Towards Transforming an Industrial Automation System from Monolithic to Microservices

Santonu Sarkar Gloria Vashi Abdulla PP
ABB Corp. Research India. Email: {santonu.sarkar, gloria.vashi, abdulla.pp}@in.abb.com

***Abstract*—Container technology enables designers to build (micro)service-oriented systems with on-demand scalability and availability easily, provided the original system has been well-modularized to begin with. Industry automation applications, built a long time ago, aim to adopt this technology to become more flexible and ready to be a part of the internet of thing based next-generation industrial system. In this paper, we share our work-in-progress experience of transforming a complex, distributed industrial automation system to a microservice based containerized architecture. We propose a containerized architecture of the "to-be" system and observe that despite being distributed, the "as-is" system tend to follow a monolithic architecture with strong coupling among the participating components. Consequently it becomes difficult to achieve the proposed microservice based architecture without a significant change. We also discuss the workload handling, resource utilization and reliability aspects of the "to-be" architecture using a prototype implementation.**

***Index Terms*—docker, container, migration, industrial system, microservice, modularity**

I. Introduction

Application developers are embracing virtualization in different forms (IAAS, PAAS) to build systems that can scale on-demand, and utilize resources optimally. Virtualization is typically achieved by two commonly known technologies: the hypervisor and the container. Traditionally, an application developer used to rely on hypervisor-based on-demand virtual machine provisioning. However, virtual machines are heavyweight with a full-fledged operating systems having a high memory footprint. Recently, the container technology [1], [12] has emerged and gained massive popularity where one can launch a containerized application, much like launching a virtual machine, with significantly less overhead. Due to the lightweight nature of the container, the overhead of on-demand launching of a new service or a replica of an existing service reduces significantly. As a result, software designers have started embracing containers to design microservice based application [6], [9] where each component of an application, implemented as a microservice, is deployed in a container. If the application is designed well by following the modularity principles, each module becomes an independently compilable and deployable microservice, which can then be horizontally scaled up or down by launching containers corresponding to the module.

Such an architectural pattern can be highly useful for a practitioner if it is easy to migrate an existing application to such an architecture, which is often not the case. In this paper, we focus on a distributed system, which is being used for industrial automation across several domains like automotive, chemical, and electrical plants with a broad customer base. Owners of such a system, running in a plant for a long time, are reluctant for a significant architectural change to transform the system to an on-demand, service-based one if the benefits are not very clear. However, they are willing to try a containerized deployment if the deployment does not require any fundamental change in the design of the existing application.

In this paper, we describe our early experience in migrating this application developed in Windows platform over a period. We started with the objective that we deploy the application as-is in a containerized environment and then analyze whether the original architecture is sufficiently malleable so that the modules of the system can be distributed across multiple containers in a highly decoupled manner. While it is well known that a modularized system with highly decoupled, independently deployable and stateless modules can be easily converted to a microservice, we report the extent of architectural changes necessary to migrate this real-life industrial grade engineering product to an ideal containerized microservice based architecture. We have used the popular Archi® tool[1] to describe the technical and deployment views of the architecture.

II. Related Work

Container-based deployments are gaining attention in both academia and in the industry. The traditional IAAS service providers like Amazon, Google, Microsoft, Oracle, IBM are offering container-based services. An early article by Bernstein [1] introduces the notion of containers and highlights the circumstances in which one should use container based deployment. The report by Pahl et al. [11] discusses how container based deployment becomes the new way to create a PAAS Cloud. Container technology has a profound impact on the software architecture, specifically, it helps in realizing microservices based system design [6], [9]. Practitioners have also performed performance comparison of a virtual machine based deployment vis a vis container based one [2], [8]. There are quite a few attempts in analysing the scaling of services using container-based deployment [5], [10].

