Evaluation of an OPC UA-based Access Control Enforcement Architecture

Björn Leander[1,2] [0000−0003−2488−5774] ✉, Aida Čaušević[1,3] [0000−0001−5293−3804], Hans Hansson[1] [0000−0002−7235−6888], and Tomas Lindström[2]

[1] Mälardalen University, Västerås, Sweden {bjorn.leander, aida.causevic, hans.hansson}@mdu.se
[2] ABB Industrial Automation, Process Control Platform, Västerås, Sweden tomas.lindstrom@se.abb.com
[3] Alstom Rail AB, Västerås, Sweden aida.causevic@alstomgroup.se

Abstract. Dynamic access control in industrial systems is becoming a concern of greater importance as a consequence of the increasingly flexible manufacturing systems developed within the Industry 4.0 paradigm. With the shift from control system security design based on implicit trust toward a zero-trust approach, fine grained access control is a fundamental requirement.
In this article, we look at an access control enforcement architecture and authorization protocol outlined as part of the Open Process Communication Unified Automation (OPC UA) protocol that can allow sufficiently dynamic and fine-grained access control. We present an implementation, and evaluates a set of important quality metrics related to this implementation, as guidelines and considerations for introduction of this protocol in industrial settings. Two approaches for optimization of the authorization protocol are presented and evaluated, which more than halves the average connection establishment time compared to the initial approach.

1 Introduction

Within industrial systems, such as industrial control systems, logistics, manufacturing, etc., cybersecurity is a factor of growing concern. The industrial automation systems of today are growing increasingly complex, heterogeneous, dynamic and interconnected [1–3], which implies that the currently used cybersecurity models based on implicit trust are no longer tenable. Instead using a zero-trust approach to cybersecurity is gaining ground [4, 5]. Access Control [6] is one of the major cybersecurity mechanisms in any information system and fine-grained access control is a basic requirement for a zero-trust architecture [7].

When conducting research on access control, it can be useful to structure the research according to Policy-, Enforcement- and Implementation-models (PEI), as suggested by Sandhu *et al.* [8], where the P-models describe how to the rules are formulated, E-models describe the enforcement architecture, and I-models describe the implementation of the components of the enforcement architecture.

Expressing sufficiently fine-grained access policies for dynamic industrial systems is a challenging task, but in this work we assume that such rules can be expressed, e.g., following the approach described by Leander *et al.* [9] or Knorr [10]. The mechanisms of enforcing access control policies are of great importance, and should ideally exhibit the same level of flexibility as the expressed policies.

In the previous work [11], different policy enforcement models for dynamic manufacturing systems have been introduced and discussed. The goal of this article is to describe and evaluate the implementation models for one of the most promising enforcement models from that article. The enforcement architecture uses a combination of local and centralized policy decision points, where the local decisions are static and the central decisions are dynamic, allowing a flexible and efficient architecture. The implementation uses Open Process Communication Unified Automation (OPC UA) [12] as a communication protocol, since it is the only available industrial protocol supporting policy-delegation mechanisms (to the best of our knowledge).

There are several previous works looking at quality metrics related to the OPC UA protocol, further discussed in Section 2. However, none of them look at the impact of the enforcement architecture, which we tackle in this article.

Problem statement. There is an increasing need for fine-grained and dynamic authorization in industrial manufacturing systems. There are available solution on how to express such policies. However enforcing the policies, and how the enforcement affects different quality metrics of the system is so far not widely explored.

Paper objectives. This article focuses on describing and evaluating the implementation of a policy enforcement architecture that deals with dynamic access control in industrial systems, using a workflow-based approach for policy decisions, and the widely adopted OPC UA protocol for communication.

Contributions. The following are our main contributions:

- Description and analysis of a tokens-based OPC UA authorization protocol, described in Section 3.
- Description of required implementations, described in Section 4.
- Experimental evaluation of impact of enforcement: (Sections 5 and 6)
  - Time to completion for session creation and resource requests.
  - Impact of token expiry time on resource requests.
  - Impact of token size on session creation.
- Two approaches on optimization of the connection establishment protocol, described and evaluated in Sections 7.
- Recommendations and considerations, discussed in Section 8.

Conclusions and ideas for potential future work are presented in Section 9.

