Software Architecture Evolution of a Physical Security Information Management System

Oğuzhan Özçelik
ASELSAN A.Ş.
Ankara, Turkey
e-mail: oozcelik@aselsan.com.tr

Halit Oğuztüzün
Department of Computer Engineering
Middle East Technical University
Ankara, Turkey
e-mail: oguztuzn@ceng.metu.edu.tr

***Abstract*—The planned reuse mentality of software product line engineering makes it possible to deliver similar products within a short amount of time. Physical Security Information Management (PSIM) system customizations tend to be similar to each other with fundamental requirements being more or less the same in different projects. One of the most common difference in these projects is the used sensors. Some sensors could be integrated into the PSIM system easily if they are compatible with a standard communication interface such as Open Network Video Interface Forum (ONVIF) protocols. But sensors that use a special communication interface need to be integrated one by one. A PSIM system is always expected to integrate additional sensors to its catalog. In order to do this easily, the parts that need to be developed to integrate a sensor must be segregated and developed individually for each sensor. In this work, we aim to segregate the sensor integration of a PSIM system and compare the old and new generations of the architecture qualitatively, based on architecture models.**

Keywords-Physical Security Information Management Systems; Physical Protection Systems; Software Product Line Engineering.

I. INTRODUCTION

A Physical Security Information Management (PSIM) system integrates diverse independent physical security applications and devices. Applications such as building management or network video recorder systems, and devices such as security cameras, access control systems, radars and plate recognition systems are used interconnectedly. It is designed to ensure the physical security of a facility, city or an open field, while providing a complete user interface to the security operators to monitor and control them.

The subject PSIM system of this work is called SecureX, which is not the name of the actual system but a placeholder used for confidentiality reasons. SecureX is a PSIM system that aims to satisfy the needs mentioned above and also to provide an easy integration environment for new sensors and applications. The ever-increasing number of such new systems and different security needs of different customers drove SecureX team to embrace a software product line engineering approach in order to reduce the response time to reply to the customers' demands. These demands vary from practical improvements to integrating a new sensor or security application as a feature to the system. SecureX is deployed with the full feature set and only at runtime these features are reduced to the ones required by a given customer, using different configuration files. Any new integration required by a customer needs to be developed as a feature in SecureX. Afterwards, a new SecureX build must be generated. Following every new integration, a new testing process takes place and because the previously integrated system might not always be available for testing, it must be guaranteed that the new integration will not affect the other integrations. In this work, a new method for integrating such new systems while reducing the number of required tests is proposed.

The rest of the paper is structured as follows. In Section II, several PSIM products and their specializations are mentioned. Also, we briefly explain how they approach the sensor integration problem and why that is not enough in the case of SecureX. In Section III, the general architecture of SecureX is described and the point where sensor integration takes place is shown. In Section IV, this sensor integration point is described in more detail. In Section V, the problems with the current architecture are explained and in Section VI, a new architecture that solves those problems is described. In Section VII, the benefits of the new architecture are shown by explaining how it solves each problem of the current design.

II. RELATED WORKS

There are several companies offering PSIM products. Although they provide every essential feature of a PSIM system, they may have different specializations. Genetec [1] provides a video analytics tool to detect intrusions. Milestone [2] uses its own Network Video Recorder (NVR) systems and provides an easy to use video management system. Nedap [3] is specialized in access control systems. However, not many details exist on how they work internally. These products integrate some general communication standards like ONVIF [4] protocols and also release Software Development Kits (SDK) and expect sensor manufacturers or customers to integrate their custom subsystems into the PSIM system as well. This way, they accelerate sensor integration by including numerous 3rd parties. While developing an SDK to use in integrations is a feasible solution, in the SecureX's case, the main objective is developing an architecture that can simplify not only the sensor integrations, but also the component selection to

 ISBN: 978-1-61208-946-1

deploy because different customers have different requirements. Another requirement is that the new architecture will be able to remove the update and test overhead. A software product line architecture would be suitable to accomplish this goal.

Recently, Tekinerdogan et al. [5] described how a PSIM system should be designed with software product line engineering methodologies to reduce the cost of development by improving reuse. The present work describes a step in architectural evolution toward a product line architecture.

