Resource Awareness in Complex Industrial Systems – A Strategy for Software Updates

Petar Rajković [1], Dejan Aleksić [2], Dragan Janković [1], Aleksandar Milenković [1], Anđelija Đorđević [1]

[1] University of Niš, Faculty of Electronic Engineering, Aleksandra Medvedeva 14, Niš, Serbia
[2] University of Niš, Faculty of Science and Mathematics, Department of Physics, Višegradska 33, Niš Serbia

Abstract

The complex industrial systems consist of many heterogeneous devices running different pieces of software in a connected and layer-organized environment. Software instances in different levels communicate between each other using different protocols and are developed using various technologies. Available storage space and network throughput vary from layer to layer.

When deploying a new version of the software to some device, an update package, which is, in some cases, of significantly higher volume than usual data traffic, needs to be distributed via a network, verified, and stored to the destination device. The old version needs to be backup in case of rollback.

To reduce the impact of the mentioned problems, and to reduce the potential system downtime, we aimed to define the more general deployment approach that could be configured to use the combination of blue-green and canary deployment styles in combination with both shared and local backups.

The main objective of this paper is to highlight the common problems with software updates across multiple layers and to bring the set of recommendations and guidelines for, from the resource awareness point of view, the most effective and the cheapest software updates, with the special focus on the lower levels.

Keywords

Industrial software, IoT nodes, Software deployment strategy, Resource awareness

1. Introduction

The complex industrial systems consist of many heterogeneous devices running different pieces of software in a connected and layer-organized environment [1]. Starting from the layer consisting of sensors and the actuators (in our work we will reference it as IoT layer) [2], through the Edge layer [3][4], via SCADA [5] and manufacturing execution systems (MES) [6] to enterprise resource planning (ERP) [7], all pieces of equipment run the software that needs to be updated from time to time.

The update process itself comes with the risk of diverse potential failures that could leave parts of the system unresponsive, running with unpredictable behavior, or emitting erroneous data. For this reason, the update process must be executed in a highly controllable environment that allows easy and efficient rollbacks in case of flawed deployment is detected.

All software components that are present in the industrial system are usually organized in layers. Layers exchange data with each other using different software protocols. The mentioned facts make the overall software update process a bit more complex than within a standard information system

CERCIRAS WS01: 1st Workshop on Connecting Education and Research Communities for an Innovative Resource Aware Society
EMAIL: petar.rajkovic@elfak.ni.ac.rs; alexa@pmf.ni.ac.rs; dragan.jankovic@elfak.ni.ac.rs; aleksandar.milenkovic@elfak.ni.ac.rs; andjelija.djordjevic@elfak.ni.ac.rs

environment, and every error could lead to serious domino effects [8] [9]. Updating software in one layer could have an effect not only on the targeted device but also on other devices in the same layer as well as on the other layers. I.e., the update that is performed in the device running in Edge level could affect software instances running both in IoT and MES layers.

The additional limitation point is not only the expectation for the highest-possible-performances but also the requirement that software must run using as a small number of resources as possible. The complete system must have a high degree of resource awareness, and both storage space and network bandwidth usage must be carefully planned during the update process in order not to significantly reduce the execution of the running components [10][11].

In this paper we will present the following scenarios and the effects of different deployment configurations:

- Software update in an IoT layer
- Communication protocol update between layers (leading to software update on both sides)
- Complex update including protocol and software changes in an IoT layer
- Backup fails strategy
- Deployment fail/ Rollback strategy
- Rollbacks fail strategy

The main objective of this paper is to highlight the common problems with software updates across multiple layers and to bring the set of recommendations and guidelines for, from the resource awareness point of view, the most effective and the cheapest software updates. The special focus in this paper will be on the lower levels since they request the highest resource awareness level. The upgrade planning for the lower levels is particularly important since it is very easy to use complete free space on the device as well as the network bandwidth during the upgrade process.

