A Model-Based Approach Toward Executable Middleware Architecture for Tactical C2IS

Norman Jansen, Daniel Krämer, Marc Spielmann
Fraunhofer FKIE
Fraunhoferstr. 20, 53343 Wachtberg
GERMANY

{norman.jansen, daniel.kraemer, marc.spielmann}@fkie.fraunhofer.de

ABSTRACT

The paper illustrates the elementary steps underlying model/architecture transformations, such as matching, refinement, extension, and reduction. To this end, a high-level model of a tactical middleware is embedded into the system architecture of an existing land C2IS solution: the middleware model is used as guidance (or reference) for transforming the system architecture of the target system into the system architecture of an enhanced system (i.e., the system enriched with middleware functionality). To demonstrate feasibility of the transformation, a prototype of the enriched system has been implemented. The implementation closely relates to both our middleware model and the system architecture of the target system, and is in this sense model-driven.

1 INTRODUCTION

To enable the use of Command and Control Information Systems (C2IS) from higher command echelons (usually equipped with Ethernet-like high-speed communication networks) down to tactical echelons (where narrow-band radio links are still the main means of communication) a close coordination between C2IS and the underlying communication infrastructure is of critical importance. This is particularly true for the tactical level, where an efficient utilization of the scarce network resources is essential. In order to optimize the utilization of network resources, we have developed a middleware model for coordinating C2IS applications and (tactical) communication services. Purpose of the envisioned middleware is to achieve a better matching of application requirements with the available network resources. The model itself is formulated on a relatively high abstraction level in order to provide a succinct overview of the main concepts (functionalities, components, interfaces, dependencies), while leaving implementation details and target system artefacts to subsequent transformation steps.

In this paper, we demonstrate how to embed our middleware model into the system architecture of a land C2IS solution currently in use in the German army. More precisely, we show how the system architecture of the target system is enriched with new components and interfaces deduced from our middleware model. When implemented and integrated into the target system, the new components and interfaces provide (part of) the envisioned middleware functionalities. The process of embedding our middleware model into the system architecture illustrates the basic steps underlying model/architecture transformations, such as matching elements (components, interfaces), refinement and reduction of elements, and introduction of new elements.

2 MIDDLEWARE MODEL

The main concepts of our middleware design [1, 2] are depicted in Figure 1. Essentially, we follow a cross-layer design, where two new convergence layers (see »*Transport Convergence*« and »*Technology Convergence*« in Figure 1) complement the existing network layers of the classic ISO/OSI protocol stack and where two new cross-layer components (see »*Generic Network Access*« and »*Cross Layer Coordination*« in Figure 1) handle information exchange between the various layers of the enriched protocol stack. The new layer »Transport Convergence« unifies the access to different transport protocols: depending

on the communication links currently available it chooses transport protocols appropriate for the specific link (channel) properties. The main role of the new layer »Technology Convergence« is to unify the access to different communication technologies (e.g., VHF, HF, SATCOM, etc.). The new cross-layer component »Generic Network Access« provides technology-independent interfaces (see »*Network Information*« and *Network Configuration*« in Figure 1) for extracting status information from and inducing configuration data into network protocol layers. The main role of »Cross Layer Coordination« is to control the exchange of cross-layer information and to provide mechanisms enabling C2IS apps to adapt to the currently available communication services (more precisely, to the currently available quality of these services). To this end, »Cross Layer Coordination« provides a »*Control API*«, featuring methods for exchanging control information between C2IS apps and »Cross Layer Coordination«. The actual message transfer between C2IS apps and the enriched protocol stack is handled via a »*Transport API*«.

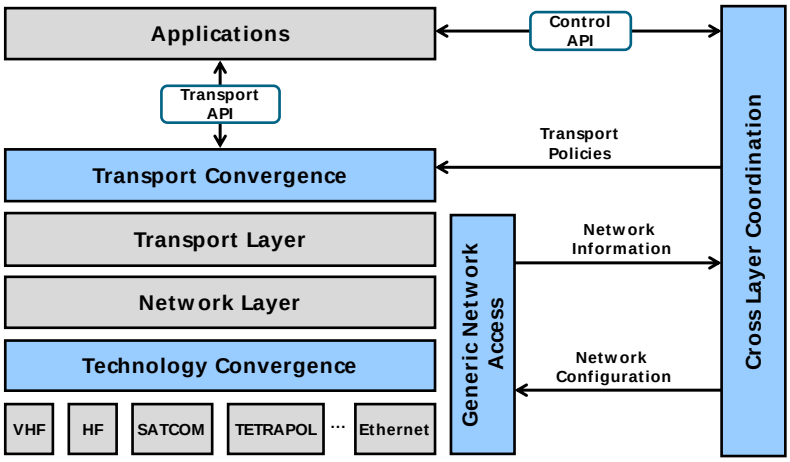

Figure 1: High-level model of a middleware for tactical C2IS and communication systems.