III. ARCHITECTURE OF SECUREX

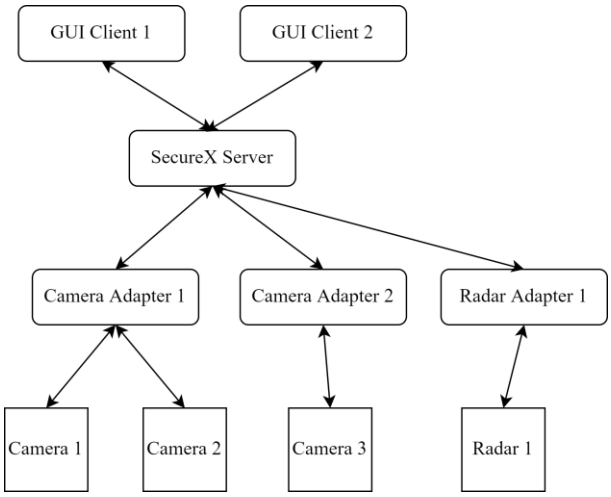

Figure 1. Deployment model of SecureX

SecureX has a distributed architecture which can be seen in Figure 1. Graphical User Interface (GUI) Clients of SecureX are installed on the computers of security officers, enabling them to monitor the entire security infrastructure of the area under surveillance. These clients are connected to the SecureX Server application which handles the communication between SecureX components. The server is also responsible for recording events, including detections and errors sent from adapter components to the central database. SecureX could also be installed in a hierarchical fashion in which higher servers could also control and monitor the security components that are connected to the servers under them. Under the SecureX Server, there are adapter applications for each sensor group such as camera, radar, plate recognition systems, access control systems, etc. These adapters are the points where the SecureX environment makes its connections to the outer world.

When a user wants to perform some action with a sensor, after pressing a button in the SecureX GUI Client, a message will be sent to the SecureX Server. Then, the server delegates this message to adapters and other servers that are hierarchically under that server. The message arrives at the sensor's adapter and, according to the Interface Control Document (ICD) used in its integration, a message would be sent to the sensor to perform the desired action. Events and detections caught by the sensors would follow the reverse route and find their way to the SecureX GUI Clients.

IV. EXISTING ADAPTER ARCHITECTURE

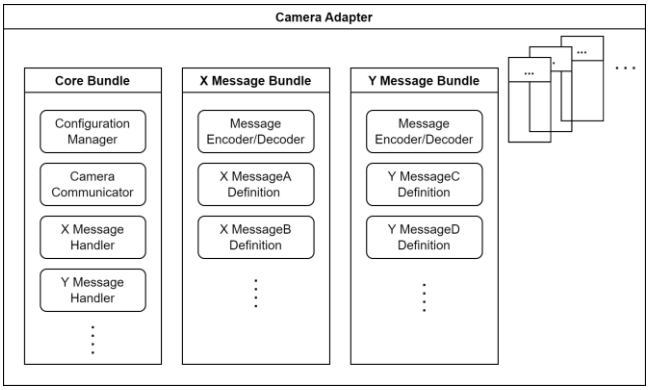

Figure 2. Simplified Camera Adapter model in the existing architecture

SecureX is developed using the Open Services Gateway Initiative (OSGi) framework, which is a Java [6] framework to develop modular software [7]. These modules are called "bundles" and the framework could install, uninstall and update them, even at runtime [8]. The bundles to be installed and their start levels are stated in bundle configuration files. A few of these bundles can be seen in Figure 2. SecureX uses this framework to take advantage of its service architecture. We use the Camera Adapter application to describe the adapter architecture, but all adapter applications of SecureX are quite similar.

The Camera Adapter application consists of many OSGi bundles whose purposes vary from providing network connection interfaces or utility tools, to message definition of sensors. These message definition bundles contain the methods for encoding and decoding messages to and from the sensor. Generally, the message formats for each sensor are different. They have different data types, header types, checksum calculation methods, big or little endian formats. Some sensors accept JSON formatted string messages and some require encoding messages in a certain length byte arrays and sending them. Information about how to communicate with a sensor is given in its ICD. A message bundle is basically an implementation of the related ICD.