2. Related Work

The existing literature offers a wide variety of deployment strategies evaluations and recommendations, but in most of the cases, the research covers software that runs in layers such as MES and ERP. These higher layers deal with a large number of clients transferring a significant amount of data and executing numerous transactions. When defining development strategies for lower levels, the common approaches from the literature are not directly implementable due to the unique limitations.

The most critical points for resource management in lower levels are the storage capacity and the data traffic through the connecting networks. The overall effect is not the same on all layers [12]. I.e., manufacturing execution systems (MES) run in a shop floor environment on devices which processing power is close to standard computers.

For devices running MES or ERP software the storage space is not a critical requirement, but they are usually connected to their server using the wireless network. The wireless networks in the industrial environment could experience different disruptions as the result of operating nearby machines generating high-frequency harmonics as well as different security threats [13]. For MES and ERP client nodes, data package verification and data consistency are the most important points. When deploying a new version of the software to some device, an update package, which is of significantly higher volume than usual data traffic, needs to be distributed via a network, verified, stored to the destination device, and the old version needs to be backup in case of rollback [14] [15]. Next, the Edge layer has the main mission to collect all the data from sensor networks and pass it to the MES. In this case, the proper buffer implementation ensures smooth software upgrades.

All the mentioned layers are highly heterogeneous, with different pieces of hardware running the software instances with diverse category of software. Overall, in the complete industrial system, the type of used devices, their number, the amount of transferred data (per device) could be anything between 1kB and 1GB. To make the complete process more demanding, sometimes devices themselves do not have enough memory to store two versions of the software, thus they would require backup on a different location. This leads to the situation that sometimes is nearly impossible to have an upgrade with no, or at least with very low, downtime [16].

As with every process, a software update could fail due to numerous reasons. In that case, a complete deployment approach or deployment system needs to provide the possibility to roll back to the previous version [17]. The rollback will then take more resources and make the situation even worse, so we need to ensure that system governance successfully goes through the process [18].

To reduce the impact of the mentioned problems, and to reduce the potential system downtime, we aimed to define the more general approach that could be configured to use the combination of blue-green [19] and canary deployment [20] styles in combination with both shared and local backups [21]. This looks like the most promising approach for the IoT level.

3. Testing Environment

The environment we used for testing consists of a set of 100 nodes that contain sensors and actuators, in further text IoT (IoT = Internet of Thing) nodes. The global system overview is shown in **Figure 1**. Generally, each IoT contains a different number of sensors and actuators which count within the node could be anything between a few and 1000.

Sensors within one IoT node could be different, and all of them could run a different piece of software. Sensors could be active either constantly or just in predefined periods. During their operation time, they could collect very heterogeneous data with different sample rates. All these facts make the IoT level very dynamic from the operational point of view and could increase the probability that the complete node went out of a stable state in case of problematic deployments.

The amount of available memory space is usually between 1 and 5 MB per device, which is nearly enough for the necessary software. The nodes in the IoT layer are connected using various methods – ranging from cable network connector to LoRaWan, making the inconsistent environment in terms of connection speed and quality. The most complex situation is with LoRa connected devices since their bandwidth could be in a range of only 10-20 kbps.