Notice that our choice of depicting the middleware design in the form of an (enriched) ISO/OSI protocol stack is not by coincidence. In fact, our middleware is a distributed system similar to a communication service deployed in a communication network, where system/service components are distributed over various network nodes and where peer components (i.e., components of the same "type" but at different network nodes) may exchange information via some internal protocol. For instance, our cross-layer component »Cross Layer Coordination« is instantiated at various network nodes, and the different instances (or "copies") of the component may exchange information by means of a middleware internal protocol. The information exchange may concern, e.g., the general network topology or the current network status.

The main purpose of the middleware model is to serve as (partial) reference model for the development of future C2IS and communication system solutions. However, in order to demonstrate that middleware functionality can even add value to operational equipment, we have applied the middleware model to the system architecture of a land C2IS solution currently in use in the German army (see also [3]).

3 INITIAL SITUATION

The target system considered consists of a C2IS (basically featuring apps for force tracking, situational picture and messaging), a communication server (Com server for brevity) and various communication technologies like VHF based radios, SATCOM, and trunked radio systems (see Figure 2, left hand side). For simplicity, we here focused on the situation where the Com server is connected only to one or more VHF radios. To the application layer (i.e., to the C2IS apps) the Com server offers a message transfer service by means of the Extended Simple Mail Transfer Protocol (ESMTP). The server also features a configuration service called »*Airport*« for querying the server's state and manipulating the server's functionality (both during runtime). The »Airport« service already implements certain aspects of the functionality of our cross-

layer component »Generic Network Access«. In order to illustrate this observation, we have replaced in Figure 2 »Airport« with »Generic Network Access« in dashed lines. A similar observation can be made for an abstraction layer already implemented in the Com server: the layer can be viewed as a rudimentary version of our convergence layer »Technology Convergence«. This fact is illustrated in Figure 2 by showing »Technology Convergence« in dashed lines.

Both the C2IS and the Com server run on separate hardware platforms (hosts; see Figure 2, right hand side) which are encapsulated in ruggedized boxes and which are connected via a local area network (Ethernet).

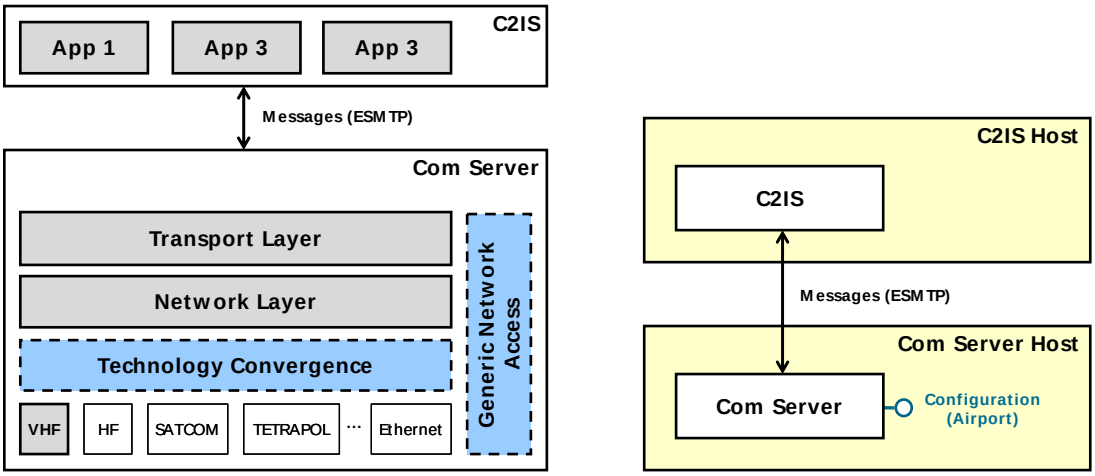

Figure 2: Conceptual views of the software architecture (left) and of the system architecture (right) of the target system.