The *Configuration Manager* class in the *Core* bundle is mainly responsible for opening a Transmission Control Protocol (TCP) port to accept incoming server connections and initializing the *Message Handlers*. Each sensor's type, model, unique identifier key and required information about establishing a connection to it is written in a configuration XML file. The *Configuration Manager* constantly iterates over these files, creating a *Camera Communicator* and a specific *Message Handler* for every new or updated file. Messages are received by the TCP server and forwarded from there to the *Camera Communicator* and lastly to the sensor's *Message Handler*.

 ISBN: 978-1-61208-946-1

A *Camera Communicator*, which extends from the *Sensor Communicator* class as in every other sensor family, is the class where the processing of messages that came from the server starts. It handles generic messages or preprocesses them before the messages arrive at the *Message Handler*. When a message is received from the server, it is added to the message buffer of every active *Camera Communicator* in that adapter. *Camera Communicators* take this message and decide if this message is meant for their sensor. To do this, they use the sensor identifiers in the messages. If the identifier is the same with the *Message Handler* they have, the message gets processed as will be explained in the subsequent paragraph, otherwise it is discarded.

The processing of the messages starts at the *Camera Communicator* level. Some messages are not specific to different sensor integrations and can be handled at the *Camera Communicator* level. Alternatively, some messages require a preprocessing step such as transforming some variables before they get forwarded to the *Message Handler*. After the initial processing is done, the *Camera Communicator* sends the message to the *Message Handler*.

The *Message Handler* is where the connection to the sensor is established using the protocol the sensor uses, which could be TCP, User Datagram Protocol (UDP), WebSocket, serial port, (Representational State Transfer) REST or any other network connection method that is stated in its ICD. The *Message Handler* knows how the connection should be established and how the incoming and outgoing messages should be processed. It receives the incoming message from the communicator and sends necessary commands to the sensor. The *Message Handler* needs a utility bundle to do the message conversions. When it needs to encode/decode messages to/from the sensor, it uses the Message bundle of that sensor that contains the message types, formats, checksum methods and the information of exactly how a message should be generated. After a message is generated, the *Message Handler* sends it to the sensor using the connection interface.

V. The Integration Problem

When the adapter starts, the *StartLevelEventDispatcher* thread in the OSGi framework initializes all bundles that are marked for auto-start in the bundle configuration file. In Figure 3, initialization of the *Core* bundle is shown. The *Core* bundle is the one that starts the main Camera Adapter process with its thread "*ConfigurationMonitor*". In the initialization of the *Core* bundle, a single *Configuration Manager* instance gets created. The *Configuration Manager* then opens a port to listen to incoming SecureX Server connections. After that, it starts a thread that periodically checks sensor configuration files to find new or updated configurations. If there is such a file, then the *Configuration Manager* creates a *Camera Communicator* and the *Message Handler* for that sensor. In the existing architecture, in order to create a *Message Handler* instance, the *Configuration Manager* has to know which *Message Handler* needs to be used for which sensor configuration. In the configuration file, the identifier of the correct *Message Handler* is given and the *Configuration Manager* uses that identifier to construct the *Message Handler*. But these *Message Handler* classes are inside the *Core* bundle and the *Configuration Manager* has a class dependency for them. This is the root problem in the current architecture.