2 Related Work

The enforcement architecture used in this article is based on suggestions from a previous work [11], there are however other suggestions and approaches of access

control enforcement architectures for industrial systems. E.g., Alcaraz *et al.* [13], discuss a policy enforcement system for the distributed smart grid, using authentication tokens similarly as us. Martinelli *et al.* [14] describes an alternative enforcement architecture for OPC UA supporting the Usage Control (UCON) policy model [15], adding an extra protocol layer for handling the UCON policy decisions. The focus of these works are on the description and formalization of the enforcement architecture, while we in this paper focus on performance evaluation of an architecture based on the OPC UA standard.

Several previous works look at performance metrics for different aspects of the OPC UA protocol. Cavalieri *et al.* [16] model a part of the OPC UA stack in a network simulator, evaluating the impact of, e.g., signing and certificate validation on connection establishment and data read, similarly as our work. Kohnhäuser *et al.* [17] investigate the feasibility and performance of secure OPC UA communication including e.g., connection establishment time for different combinations of security policies.

Rocha *et al.* [18] compare the performance of the OPC UA publish/subscribe mechanism with the Message Queue Telemetry Transport protocol (MQTT) [19]. Similarly, Burger *et al.* [20] look the OPC UA publish/subscribe, investigating memory and CPU consumption, reaching the conclusion that memory and network overhead are small, as well as usage of data encryption, while CPU utilization is identified as the bottleneck. The publish/subscribe mechanism is not covered in this article, but the observations on network and memory load v.s. CPU utilization are similar.

Silva *et al.* [21] evaluate a series of Internet of Things protocols, including MQTT and OPC UA, in an experiment measuring completion times related to data transport, similarly as this done in this article.

Ladegourdie and Kua [22] investigate the performance in terms of CPU and memory consumption on different sets of traffic scenarios, in an experiment also utilizing a RasberryPi as the OPC UA server.

All of these mentioned related articles investigate important aspects of the OPC UA protocol, and several of them evaluate similar performance metrics as done in our article, e.g., time for connection establishment and response time of signal reads. However, none of these previous works include the authorization flow in their measurements, which is the focus of this work.

3 Architecture

In this section the system architecture is briefly described and the authorization protocol analyzed to provide the theoretical foundation for the work to be presented. The goal of this article is to study how these theoretical constructs behave when deployed in practice.

An **Access Control Enforcement Architecture** describes required mechanisms and components related to access control, together with their respective placement in the system architecture. The architecture used in this work is based

on the approach suggested in [11], with basic components and placements as depicted in Fig. 1. The architecture is using components from the eXtensible Access Control Markup Language (XACML) reference architecture [23, 24], with the main Policy Decision Point (PDP) outsourced to an authorization service. It makes the active policy decisions upon a client request and returns the policy decision in the form of an authorization access token (AuthZ token). The client transfers the policy decision to the resource server that makes a local policy decision, based on the access token content and local policy data. The policy decision is enforced by the resource servers' Policy Enforcement Point (PEP).

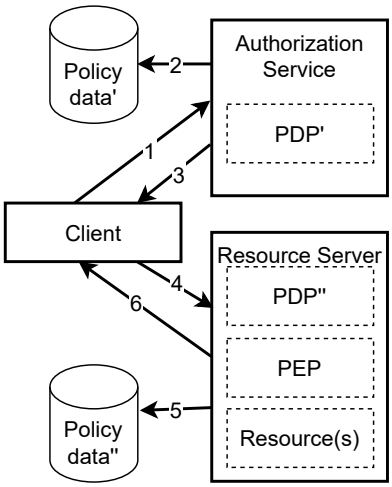


Fig. 1: An access control enforcement architecture, from [11].

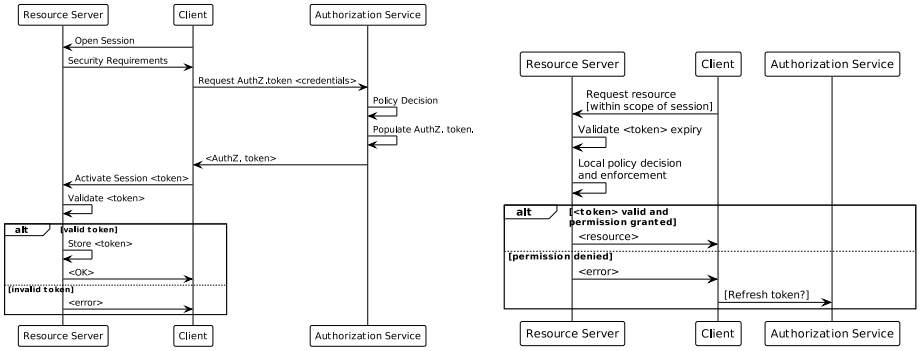
The **Authorization Protocol** used in the architecture is based on the implicit authorization flow of OPC UA[4], executed in two different phases. The primary policy decisions are taken during session establishment phase using the protocol described in Fig. 2a in which the authorization server makes the high-level decisions on valid permissions for the duration of the session encoded in an access token. The client acts as a mediator of the policy decisions by sending the access token as a part of the Activate Session call. The resource server validates the access token before the session is activated.