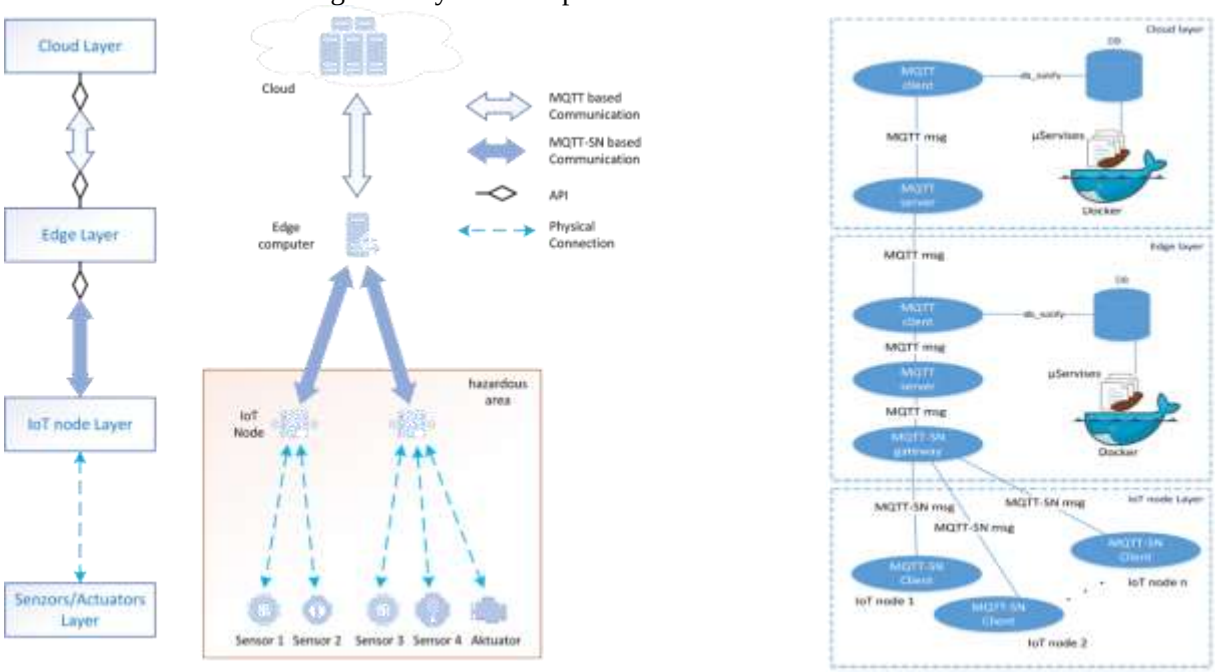

Figure 1: The composition of the examined system

IoT node layers are further connected to Edge computers or Edge nodes. Edge nodes are responsible for communication between the shop floor and hazardous areas on one side and higher levels such as MES and enterprise resource planning (ERP) on the other side. Edge nodes are devices based on Raspberry Pi or similar base sets and are usually connected by a Wireless network with an effective network speed of around 20 Mbps. Their space requirements are around 30 MB per node. There were 10 of these nodes in our test environment.

From the resource awareness point of view, software components on MES and ERP levels are easier to handle. They are running on desktop/laptop computers with enough processing power, disk space, and bandwidth, but even with them, resource planning is inevitable. In our test environment, we used 50 MES clients connected to 2 MES servers (one main and one redundancy), and a similar number of ERP clients connected to the same server configuration (Microsoft Dynamics). All the clients in this level are a few hundred megabytes in volume, but they are located under a gigabyte network.

4. Deployment Strategy for IoT node

The process of a software update for IoT nodes and sensor/actuator devices running in a production environment is considered as particularly sensitive. Small components, both in size and capacity, running in a hazardous environment where the only possible connection are relatively slow LoRa networks with no wiring possible and limited physical access, require detailed planning before an update (**Figure 2**).

Besides the slow network, the low-performance hardware is one additional potential problem. This fact could result in an unacceptable long update process which could move the targeted device off the system for an extended period. The last, but not the least important is the problem of energy consumption. The software update is an activity that requires significantly more energy than the regular data collection and data transmission processes. Thus, this process must be planned for the period when the battery is charged to the highest possible level, and when the eventual rollback will not drain the battery.