The work by [3], [4] are close to ours where the authors have considered industrial PLC applications and shared the experience of containerizing such systems with the objective

[1]https://archimatetool.com/ contains further details of the tool

978-1-5386-7108-5/18/$31.00 ©2018 IEEE

of achieving intelligent and reconfigurable cyber-physical solutions aimed for the next generation industrial automation.

III. Existing Architecture

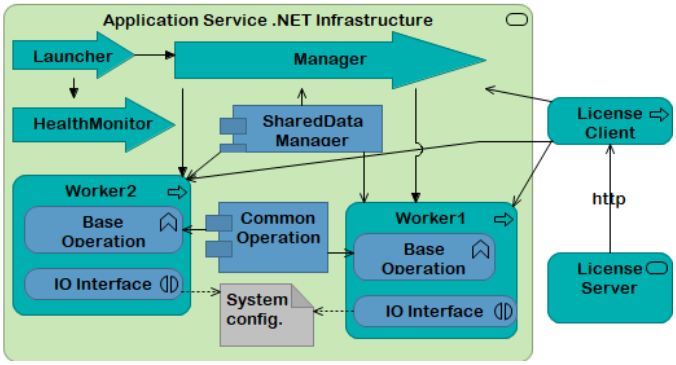

Fig. 1. AS-IS Technical Architecture of the Application

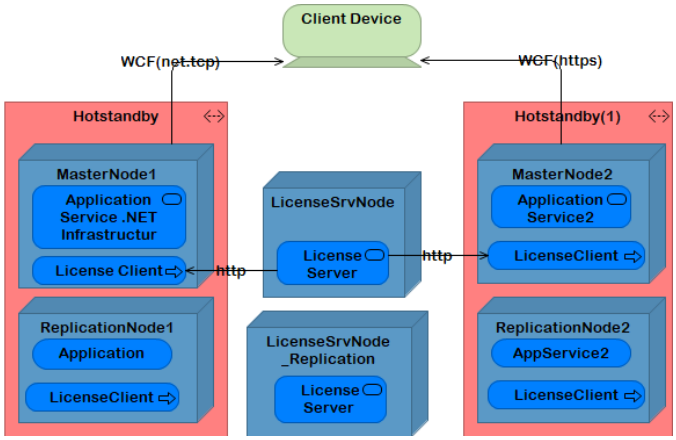

Fig. 2. AS-IS Deployment Architecture of the Application

The industry automation system under consideration has 129 classes spread over nearly 200K lines written in C# and C++. The application has 22 components of different sizes and 30 deployable components (shared libraries and executables). The important runtime components, shown in Figure 1 are:

- A Windows-based master launcher service launches the Manager and the system Health monitor.
- The Manager controls the lifecycle of a set of functional services, deployed in the node where the manager resides. The manager acts as a broker between the end client and the services that the clients wants to consume.
- A functional service, implemented by a Worker, runs as an independent process, and controlled by the Manager. A set of workers exchange data with each other through shared memory. The shared memory is managed by the SharedDataManager component. They also use a third party shared library (CommonOperations module) for common statistical analysis.
- The health monitor application periodically checks the status of the launched functional services and the overall progress of a computation performed by a service.

Despite being distributed, the "as-is" system follows a monolithic architecture [7]. The system is deployed as a set of Windows services in a primary server with a hot-standby backup server, shown in the deployment view in Figure 2. This view indicates that there can be multiple such physical nodes (each with its standby server), each of which hosts a manager and a set of worker processes. The system uses a legacy license server that uses Windows DCOM protocol.

A. Containerization Challenge

Most of the reported efforts in implementing microservice based containerization uses Linux as the underlying operating system. Since the docker based containerization relies mostly on Linux LXC and `cgroup`, it is natural that docker ecosystem is more up-to-date for Linux based platform. Docker on Windows OS is still not as mature as the Linux counterpart and the community support for Windows-based applications is lacking compared to a Linux platform. New capabilities related to Docker-based containerization are continued to be released in Linux first. The non-availability of explicit checkpointing of a container in Windows platform is an important case in point.