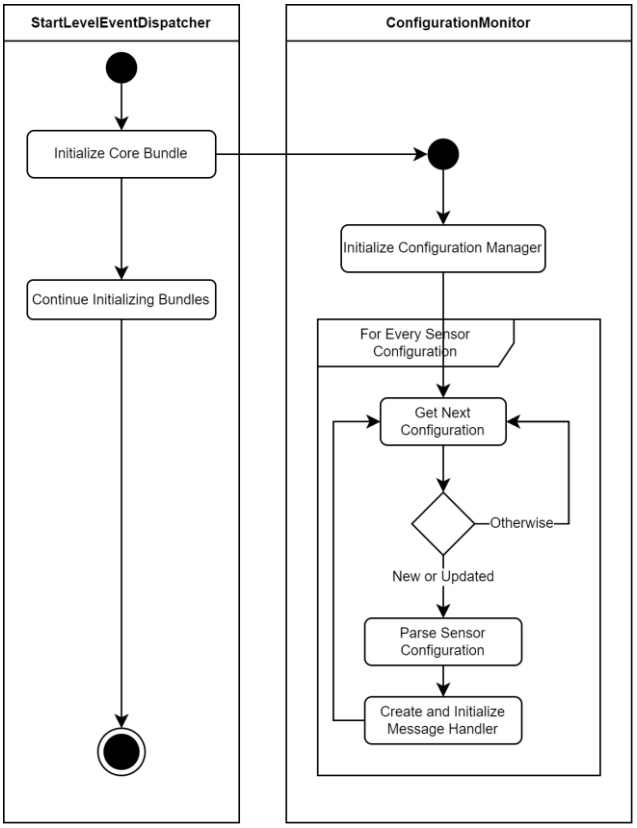

Figure 3. Message Handler initialization in the existing architecture

A. Difficulties with the Existing Architecture

In order to carry out a new sensor integration, the message definition bundle has to be added in the Camera Adapter product file and its *Message Handler* has to be included in the *Core* bundle. The *Configuration Manager* class needs to know with which configuration identifier the new *Message Handler* should be constructed beforehand, hence the dependency. Because of this design, integrating or updating the integration of a sensor requires updating the *Core* bundle in the adapter. The components in the *Core* bundle, such as *Configuration Manager* and *Camera Communicator*, are used in every *Message Handler* and need to be compatible with all of them too. Therefore, any change in those components in the integration of a sensor could affect the already integrated sensors and cause them not to function as intended. Alarms detected by the sensor might start not to be forwarded to the server or changing the orientation of the sensor becomes difficult because of a change in some movement speed calculations.

In the current design, to update an already deployed system, a complete new build needs to be generated and tested. But testing of the previous sensor integrations are not always easy or even possible. These sensors could be

 ISBN: 978-1-61208-946-1

produced in very limited numbers and they can only be found in the customer's facilities, working with the previous SecureX version. The location of these facilities might be difficult to access too and trips to these locations are not only costly, but sometimes, also dangerous. Because these sensors are almost always used in closed networks, the only way to test them is by going to these facilities, increasing the test cost. Also, customers would not want testers to separate these sensors from the PSIM system to test with the new version, creating a window of vulnerability.

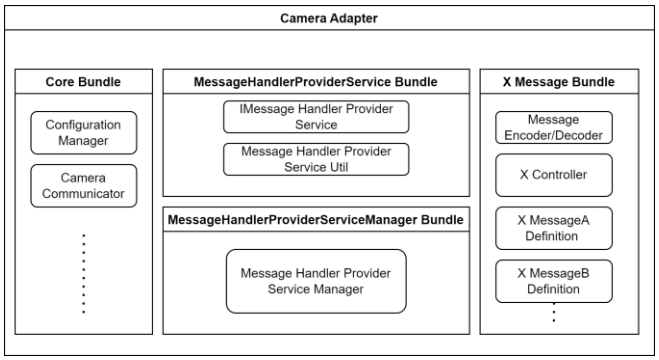

Figure 4. Simplified Camera Adapter model in the new architecture

Even if the tests are somehow completed, the update procedure has its own problems. To quickly update systems used in remote locations with little to no network access, or used in thousands of mobile locations without stable internet access, the update size must be minimal. But, with the current architecture, the whole adapter build needs to be updated, rather than just a couple of bundles.

Also, to catch up with new and updated sensors or security systems, 3rd party companies are employed for integrations. But this process is done through signing a Non-Disclosure Agreement (NDA) and sharing huge parts of the adapter code with them to be used to integrate the sensors. Any one of them could expose the code at any point and this indeed is a security vulnerability.

Because of these reasons, there is a need for an architecture that ensures that the new integrations will not affect the existing ones. The main problem with the current design is, for every new integration, it has a need to update the *Core* bundle. The reason for that is the *Configuration Manager* class needs to know all available *Message Handlers* and for what kind of sensor they need to be used beforehand via class dependencies. In the new architecture, this problem is targeted with the aim to reduce testing overhead, reducing the amount of code that is shared with 3rd parties and also enables updating the deployed systems with very low data.