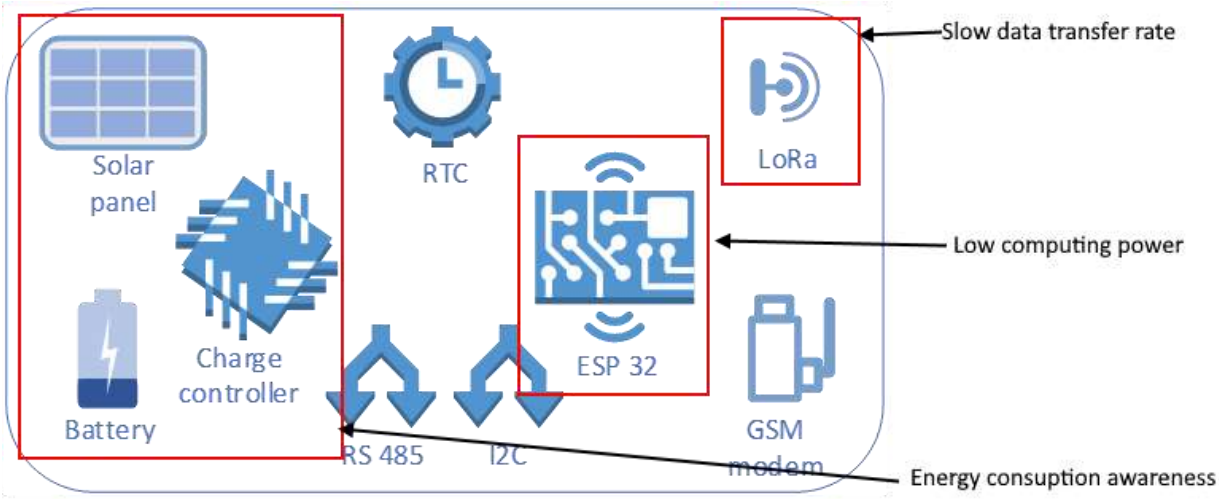

Figure 2: IoT node elements

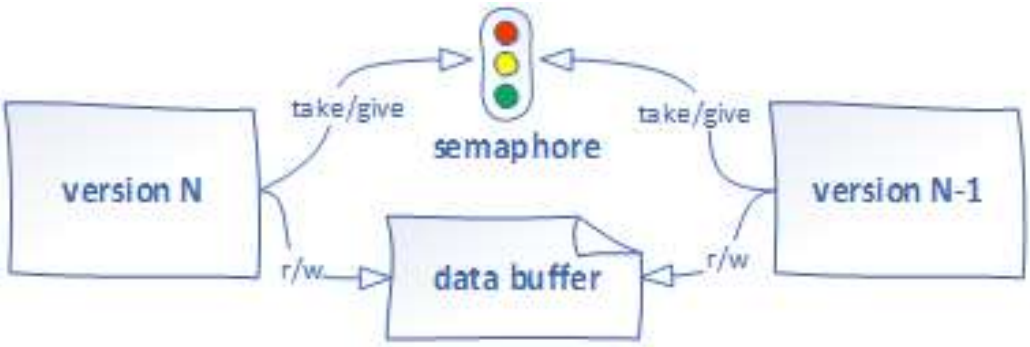

Figure 3: Semaphore-based blue-green deployment strategy

4.1. Software Update Approach for a Single IoT Node

Looking at the single node, our choice for a software update is a semaphore-based green/blue approach (**Figure 3**). This approach is possible with devices that could store at least two versions of the software at the same time. The critical points, in this case, are usually low bandwidth and possibly low battery levels. The approaches to solving these two problems are not in the scope of this paper, and they will be addressed in another future work.

The main idea here is to ensure that the target device always keeps two software versions – actual (version N-1) and previous (version N-2). The update process starts by replacing version N-2, with the new version – version N. At that moment, version N-1 is still active, and the device is running uninterrupted. During that period device experiences higher-than-average network traffic and battery use. Once when version N – 2 is deleted, and version N is uploaded and verified, the switchover could start. The device starts version N, but its communication points are still inactive. When version N is fully up and running, the semaphore opens communication to version N and stops version N-1.

In that case, there is no operation downtime, and the complete update process is seamless for the customer (**Figure 4**). If the process is well-planned there will also be no data loss during the switchover process. In the worst case, only the signals that arrived during the switchover (which usually takes up to several seconds) could be lost and not processed.

This approach is good in case of update errors since it offers an easy way to return to the previous (valid and proven) version N - 1 without the need for immediate additional traffic. Once when the error gets solved, version N could be replaced with the next update. This setup also supports both full and partial version updates, and it is even more suitable for more powerful devices – these that uses GSM modems instead of LoRa adapters.