- Container Image Creation: The software has been packaged with the Windows-based GUI installation (MSI) infrastructure. If the windows MSI installer is copied inside a windows server container, the installation obviously fails as the installer expects a GUI based interaction. One may be successful in running the installer (msiexec.exe) in a command line mode inside the docker certified Windows container provided the original installation process has been designed with a command line option in mind. Practitioners have reported this issue and suggested tricky roundabout solutions[2].
  Another approach would be to operate from the source code and build the product while building the container image. This however would require access to the source code and modification of the product build process, which violates our initial assumption of not changing the product at all.
- Coupling: While the system has been designed with distributed architecture in mind, the worker and the manager components are strongly coupled as they exchange data through shared memory. The set of workers deployed in a node also also uses a shared library provided by a third party tool. Such a strongly coupling makes it difficult to deploy the manager and the workers in different containers.
- State: An important tenet of the microservice architecture is that a microservice (deployed in a container) is stateless [7]. The statelessness makes the system highly scalable, and reduces the recovery time to a great extent. The current system under consideration performs computation which sometimes depend on the states of the

[2]https://github.com/docker/docker.github.io/issues/4668
https://github.com/Microsoft/iis-docker/issues/13
https://github.com/moby/moby/30395

set of workers managed by a manager. It is possible to containerize the system by deploying each service in a separate container if we compromise on the scalability and reliability benefits to some extent.

- Communication: The current system uses WCF and COM-based communication among the processes. While WCF and COM support traditional distributed communication (WCF is the more advanced with an additional authentication and security options), these are still based on the traditional request-response based blocking calls. In the ideal microservices based architecture [7], it is desirable to have asynchronous message-based non-blocking communication which not only reduces coupling but also achieves scalability and fault-tolerance.

IV. Proposed Architecture

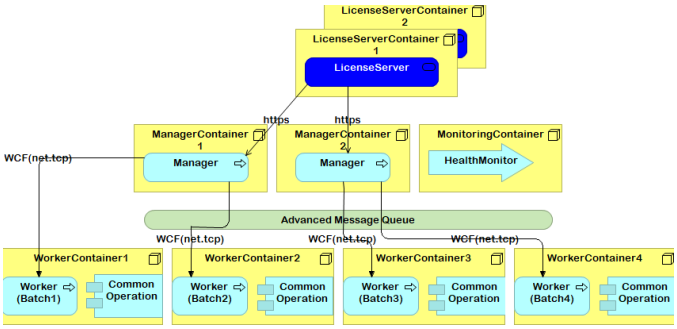

Fig. 3. Proposed (Containerized) Technical Architecture

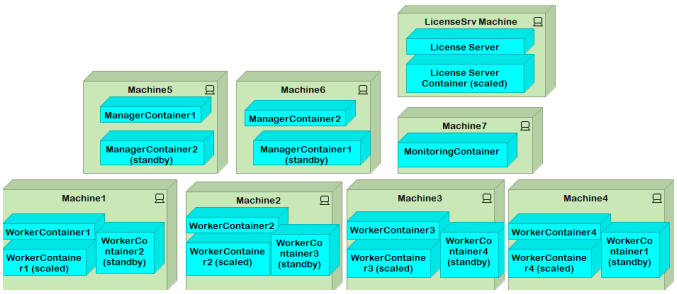

Fig. 4. Proposed (Containerized) Deployment Architecture

In the current work we have developed a workaround to put the entire application in a single Windows server container with additional `dotnet` libraries. The Dockerfile snippet for this container is as follows.

```
FROM microsoft/iis:10.0.14393.206
#add user to the image
RUN ["NET USER","AppUser","passwd","/ADD"]
#Download VC++ 2010 SP1 Redistributable Package (x86)
....
#Download VC++ 2012 Update 4 Redistributable Package (x86)
.....
# Download VC++ 2013 Redistributable Package (x86)
......
# Download VC++ 2015 Update 3 Redistributable Package (x86)
......
#RUN Installer inside container
RUN Start-Process msiexec.exe -ArgumentList @('/i',
'"Installer.msi"','/qn','/lv "$env:temp\InstallLog.txt"',
'norestart','USER="user"','PASSWORD="pwd"','SILENT=TRUE') -Wait
```