In order to provide a short-term solution (max. 1 year) we had to observe the following constraints:

1. Neither the introduction of new hardware components nor the modification of existing hard- or software components is allowed.
2. New software components must be deployed on the existing hardware platforms and must use existing software infrastructure (runtime environments, interfaces).

As for constraint (2) we found Python interpreters on both the C2IS host und Com server host. These interpreters can provide sufficient runtime environments for our middleware functionality.

4 MODEL TRANSFORMATION

In this section, we show how to embed our middleware model (Section 2) into the system architecture of the target system (Section 3). Essentially, we stepwise enrich the system architecture with new elements (components, interfaces) deduced from the middleware model. When implemented and integrated into the target system, the new middleware elements provide (part of) the envisioned middleware functionalities inside the enhanced system.

As a first step of the transformation process, we have matched elements of our middleware model (Figure 1) with already existing elements in the software architecture of the target system (Figure 2, left hand side). For instance, the Com server already implements rudimentary versions of the components »Technology Convergence« and »Generic Network Access« and of the interfaces »Network Information« and »Network Configuration«. Since the Com server does not provide the full functionality of »Generic Network Access«, only reduced versions of the interfaces »Network Information« and »Network Configuration« are available. To indicate this reduction in functionality we have renamed the interfaces »Network Information« and

»Network Configuration« to »*Radio Link Info*« and »*Mesg Delete Request*«, respectively (see Figure 3, left hand side). The new names are intended to provide intuitive descriptions of the reduced interface functionality.

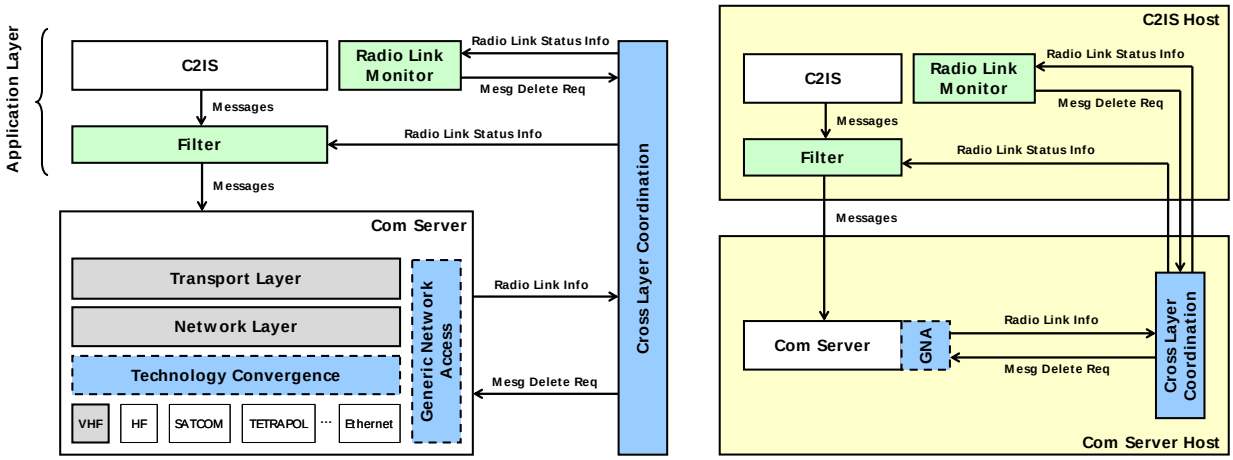

Figure 3: Conceptual views of the software architecture (left) and of the system architecture (right) of the enhanced system.

We have then identified components of our middleware model whose functionality is not present in the target system and introduced these components into the target system's software architecture. This immediately applies to our »Cross Layer Coordination« component. But it also – indirectly – applies to our »Control API« (Figure 1) which is closely tied to the functionality of »Cross Layer Coordination«: together »Cross Layer Coordination« and »Control API« provide mechanisms enabling C2IS apps to adapt to the currently available quality of communication services. Because of constraint (1) in Section 3 (see at the end of the section) we are not allowed to modify C2IS apps. Thus, in order to be able to implement at least a rudimentary version of the combined functionality of »Cross Layer Coordination« and »Control API« we have introduced two new components into the target system's software architecture: a component »*Filter*« whose role is to delay and – under certain circumstances – delete messages of the force tracking app, and a component »*Radio Link Monitor*« whose role is to provide C2IS users with firstly a visualization of certain network information (more precisely, the current radio link status and the current message queue status) and secondly, some means to eliminate messages from the message queue. Notice that the introduction of »Filter« and »Radio Link Monitor« can be seen as an extension of the functionality of the C2IS. From the perspective of our middleware model it rather is a reduction of the "network status adaption" functionality of future applications which make use of the combined functionality of »Control API« and »Cross Layer Coordination«.