Usually, newer versions consume slightly higher data space, and the additional challenge that could appear during the software lifecycle is the situation when the new version requires more space than the available in the target device. In this case, the described approach will not be efficient any longer, and the solution must include additional components.

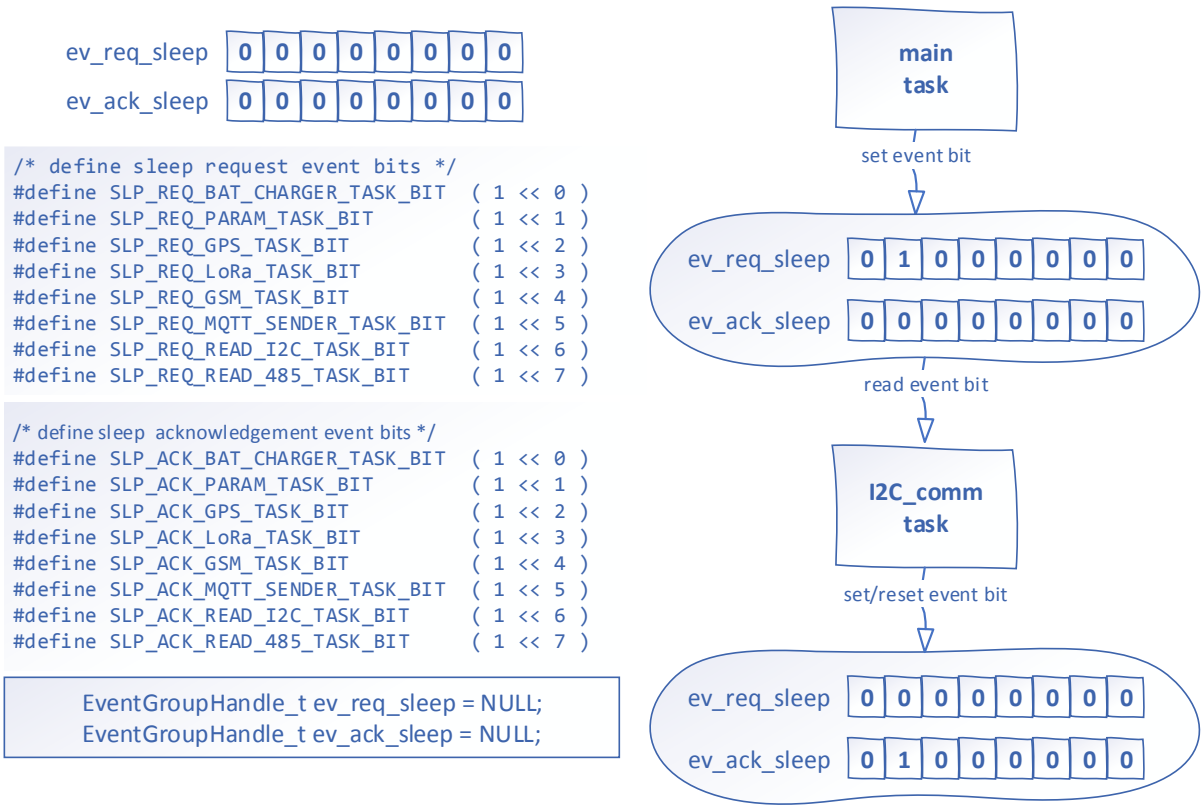

Figure 4: Software update sequence with the sleeping sequence

4.2. Software Update Approach for Devices with Limited Storage Space

As it is mentioned, the more demanding situation is a case when it is not possible to have both versions N and N – 1 copied to the destination device simultaneously. The proposed solution for this problem is to use an additional device of the same type with (if possible) larger storage space – a backup node. The backup node is used to keep the backup versions of the running software. In the situation where the IoT layer consists of multiple similar (or same) nodes, adding one additional device to the system will not be considered as a drawback, but rather as an acceptable small cost.

The deployment process starts with copying the new version (version N) to the backup node. Once this action is finished, the backup node will distribute version N to all devices running the same piece of software. In this situation, the overall downtime will be a bit higher since the target node must stop the previous version (N – 1), get a new one, and then start version N.