However, we also propose an ideal partitioning of the system into a set of microservices as shown in Figure 3 and Figure 4 by containerizing each executable that was originally designed as Windows services. An effort is underway to address the challenges highlighted earlier to achieve the desired partitioning. In what follows, we describe the noteworthy characteristics of the TO-BE architecture.

A. Architectural views

The technical view of the proposed new architecture is shown in Figure 3. Here we propose the following:

Coupling: The new system is decoupled by making the Worker, the Manager, the HealthMonitor independently deployable components, running in separate containers, as microservices. The worker is bundled with the third party library it needs for its functioning. The manager and worker continues to communicate through WCF mechanism. We have been able to create a customized container image for the license server which can perform a COM-based interaction within a container.

State: We propose to modify the inter-worker and manager-worker data sharing mechanism. Instead of shared memory, we propose a AMQP[3] based persistent message queue to exchange data between the component. However, this requires a significant modification of the existing system.

Installation: We propose that the installer should be modularized where one can install individual microservices rather than a monolithic installation of the entire system. This also requires a change in the configuration management process.

Communication: We propose to keep the WCF TCP/IP based communication between the manager-worker, and client-manager. Eventually, the TCP/IP can be replaced by a *http* based communication. However, we suggest an asynchronous message-based communication with HealthMonitor to exchange data and to notify the health of each microservice. Therefore, the interaction between the health monitor and the rest of the system should be based on asynchronous messaging.

The deployment view in Figure 4 illustrates the scalability aspect of the TO-BE architecture. The noteworthy aspect here is that each microservice is also replicated across different nodes, and the HealthMonitor is deployed in a separate machine. With this distribution, we can achieve a better resource utilization of each node.

V. Results

Architecture evaluation is an important activity to reason about important architectural decisions. In our case, we have built a prototype of the "to-be" system where we have implemented the interaction of the manager and three types of workers that interact through WCF TCP/IP mechanism. The memory footprints of these components are kept identical to the actual system. In the prototype, the manager spawns a new worker container on-demand. We have subjected the prototype

[3]https://www.amqp.org/

with a customized load generator which creates a large number of random requests to the manager. We have measured the time to start of a containerized service (of the worker), and the overall response time for each request. We have used a Windows machine with a quad-core Intel® Core™ i7, 16GB of memory and 256GB of SSD storage, to deploy all the containers. The result is shown in Table I. The "workload-mix" column indicates how the load generator has randomly generated requests for each worker type. The TTS column shows the initial time to start and the median time to start a particular worker. For instance, in row 1 (15910,0) indicates the initial time to start a *container* corresponding to worker type 1 is 15910 ms, and then the median time to access this worker is negligible once the container is up. Likewise, (3302, 18.65) indicates the initial service response time for worker type 1 is 3302ms (when its initial TTS is 15910 ms), but the average response time after the first response time is 18.65ms.

TABLE I
PERFORMANCE ANALYSIS OF THE TO-BE PROTOTYPE

Workload mix	Time to start (TTS)	Response Time
32%-35%-33%	$(15910, 0)$, $(15109, 0)$, $(14927, 0)$	$(3302, 18.65)$, $(3341, 0.09)$, $(3312, 0.82)$
33%-34%-33%	$(21, 0)$, $(98, 0)$, $(23, 0)$	$(26, 10.63)$, $(24, 2.5)$, $(24, 2.8)$

Row 2 corresponds to workers running as a native OS process without containerization. The noteworthy point is that the containerized version provides a better response time for all the services, except worker type 1. The first worker contains a recursive version of the n^{th} fibonacci number calculation (the value of n generated randomly by the load generator) that consumes large computing and memory resource. Since a container reserves a fixed amount of resource, the resource intensive task may have a performance bottleneck if the container resource is exhausted. On the other hand, the container level resource isolation becomes beneficial for worker type 2 and 3 where the worker gets an exclusive resource inside the container, whereas in the native mode, the underlying OS schedules the resource based on its scheduling policy.