The second phase of the authorization protocol is executed when the client requests a resource from the resource server through the active session, following the protocol shown in Fig. 2b. The resource server validates token expiry and checks whether the requested resource permissions are included in the set of permissions granted by the central policy decision. If the resource is not granted, the client may attempt to fetch a new AuthZ token from the authorization service

[4] `reference.opcfoundation.org/GDS/v105/docs/9`

and refresh the session. If there are changed conditions implying new permissions the resource may be granted. The client must then refresh the session with a new token before token expires to avoid a new round of a session establishment.

The refresh token flow is identical to the session creation flow, except that the Client does not need to Open Session. The Client directly requests a new AuthZ token from the authorization service, and call Refresh Session, instead of Activate session.



(a) Create Session (b) Access Resource

Fig. 2: Phases of the Authorization Protocol Flow.

Session creation. The session creation procedure is conducted in three separate stages:

1. Open session.
2. Request AuthZ token.
3. Activate session.

Opening a session includes the following steps: 1) establishing a channel with the server; 2) sending client instance certificate; 3) receiving service instance certificate along with connection options for the server, including security options for secure communication and user authentication/authorization.

If the options for user authentication/authorization indicate that an AuthZ token is required, the client needs to request a token from the endpoint as designated in the security information received from the the resource server. The token is then used to activate a session with the resource server.

From the resource server perspective, the session establishment is done in two steps: 1) an open session requests arrives and the resource server replies with the list of security configurations it supports, including options for authorization; and 2) upon session activation, the received access token is validated and associated with the active session, if accepted. Several other checks are also done on session establishment, e.g., the client application instance certificate must be directly or indirectly trusted by the server. AuthZ token validation includes: 1) decryption

(if encrypted); 2) validation of the token signature and expiry time; and 3) control that the token is issued for the client of the session and for the resource server.

Communication through active session. Once the session is activated, the client may access resources if permitted by the resource server, e.g., browse the name-space, read values, write values and execute methods. In due time before session expiry, the client can request a new access token from the authorization service and refresh the active session. An expired session, regardless of content, cannot be used for resource requests.

3.1 Protocol Modeling

To verify some basic properties of the authorization protocol, a model is created using the UPPAAL [25] tool environment, illustrated in Fig. 3. The model has separate templates for a client, a resource server and an authorization service. This model does not contain any details on access token content, instead we assume that a correctly issued token will contain permissions for the desired resource. The client and resource server models contain all states and transitions related to the authorization protocol outlined above.

Using this model we show that resource requests are not possible unless the model is in an active session. Furthermore we show that outlined protocol must be followed in order for the client to be granted access resources.

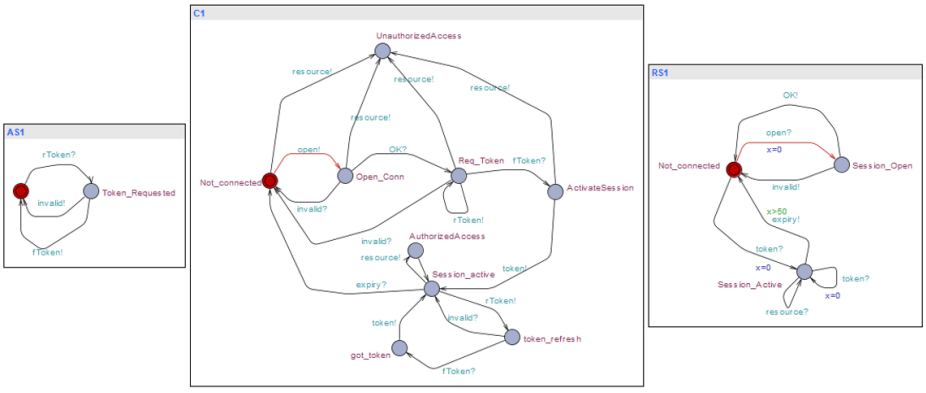


Fig. 3: Resource Server (RS1), Client (C1) and Authorization Service (AS1), modeled in UPPAAL.