With this approach having been implemented, the needed amount of traffic is higher, but this setup has its advantage when comes to potential rollbacks. After version N is uploaded to the backup node, deployment to sensor nodes will go one after another. It will start with the sentinel device (the concept borrowed from the canary deployment), and complete validation in production conditions will be done there. In case the new version is valid, the update of consecutive nodes will follow. If not, the rollback sequence will be done only on the sentinel device.

The update itself in the second scenario does not allow continuous uptime on the device. In such a case, the currently running version (N-1) must be first put to sleep mode and then removed from the destination device. Next, the new version (version N) must be uploaded, configured and then the wake-up command will be applied to version N. Until version N is not started yet, the node will be in downtime and without the possibility to collect and exchange data, which is the potentially unavoidable weak spot.

4.3. Software Update in Edge Layer affecting IoT nodes

The third scenario that will be presented is the effect of the software update in the Edge layer on the nodes in the IoT layer, in the scenario when the device from the Edge level must remain inactive for a period of deployment. In that case, devices at the IoT level will get disconnected for the same amount of time.

The course of events in the IoT node will be as follows:

- Devices in IoT nodes detect disconnection event
- Devices raise the internal alarm
- Start reconnection procedure in predefined time frames

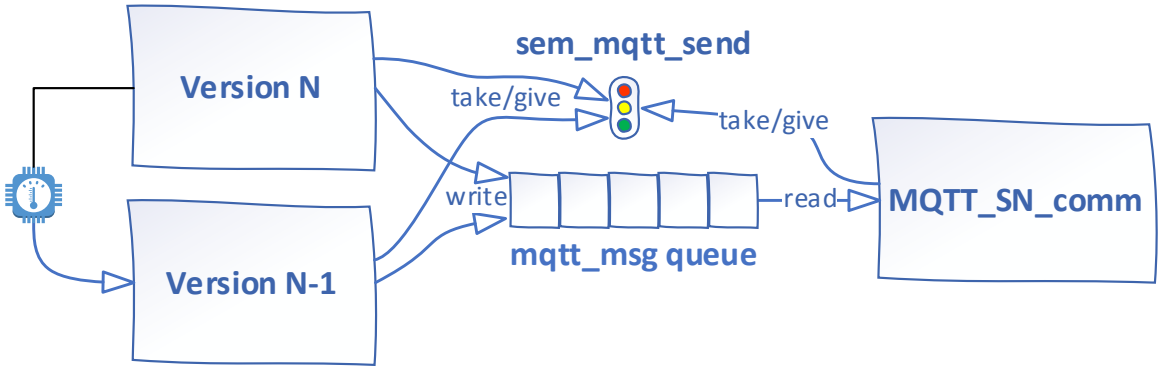

Figure 5: Software update scheme with message queue

While the Edge level node is not running, IoT nodes will have not have a destination where to send processed data. This will cause significant data loss for the complete deployment areas, which could be unacceptable if the process consumes an extensive amount of time. This problematic state will last until the Edge layer node starts running again. When a node from the Edge layer restarts and comes back online, IoT nodes will get connected again and continue to exchange data.

In some cases, IoT nodes will not be able to connect back, either due to the change in communication protocol or to any hardware error. In these cases, IoT nodes will run a general alarm, and then the Edge node must be moved back to the previous version. In case when the update is needed in both layers, the general alarm will be stopped by the update notification signal, and then all IoT nodes will get updated one by one. The update will be driven from the backup node.

One of the commonly used solutions to reduce the necessity for frequent updates across the levels is the using a buffer between the layers (**Figure 5**). In this case, the buffer is implemented as the message queue, and in most of the cases, when the communication protocol gets changed, only the synchronization buffer will be updated while all the nodes in the IoT layer will continue to work. In this way, downtime will hit only one layer (in this case Edge layer) while the other layers will continue to run almost without interruptions.