Further transformation steps concerned the refinement of elements. For instance, since »Filter« and »Radio Link Monitor« do not make use of the combined functionality of »Control API« and »Cross Layer Coordination«, it is not necessary to provide this (combined) functionality on the target system. For that reason, we have replaced in the target system's software architecture »Control API« with two interfaces »*Radio Link Status Info*« and »*Mesg Delete Req*« (with the obvious functionality).

In the final phase of the transformation process, we have considered possible mappings of the resulting software architecture (Figure 3, left hand side) into the system architecture of the target system (Figure 2, right hand side), and assessed the feasibility and practical implications of these mappings. Notice that the decision of where to deploy new software components on the target system crucially depends on technical, financial, and legal restrictions imposed by the target system and its supporting projects (development, procurement, operations, maintenance). Figure 3, right hand side, shows the mapping which we have chosen.

Our choice was guided by the following aspects:

- In the nearby future there will be a regeneration of the Com server. Extensions of its functionality are already planned. An introduction of »Generic Network Access« functionality and/or »Cross Layer Coordination« functionality is a realistic option for the regeneration process.
- A regeneration of the C2IS is currently under consideration, but not yet as "mature" as in the case of the Com server. The current version of the C2IS is a closed system: it has been proven almost impossible to extent its functionality, for various reasons. However, the Python interpreter present on the C2IS host provides a runtime environment sufficient for implementing the functionality of both »Filter« and »Radio Link Monitor«.
- From a conceptual point of view, the functionality of both »Filter« and »Radio Link Monitor« should be part of a future C2IS.
- From a conceptual point of view, future implementations (including extensions) of the functionality of »Control API« should be backward compatible. The same holds true for the two interfaces »Radio Link Status Info« and »Mesg Delete Req« (as possible parts of a future implementation of »Control API«).
- Because of the last postulation (backward compatibility of »Control API«) plus the fact that the functionality of both »Filter« and »Radio Link Monitor« are not very complex, do not consume many resources, and thus do not crucially interfere with the performance of their host, mapping »Filter« and »Radio Link Monitor« onto the C2IS host is a feasible solution.

The chosen mapping implies a distribution of our middleware functionality among the two hosts of the target system (as depicted in Figure 3, right hand side). Possible implications for the development and maintenance of software components implementing our middleware functionality are as follows:

- Releasing of software components implementing the functionality of »Generic Network Access« or »Cross Layer Coordination« should be synchronized with the releasing of Com server updates.
- Since »Filter«, »Radio Link Monitor«, and the C2IS are only loosely coupled (in fact, there is only the ESMTP interface between »Filter« and the C2IS, which has been stable for years for backward compatibility reasons), there is no strong argument for synchronizing the releasing of C2IS updates with the releasing of software components implementing the functionality of »Filter« or »Radio Link Monitor«.
- If changes of the Python runtime environment of »Filter« and »Radio Link Monitor« do not occur "frequently", one may consider including »Filter« and »Radio Link Monitor« into the releasing of Com server updates.

Notice that introducing new software components sharing interfaces with different hard- and software components of the target system (in our case the C2IS, the Com server, and their corresponding hosts) and the fact that those components are sponsored and supported by different procurement projects inevitable increases the complexity of the overall system and entails additional coordination measures between the involved projects and project teams. However, we are optimistic that the system architecture for the enhanced system outlined in this section keeps the additional coordination effort necessary for introducing middleware functionality into the target system to a minimum.

5 MODEL-DRIVEN DEVELOPMENT

Using the developed system architecture for the enhanced system as starting point, we describe in this section how a low-level implementation model can be obtained by refining the new middleware elements of the architecture. We also provide a glimpse on the actual implementation based on the implementation model. Since the entire development process – from high-level middleware model to low-level

implementation model – is guided by model transformations, the development of our supplementary software components can be viewed as a model-driven development process.