Using the temporal logic we can verify that the modeled protocol works as expected, i.e., that the protocol is free from deadlocks (A[] not deadlock) and that it is possible to activate a session (E<> RS1.SessionActive and C1.SessionActive). Further, we check that the resource server cannot be in *not connected* state while the client is in *session active* (A[] RS1.NotConnected imply not C1.SessionActive).

Also, we were interested to see whether it is possible for the client to access a resource only by the *authorized access* state (E<> C1.AuthorizedAccess, A[] not C1.UnauthorizedAccess). All the checks have been successfully performed, showing no deviations.

4 Implementation

This section provides detailed information on the implementation done to support the evaluation work. Even though the authorization flow, as described in the previous section, is part of the standard, no available software stack fully supports the flow yet. The required implementation for supporting the authorization flow according to the standard is outlined below. The implementation uses the .NET stack implementation from the OPC foundation[5], as it currently has the best support for the outlined authorization flow, and, being open source it is quite easy to extend.

Resource server. All the basic logic for transmitting and receiving access tokens are implemented in the stack. However, token validation and handling of permissions based on token content has been implemented as part of this evaluation, as well as the behavior for handling token expiry.

Client. The available base-class for an OPC UA client is extended with the functionality needed to support the authorization flow:

- Decoding the user access token policy data as part of the security requirements returned from a server on open session.
- Establishing a session and request access token from the Authorization service according to data received from a resource server.
- Using the access token to activate the session.
- Managing a token renewal before the expiry.

The expiry time of the access token is an important aspect in enabling a sufficiently dynamic mechanism for permission delegation. It is the responsibility of the client to refresh an active session before the access token expires. Token renewal is implemented in a way that a new access token is requested from the authorization service when 80% of the token life-time has passed. As soon as the new access token is received, the session is refreshed.

Authorization Service. To support the evaluation experiment, a simple authorization service is implemented following the OPC UA standard[6], with the authorization service being accessible through an OPC UA server endpoint. On an AuthZ token request from a client, the implemented authorization service will always return a valid access token, with a configurable expiry time and a configurable size. This means that we have minimized the policy inference time of the authorization service.

JSON Web Tokens (JWT) [26] is used for encoding the policy decision, which is the preferred encoding according to the OPC UA standard.

[5] `github.com/OPCFoundation/UA-.NETStandard`
[6] `reference.opcfoundation.org/GDS/v105/docs/9.6.5`

5 Experiment

In order to evaluate the two phases of the authorization protocol, as outlined in Section 3, a set of experiments are constructed and they are executed twice, first in a system configured to use no authorization as a baseline, and second following the authorization protocol. Measuring the time to complete for different parts of the protocol allows a quantitative estimation of the operational properties of the authorization protocol, compared to the baseline.

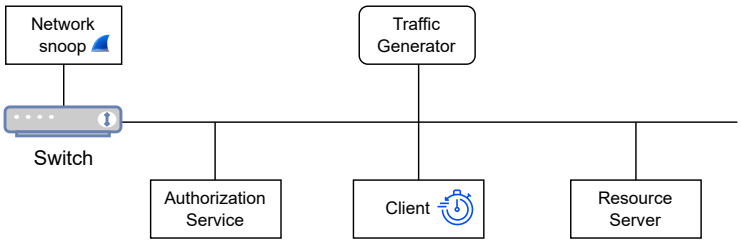


Fig. 4: Experiment setup.

In order to understand the sensitivity of the architecture in relation to traffic load, indicating its scalability, experiments are executed using two traffic load scenarios, *High traffic load* and *Low traffic load*. In the *Low traffic load* scenario, no additional traffic is generated. In the *High traffic load* scenario the client has additional connections to 5 other servers, where each accesses variables on 10ms intervals, representing approximately 5000 reads/s. Additionally, two clients are connected to the resource server, one making 6 resource requests (2 read, 2 write, 2 execute) on a 15ms clock, and one making 7 reads on a 10ms clock, representing approximately 1100 additional resource requests per second towards the resource server. Measuring network load, the high-load scenario generates approximately 1.2 Mbps traffic to the resource server and 2.1 Mbps from the server. The high-load scenario pushes the CPU load of the resource server to about 40% for each of its' four processor cores.

The *High traffic load* scenario is meant to represent a realistically high load for the resource server and client respectively.

The experiments are executed in a system containing a resource server, a specially developed client which can execute and measure the completion times, an authorization service, and a variable subsystem for generating the traffic load scenario. The system setup for the experiment is illustrated in Fig. 4.

During the experiments related to session establishment, the completion time for the three stages of the session establishment protocol is measured (i.e., open session, handle token, activate session). For the resource access phase of the protocol, experiments are performed for read and write of signals and execution of methods.