VI. New Adapter Architecture

To solve the problems with the existing architecture, a new adapter architecture shown in Figure 4 is developed. With this new architecture, all *Message Handler* classes moved to their message definition bundles and an OSGi service called *IMessage Handler Provider Service* that provides a *Message Handler* constructor for a given configuration identifier is developed. With that change, now the *Core* bundle does not depend on the *Message Handlers* or message bundles, but it depends on the *Message Handler Provider Service* bundle. Message bundles also depend on this service bundle too. This fixes the problem of the *Core* bundle depending on *Message Handlers* and its need to be updated to include a dependency with every new sensor integration. These message bundles, similar with every other OSGi bundle, can be extracted as a compiler .jar file and be

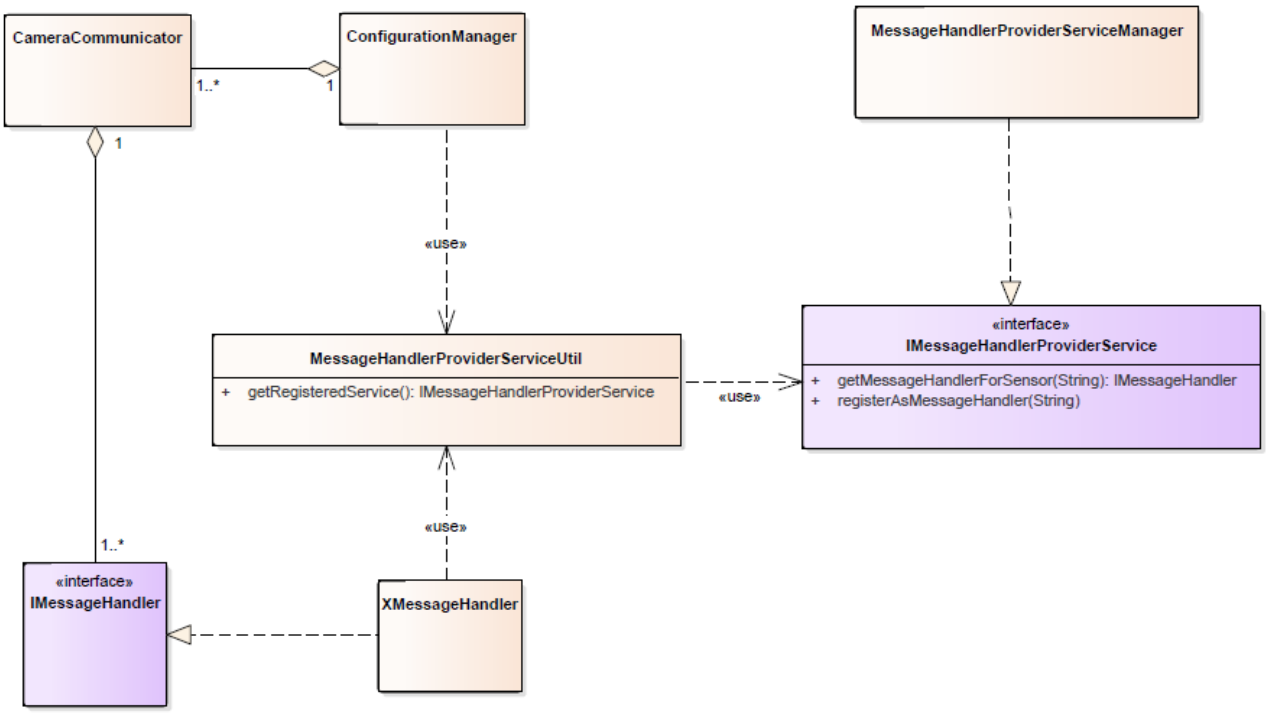

Figure 5. Camera Adapter Class Diagram (Simplified)

 ISBN: 978-1-61208-946-1

installed externally.

Figure 5 shows the new classes and their hierarchies while Figure 6 shows the new message handler initialization procedure. The *Message Handler Provider Service Manager* implements the *IMessage Handler Provider Service* interface and when it is initialized by the *StartLevelEventDispatcher*, it reads a directory in which the new sensor integration bundles are placed as .jar files. The manager installs those new integrations and, after the initialization of every new bundle, it registers itself as an instance that implements the *IMessage Handler Provider Service* interface to the OSGi context.