Derivation of implementation model. We started by defining software packages corresponding to the new middleware elements of the enriched system architecture. Figure 4 provides an overview of the software packages of our implementation model. We also introduced two subsidiary software packages (see »*Interface_Stubs*« in Figure 4) whose role is to encapsulate implementations of new interfaces (stubs) on the C2IS host and on the Com server host, respectively. As a first step toward refining the defined software packages we investigated existing technologies/solutions which can be used for implementing some part of the functionality of our new middleware elements. For instance, there exists an Application Layer Traffic Optimization (ALTO) server [4, 5] providing a certain portion of the functionality of »Cross Layer Coordination«. We decided to employ this ALTO server in our middleware implementation and introduced a component »*ALTO_Server*« (as part of the software package »*Cross_Layer_Coordination*«, see Figure 4) representing the existing implementation of the server.

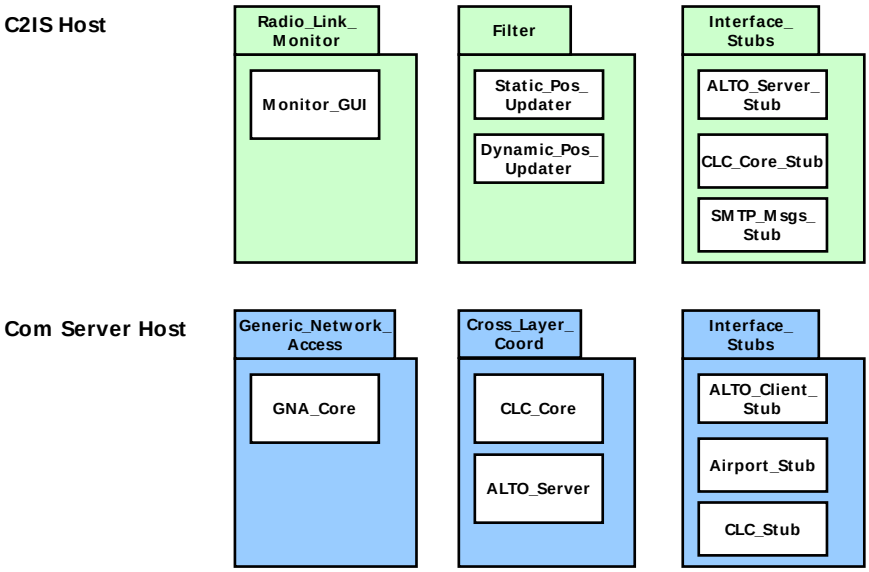

Figure 4: Implementation model derived from the enriched system architecture (see Figure 3 right hand side) by refining the new middleware elements.

Another example for (re)using existing software infrastructure concerns the Com server itself: as already mentioned in Section 5, the server provides a configuration service »Airport« which, when adequately masked by additional functionality (e.g., by means of a software wrapper), can be viewed as a rudimentary implementation of our cross-layer component »Generic Network Access«. Hence, we introduced a component »*GNA_Core*« (as part of the software package »*Generic_Network_Access*«) for providing the additional functionality. Notice that, since the »Airport« functionality and interface is already available on the target system (as part of the Com server), there is no need to introduce an explicit representation (component) for »Airport« into our implementation model. Instead, we introduced a component »Airport_Stub« (as part of the Com server host's »Interface_Stubs«) providing functionality for accessing »Airport«.

Further refinement steps concerned the packages »Radio_Link_Monitor« and »Filter« as well as the two interface packages »Interface_Stubs«. In a first refinement step of »Radio_Link_Monitor« we introduced a component »*Monitor_GUI*« whose role is to implement a graphical user interface (GUI) suitable for handling user interactions with our »Radio Link Monitor« (see Section 4). »Monitor_GUI« receives radio link status information from »Cross_Layer_Coordination«, which in turn obtains the information from the ALTO server (wrapped by »Cross_Layer_Coordination«). In order to represent this information flow in our

implementation model, we introduced two interface stubs »ALTO_Server_Stub« and »ALTO_Client_Stub« into the two interface packages »Interface_Stubs«. The »ALTO_Server_Stub« acts as a proxy for the ALTO server running on the C2IS host and enables »Monitor_GUI« to send requests to the ALTO server. In the other direction, the ALTO server has to answer to those requests. Therefore, a proxy »ALTO_Client_Stub« representing the server's client (in this case »Monitor_GUI«) is needed on the Com server host.

Another refinement step of »Radio_Link_Monitor« concerned the functionality of »Radio Link Monitor« which enables C2IS users to delete messages from the Com server's message queue. This functionality can not be provided by the ALTO server. Therefore, we introduced a component »*CLC_Core*« and an interface stub »*CLC_Core_Stub*«. »Radio Link Monitor« can send »Mesg_Delete_Request« messages via »*CLC_Core_Stub*« to »CLC_Core« which handles those messages (by transferring them to the Com server via »Generic Network Access«).