5. Results and Discussion

Our research was led by the request to reduce the potential downtime during the software update in a challenging environment such IoT layer is. To achieve this goal, we decided to replace the standard deployment (stop-copy-run) with a combination of blue-green and canary deployment strategies, extended with the buffer component. Combining these three well-known approaches in the proposed way, we tried to benefit from all the positive aspects we could get:

- blue-green deployment gives the possibility for a fast version switch
- canary deployment allows prompt identification of deployment errors
- the presence of a synchronization buffer allows us to keep one layer insulated and still operative while the connected layers are in downtime or performing an update.

The proposed approach is initially tested at the IoT level since there we faced the toughest limitations regarding software resources, network bandwidth, and even energy consumption. Another reason to choose this layer for the initial test is also the fact that they are running in critical and hazardous areas. Thus, it is highly requested to reduce the possibility of direct human interaction, or the installation of the additional infrastructural elements such are power or network cables.

To make the situation even more complex, we must note that physical access to the IoT nodes could not be achieved easily. It is often connected not only with technology but with mechanical and security procedures. In some cases, different mechanical elements must be removed to physically reach the device at the IoT level. Furthermore, IoT devices could run in a dangerous environment (for human beings) and then the strict procedure must be followed to access the device itself.

The approach that was used before was the standard update, where the software component was just replaced with the new version – either fully or partly (stop-copy-start). The problems with the standard updates could be summarized as:

- The downtime was always present. If the software component is in the updating process, the software device could not be used
- In case of erroneous update, software should be brought back to the previous version which would lead to the further downtime
- Restore process sometimes could drain the battery which would require that the personnel member must go to the hazardous area
- Connected layers could not continue to work normally since they get flooded with alarm signals

The results we achieved (Table 1) with the proposed combined deployment approach proved our expectation and vary between different software layers and scenarios. Applying the proposed strategy reduced the overall downtime and number of unnecessary rollbacks. This was achieved by the cost of

the implementation of the backup node, the implementation of the buffer level, and by a slight increase in data traffic.

We plan to expand the concept also to other layers and make their update processes as effective as possible. Using backup nodes with the optimized buffering will be the first approach to move to the MES level, and this will be followed with the buffers between Edge and MES as well as between MES and ERP.

Table 1
The effects of the proposed deployment strategy on IoT level containing 50 IoT nodes connected to a single Edge node (TD – time to shut down the software in the node, TU – time to start the software in the node, TS – time switch between the versions, IS – software instance size per node, NN – number of nodes)

Measurement	With standard deployment	With proposed deployment
Number of software uploads to IoT level – successful deployment	NN	1 (only to the backup node)
Number of internal uploads – successful deployment	0	NN
Number of software uploads - unsuccessful deployment	Average 10% of NN	1 to the backup node
Security check on upload	NN	1 (only to backup node)
Number of internal software uploads – unsuccessful deployment	0	1
Rollbacks with unsuccessful deployments	10% of NN	1 + 1
Downtime per node	TD + TU (in seconds)	TS (in milliseconds)
Used space for software per node (with blue-green approach)	1 x IS	2 x IS
Used space for software with buffer node	NN x IS	NN x IS + IS
Update distribution	Manual or with task scheduler	Optimized by backup node
Downtime when connected layer update	If update is running	Until buffer has data

6. Conclusion

With the presented research, we managed to make a significant step forward in the design of the deployment strategy for complex, layer–organized, industrial software systems. When software update must be deployed, the common problems are downtime, network traffic increase, and storage space occupation. In lower levels, even the energy consumption during the deployment process could be an issue.

To reduce the effects of the mentioned problems, especially in the cases when the rollback is needed, we defined the hybrid strategy containing a mix of blue-green and canary deployment supported by inter-layer buffer and backup node. Having this approach implemented we managed to reduce the overall downtime from 50% to close to 0. With the backup node active, we managed to reduce the number of software uploads in case of an erroneous update to less than 1%.

On the other hand, to achieve mentioned results, we added the additional backup node to the system, but since its volume is slightly higher than the volume of regular IoT nodes, we find it acceptable. The results seem promising and for future work, we plan to adapt and extend this approach to the other layers of the complex industrial systems.