While those bundles are initialized, they register themselves with the *IMessage Handler Provider Service* in the OSGi context using the configuration identifier to indicate the sensor they should be used for. Accessing the registered *IMessage Handler Provider Service* is made possible through the *Message Handler Provider Service Util* class. This access technique blocks the requester thread until a service instance registers. The *Message Handler Provider Service Manager* registers itself after it initializes every integration file. Because *Message Handlers* access this manager using the same blocking technique, they can only register themselves after the service manager finishes its job. This causes all *Message Handlers* to register almost simultaneously.

While this process continues, the *Core* bundle also starts by the *StartLevelEventDispatcher* thread and continues its regular processes. But this time, the *Configuration Manager* class does not know any *Message Handler* itself. The dependencies for *Message Handler* classes are removed. When the *Configuration Manager* reads a sensor configuration, it uses its configuration identifier and asks a *Message Handler* constructor from the registered *IMessage Handler Provider Service*. It uses the *Message Handler Provider Service Util* class to access the service, so it also waits until an *IMessage Handler Provider Service* finishes its initializations and registers itself. After that, if a *Message Handler* for a given configuration identifier exists in the application, the *Configuration Manager* uses its constructor to create an instance and initialize it. The initialized *Message Handler* connects to the sensor and starts its regular processes. If a *Message Handler* does not exist for that identifier, the *Configuration Manager* skips that configuration for this iteration.