In order to reduce the initial TTS, researchers have attempted to checkpoint a container [10]. We have considered this approach for this prototype. Since the Windows version does not support checkpointing, we have created a replica of this prototype for the Ubuntu 18.04 platform using Docker version 17.02 by enabling its experimental checkpointing feature. Contrary to the claims made in [10], *we did not see any visible reduction of the initial startup time* if we use the checkpointed version of the container.

VI. CONCLUSION

Migration of a monolithic architecture to a microservice based distributed application is possible, provided the application is well modularized. In this paper we have considered a complex industrial application running in Windows that needs to be migrated to a microservice based containerized architecture. We have analyzed the architectural characteristics of the application and examined its readiness for migration. We have observed that the operating environment (Windows-based) plays a significant role in the migration effort. Windows MSI based user-friendly installation becomes a bottleneck for creating a container image of the system. Though the original system has been partitioned into a set of deployable components, they are still tightly coupled due to shared memory based data exchange, which makes it difficult to distribute the executables into a set of fine-grained containers. For architecture evaluation, we created a prototype of the TO-BE architecture and demonstrated that the "to-be" architecture can take significant workload with a reasonable response time. By deploying the "to-be" architecture in one machine, we also demonstrate that the "to-be" architecture will be able to utilize the resource effectively. An important future goal of this work will be to completely implement a partitioned, containerized environment of the current application and provide a quantitative analysis of the cost (migration effort) and benefit (performance, scalability, fault-tolerance) of the migration project.

REFERENCES

[1] D. Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, sep 2014.

[2] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, mar 2015.

[3] T. Goldschmidt and S. Hauck-Stattelmann. Software Containers for Industrial Control. In *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, aug 2016.

[4] T. Goldschmidt, S. Hauck-Stattelmann, S. Malakuti, and S. Grner. Container-based architecture for flexible industrial control applications. *Journal of Systems Architecture*, 84:28–36, mar 2018.

[5] S. He, L. Guo, Y. Guo, C. Wu, M. Ghanem, and R. Han. Elastic Application Container: A Lightweight Approach for Cloud Resource Provisioning. In *2012 IEEE 26th International Conference on Advanced Information Networking and Applications*. IEEE, mar 2012.

[6] H. Kang, M. Le, and S. Tao. Container and Microservice Driven Design for Cloud Infrastructure DevOps. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, apr 2016.

[7] J. Lewis and M. Fowler. Microservices: a definition of this new architectural term. techreport, ThoughtWorks Inc, 2014.

[8] S. Mazaheri, Y. Chen, E. Hojati, and A. Sill. Cloud benchmarking in bare-metal, virtualized, and containerized execution environments. In *2016 4th International Conference on Cloud Computing and Intelligence Systems (CCIS)*. IEEE, aug 2016.

[9] M. Mohamed, S. Yangui, S. Moalla, and S. Tata. Web Service Micro-Container for Service-based Applications in Cloud Environments. In *2011 IEEE 20th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*. IEEE, jun 2011.

[10] S. Nadgowda, S. Suneja, and A. Kanso. Comparing Scaling Methods for Linux Containers. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, apr 2017.

[11] C. Pahl. Containerization and the PaaS Cloud. *IEEE Cloud Computing*, 2(3):24–31, may 2015.

[12] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi. Cloud Container Technologies: a State-of-the-Art Review. *IEEE Transactions on Cloud Computing*, pages 1–14, 2017.