdgeSys '20. Heraklion, Greece: Association for Computing Machinery, 2020, pp. 67–72. ISBN: 9781450371322. DOI: 10.1145/3378679.3394531.

[14] Seif Haridi et al. "Programming languages for distributed applications". In: *New Generation Computing* 16.3 (Sept. 1998), pp. 223–261. ISSN: 1882-7055. DOI: 10.1007/BF03037481.

[15] Biagio Cosenza et al. "OpenABL: A Domain-Specific Language for Parallel and Distributed Agent-Based Simulations". In: *Euro-Par 2018: Parallel Processing*. Ed. by Marco Aldinucci, Luca Padovani, and Massimo Torquati. Cham: Springer International Publishing, 2018, pp. 505–518. ISBN: 978-3-319-96983-1. DOI: 10.1007/978-3-319-96983-1_36.

[16] Jasenka Dizdarević et al. "A Survey of Communication Protocols for Internet of Things and Related Challenges of Fog and Cloud Computing Integration". In: *ACM Comput. Surv.* 51.6 (Jan. 2019). ISSN: 0360-0300. DOI: 10.1145/3292674.

[17] Michele Albano et al. “Message-oriented middleware for smart grids”. In: *Computer Standards & Interfaces* 38 (2015), pp. 133–143. ISSN: 0920-5489. DOI: 10.1016/j.csi.2014.08.002. URL: https://www.sciencedirect.com/science/article/pii/S0920548914000804.

[18] Daniel Happ et al. “Meeting IoT platform requirements with open pub/sub solutions”. In: *Annals of Telecommunications* 72.1 (Feb. 2017), pp. 41–52. ISSN: 1958-9395. DOI: 10.1007/s12243-016-0537-4.

[19] Charilaos Akasiadis, Vassilis Pitsilis, and Constantine D. Spyropoulos. “A Multi-Protocol IoT Platform Based on Open-Source Frameworks”. In: *Sensors* 19.19 (2019). ISSN: 1424-8220. DOI: 10.3390/s19194217. URL: https://www.mdpi.com/1424-8220/19/19/4217.

[20] Debajyoti Mukhopadhyay et al. “A Web-of-Things-Based System to Remotely Configure Automated Systems Using a Conditional Programming Approach”. In: *Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications*. Springer. 2017, pp. 303–311.

[21] Diego Ongaro and John Ousterhout. “In search of an understandable consensus algorithm”. In: *Proc ATC'14 USENIX Annual Technical Conference*. 2014, pp. 305–319.

[22] Leslie Lamport. “The Part-Time Parliament”. In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229.

[23] Wisam Al Abed and Jörg Kienzle. “Aspect-Oriented Modelling for Distributed Systems”. In: *Model Driven Engineering Languages and Systems*. Ed. by Jon Whittle, Tony Clark, and Thomas Kühne. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 123–137. ISBN: 978-3-642-24485-8.

[24] Shmuel Katz. “Rigorous Fault Tolerance Using Aspects and Formal Methods”. In: *Rigorous Development of Complex Fault-Tolerant Systems*. Ed. by Michael Butler et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 226–240. ISBN: 978-3-540-48267-3. DOI: 10.1007/11916246_12.