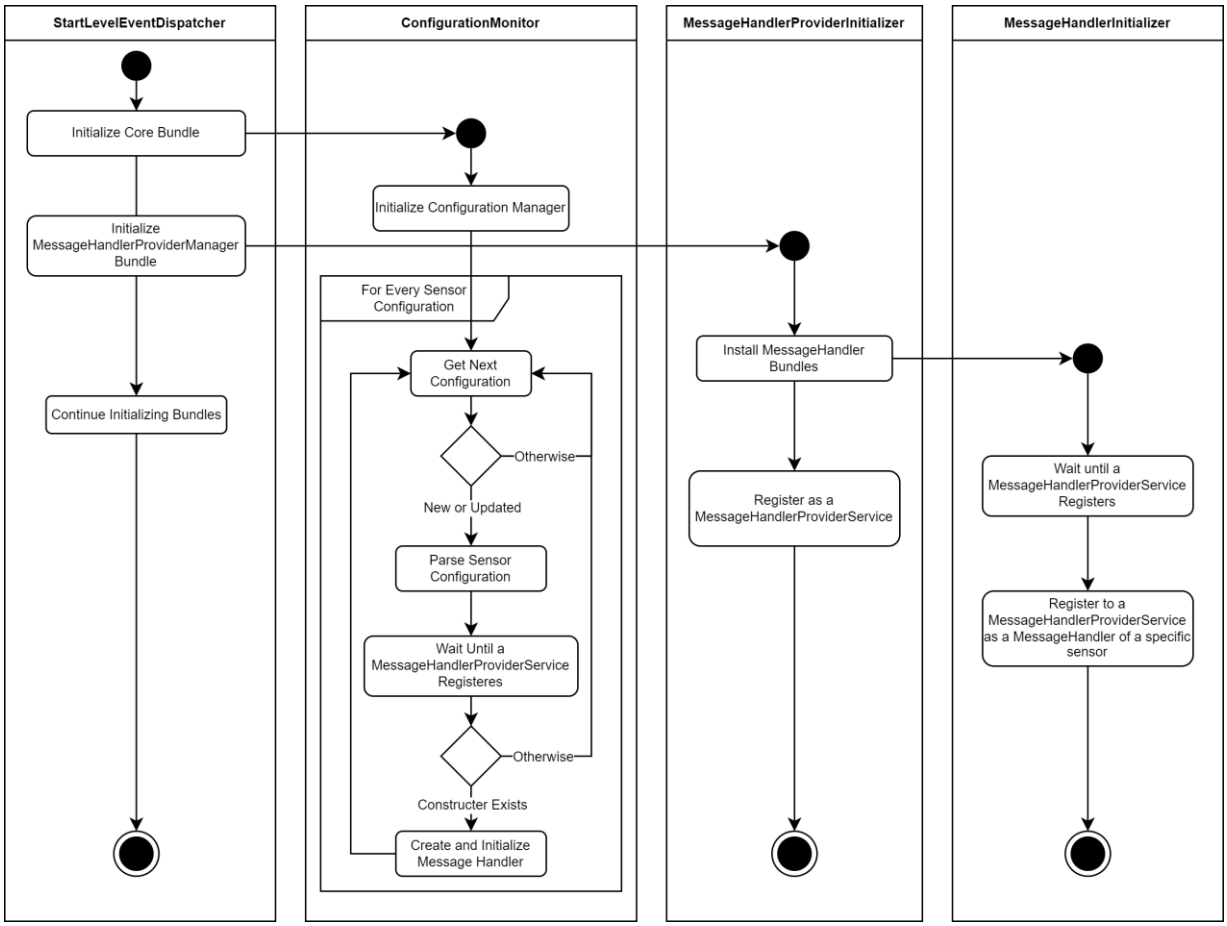

Figure 6. Message Handler initialization in the new architecture

VII. CONCLUSION

The proposed adapter architecture allows us to integrate additional sensors into the already deployed PSIM systems, without requiring to generate another complete build of an adapter. Because previous integrations are not touched, integration tests of only the newly integrated sensors would be sufficient. When the sensor is integrated, it will most probably be available and going to the field and using the sensor of a customer will no longer be needed.

The .jar files of the integration bundles are smaller than one MB so system updates can be completed even with unstable or slow networks. Even if new sensor integrations have a problem working with previously integrated sensors, simply removing the .jar file would be enough to revert back to the previous deployment.

Segregating sensor integration also enables easily selecting and combining different integration bundles according to the project's requirement, as one could expect from a system developed with software product line principles. The new design also enables employing 3rd party companies for integrations without sharing the bulk of the adapter code. Now, any integrator could develop an integration bundle only with the *Message Handler*, *IMessage Handler Provider Service* and the *Message Handler Provider Service Util* classes.

The new architecture provides a helpful pattern towards transforming SecureX into a Software Product Line (SPL). An external .jar installer service could be used not only for sensor integrations, but also for features such as additional GUI views or in the server, new alarm evaluation algorithms. Because every feature is developed as an OSGi bundle, they all could be externalized.

The sensor integration problem could be solved by developing an SDK, similar to the products given in the Section II, but this design also eliminates the need of deploying the SecureX with a full feature set and stripping it off with configuration files at runtime. As this design gets implemented in other parts of the SecureX, they could all be removed from the base build and can be added per customer demand. The new design opens a path for segregating such different aspects in the SecureX and is expected to be even more beneficial in the future.

REFERENCES

[1] Genetec KiwiVision. [Online], retrieved March 2022 Available: https://www.genetec.com/products/

[2] Milestone XProtect. [Online], retrieved March 2022 Available: https://www.milestonesys.com/solutions/

[3] Nedap Aeos Access Control. [Online], retrieved March 2022 Available: https://www.nedapsecurity.com/solutions/

[4] Open Network Video Interface Forum (ONVIF). [Online], retrieved March 2022 Available: https://www.onvif.org/

[5] B. Tekinerdoğan, İ. Yakın, S. Yağız, and K. Özcan, "Product Line Architecture Design of Software-Intensive Physical Protection Systems". IEEE International Symposium on Systems Engineering (ISSE), 2020, pp. 1-8, doi: 10.1109/ISSE49799.2020.9272239.

[6] "The Java Language Specification, Java SE 8 Edition" J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. Apr. 2015. [Online]. retrieved March 2022 Available: https://docs.oracle.com

[7] R. S. Hall, K. Pauls, S. McCulloch, and D. Savage. "OSGi in Action - Creating Modular Applications in Java". Manning Publications, 2011

[8] "OSGi Service Platform, Core Specification, Release 8," The OSGi Alliance, April. 2018. [Online]. retrieved March 2022 Available: http://docs.osgi.org/specification/

 ISBN: 978-1-61208-946-1