7. Acknowledgement

This work is partially supported by CERCIRAS COST Action CA19135 funded by COST.

8. References

[1] Shu, Zhaogang, et al. "Cloud-integrated cyber-physical systems for complex industrial applications." *Mobile Networks and Applications* 21.5 (2016): 865-878.

[2] Kondratenko, Yuriy, et al. "Complex industrial systems automation based on the Internet of Things implementation." *International Conference on Information and Communication Technologies in Education, Research, and Industrial Applications*. Springer, Cham, 2017.

[3] Sha, Kewei, et al. "Edgesec: Design of an edge layer security service to enhance IoT security." *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*. IEEE, 2017.

[4] Li, He, Kaoru Ota, and Mianxiong Dong. "Learning IoT in edge: Deep learning for the Internet of Things with edge computing." *IEEE network* 32.1 (2018): 96-101.

[5] Sajid, Anam, Haider Abbas, and Kashif Saleem. "Cloud-assisted IoT-based SCADA systems security: A review of the state of the art and future challenges." *IEEE Access* 4 (2016): 1375-1384.

[6] Coronado, Pedro Daniel Urbina, et al. "Part data integration in the Shop Floor Digital Twin: Mobile and cloud technologies to enable a manufacturing execution system." *Journal of manufacturing systems* 48 (2018): 25-33.

[7] Chofreh, Abdoulmohammad Gholamzadeh, et al. "Development of guidelines for the implementation of sustainable enterprise resource planning systems." *Journal of Cleaner Production* 244 (2020): 118655.

[8] Cozzani, Valerio, et al. "Quantitative assessment of domino and NaTech scenarios in complex industrial areas." *Journal of Loss Prevention in the Process Industries* 28 (2014): 10-22.

[9] Chen, Yusong, et al. "Research on software failure analysis and quality management model." *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2018.

[10] Usman, Muhammad, et al. "Compliance requirements in large-scale software development: An industrial case study." *International Conference on Product-Focused Software Process Improvement*. Springer, Cham, 2020.

[11] Kalunga, Joseph, Simon Tembo, and Jackson Phiri. "Industrial Internet of Things Common Concepts, Prospects and Software Requirements." *vol* 9 (2020): 1-11.

[12] Chen, Chao, Genserik Reniers, and Nima Khakzad. "A thorough classification and discussion of approaches for modeling and managing domino effects in the process industries." *Safety science* 125 (2020): 104618.

[13] Ren, Zihui, Cheng Chen, and Lijun Zhang. "Security protection under the environment of WiFi." *2017 International Conference Advanced Engineering and Technology Research (AETR 2017)*. Atlantis Press, 2018.

[14] Kim, Dae-Young, Seokhoon Kim, and Jong Hyuk Park. "Remote software update in trusted connection of long range IoT networking integrated with mobile edge cloud." *IEEE Access* 6 (2017): 66831-66840.

[15] Asokan, N., et al. "ASSURED: Architecture for secure software update of realistic embedded devices." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018): 2290-2300.

[16] Mugarza, Imanol, Jorge Parra, and Eduardo Jacob. "Cetratus: A framework for zero downtime secure software updates in safety-critical systems." *Software: Practice and Experience* 50.8 (2020): 1399-1424.

[17] Stević, Stevan, et al. "IoT-based software update proposal for next generation automotive middleware stacks." *2018 IEEE 8th International Conference on Consumer Electronics-Berlin (ICCE-Berlin)*. IEEE, 2018.

[18] Mirhosseini, Samim, and Chris Parnin. "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?." *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017.

[19] Fowler, M. "Blue-green deployment, March 2010." (2016).
[20] Tarvo, Alexander, et al. "CanaryAdvisor: a statistical-based tool for canary testing." *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 2015.
[21] Killi, Bala Prakasa Rao, and Seela Veerabhadreswara Rao. "Towards improving resilience of controller placement with minimum backup capacity in software defined networks." *Computer Networks* 149 (2019): 102-114.