Refinement of the »Filter« package was guided by two different versions of filter functionality we wanted to implement: a static version and a dynamic version. The static version »*Static_Pos_Updater*« implements a prioritization strategy for position updates of the force tracking app according to a fixed data rate. In contrast, the dynamic version »*Dynamic_Pos_Updater*« calculates a dynamic update rate based on the current radio circuit utilization. In order to forward position updates (received from the force tracking app) we introduced an interface stub » *SMTP_Msgs_Stub*«. This stub establishes a connection to the Com server and provides »Filter« with an appropriate forwarding method.

Implementation. The implementation model described above served as framework for the development of our supplementary software components. For the actual implementation, we chose the programming language Python since Python interpreters are available on both the C2IS host und Com server host. Python provides a rich collection of libraries implementing many low-level functions. For instance, there exists a Python implementation of SMTP-Proxies which we used for our implementation of »Static_Pos_Updater« and »Dynamic_Pos_Updater« (which both need to forward position updates of the force tracking app in the form of SMTP messages). Also, the existing ALTO server was implemented in Python, so that the integration in our implementation was quite easy. We simply had to modify certain portions of the original implementation. In order to provide the reader with an impression of the outcome of the last refinement step, i.e., the actual code derived from the implementation model, we show in Figure 5 an extract from the implementation of our software package »ALTO Server«. The lower box in the figure contains code for creating and starting an ALTO server instance. The middle box contains code for modifying the implementation of a request handler (in the original ALTO server implementation).

It should be noted that the derivation of the implementation model and the refinement to an actual implementation usually is not a linear process. During the implementation one often encounters technical difficulties not anticipated beforehand, so that a modification of the implementation model or even of the original architecture is advisable or necessary. Hence, in practice the overall process most likely resembles an iterative process, where certain steps (e.g., revision of a model on some level of abstraction) are repeated several times.

```
from NetworkMapService import NetworkMapData                    import related modules
from ALTO_Messages.Messages import ServerResponseMessage
from ALTO_Server.CostMapService import RspCostMap, DstCost, CostMapData
from GNA_Core import KommServerVhfStatus
…
class ALTORequestHandler(BaseHTTPServer.BaseHTTPRequestHandler):
    """Functions to handle the different commands."""
    def do_HEAD(self):
        …
    def alot_get_map_core_pid_dynamic_cost(self):
        …                                                       ALTO request handler

if __name__ == '__main__':
    ksstat = KommServerVhfStatus.VhfStatus(config_params.get("source_address"),
    config_params.get("dest_address"),ks_vhf_if)
        server_class = BaseHTTPServer.HTTPServer
        httpd = server_class((HOST_NAME, PORT_NUMBER), ALTORequestHandler)
    httpd.serve_forever()                                       server instantiation and start
```

Figure 5: Extract from Python module „ALTO_Server.py“

6 CONCLUSIONS

In order to illustrate the elementary steps underlying model/architecture transformations, we have described how the high-level model of a tactical middleware can be embedded into the system architecture of an existing land C2IS solution. The middleware model served as guidance during the transformation process, which converted the current system architecture of the target system into the system architecture of an enhanced system. From the later architecture we derived an implementation model for supplementary software components implementing middleware functionality for the target system. The implementation model was further refined to an actual implementation. The development of our supplementary software components can be viewed as a model-driven development process, for each step of the process was guided by model transformations.

7 REFERENCES

[1] Barz, C.; Jansen, N.; Thomas, D.: Middleware for Tactical Military Networks. Military Communications and Information Systems Conference (MCC), 2010, Wroclaw, Poland.

[2] Barz, C.; Jansen, N.; Spielmann, M.; Thomas, D.: Nachrichtenorientierte Middleware für militärische Kommunikationsnetze – Basisarchitektur, Final Report, 2011, FKIE.

[3] Barz, C.; Jansen, N.: Towards a Middleware for Tactical Military Networks – Interim Solutions for Improving Communication for Legacy Systems. Military Communications and Information Systems Conference (MCC), 2011, Amsterdam, The Netherlands.

[4] IETF: ALTO Protocol. 2011. https://datatracker.ietf.org/doc/draft-ietf-alto-protocol, April 2013.

[5] IETF: Application-Layer Traffic Optimization (alto) WG. 2012. https://datatracker.ietf.org/wg/alto/charter, April 2013.