In total, this sums up to 16 individual experiments, with results summarized in Section 6. Each instance of the experiment is executed a fixed number of times, i.e., 1000 times for the connection experiments and 4000 times for the access resource experiments. The client is designed to perform experiment repetitions on a clock with some randomization. The time interval between each experiment sample is between 10ms and 2000ms. This is done so that the samples are not accidentally coinciding with any of the fixed-frequency cycles of the traffic generators.

Experiments are also performed related to the impact of different token sizes and token expiry times. The size of the authorization token may impact connection establishment time, prompting an additional run of the connection establishment experiment, using variable token size. An authorization token refresh may impact response times of resource requests, which is examined in an experiment using variable token expiry time.

Equipment. The equipment used in the experiment is meant to mirror the scenario of a relatively simple resource server, such as an industrial controller, communicating with a Human-Machine Interface (HMI) client running on a standard PC, and the authorization service running on a server machine.

A Raspberry Pi 4 Model B (ARM Cortex-A72) with Ubuntu 22.04 is used for running the resource server. The ARM Cortext-A72 is normally clocked at 1.5GHz and the majority of the experiments are performed using that configuration. A set of experiments is also performed with the processor down-clocked to 600MHz, to get performance comparable to that of a CI845[7], which is an industrial hardware platform used for running control and connectivity services.

The client and the authorization service are both running on separate commodity hi-spec PCs (Intel i7-11850H (8 cores), 2.5GHz, 32GB RAM, Windows 10). The Switch is a ZYXEL GS1915-8.

6 Results

In the following sections, the detailed results of the performed experiments are presented.

6.1 Results on Connection Experiments

The connection experiments are executed for low and high load scenarios, with and without token-based authorization, with each test executed 1000 times. In order to perform a more detailed analysis, the total connection time is separated into open session, request token and activate session, following the protocol flow in Fig. 2a. Results for the experiments are presented in Table 1.

When looking at the connection test results, an obvious additional cost when using access tokens is the time related to requesting the token from the authorization service. In our experiment this adds time representing a whole additional

[7] `800xahardwareselector.com/product/ci845`

Table 1: Results from connection experiments. All results are given in milliseconds where μ is the average value and σ the standard deviation.

Authorization	None		Token	
	μ	σ	μ	σ
Low load				
Open session	15.0	3.3	15.1	3.3
Request Token	0	0	115.2	12.8
Activate session	100.0	12.3	193.9	12.9
Total	115.0	13.2	324.3	19.3
High load				
Open session	17.0	10.5	18.7	14.4
Request Token	0	0	213.8	69.0
Activate session	124.8	23.2	223.1	25.1
Total	141.6	29.0	455.6	88.6

connect cycle, on average 115ms in the low load scenario and as much as 213ms in the high load one. For all these experiments, the client is creating a new session for each access token request. This points towards a first idea for optimizing the client implementation of the protocol by keeping the session to the authorization service open in the client.

Connections using access tokens not only increase in cost by the amount related to authorization service interactions of the client, a significant increase in time is needed also for the session activation, almost doubling the average session creation time. The experiment is not constructed to directly measure what in the session activation is most expensive, but a theoretical analysis of the protocol suggests the following possible sources for this extra time:

- Transport of the AuthZ token (which in this example is 1536 bytes after encryption).
- Encryption of the token on the client side.
- Decryption of token data on server side.
- Validation of token on server side.

The access token is already transported over an encrypted channel, so a second potential optimization of the protocol would be to remove the explicit encryption of the access token, which is added by the client.

There is a clear impact of the traffic loads for both authorization scenarios, especially on Activate Session and Request Token. By analyzing different traffic scenarios, we notice that network utilization and memory consumption for both the resource server and client are only marginally affected by the high load. What is hugely affected is the CPU load of the resource server, jumping from an average of 4% CPU utilization on each core in the low traffic scenario to about 40% for high traffic load.

6.2 Results on Access Resource Experiments

To evaluate the impact of the authorization protocol on accessing resources, three separate experiments are performed, for reading, writing and execution of a method, each one being executed 4000 times, for each of the different traffic scenarios. The results are presented in Table 2.

Table 2: Results from the resource access experiments. All results are given in milliseconds, where μ is the average value and σ the standard deviation.

Authorization	None		Token	
	μ	σ	μ	σ
Low load				
Read	3.2	1.5	3.3	1.7
Write	3.2	1.5	3.3	1.5
Execute	12.3	4.0	12.7	3.3
High load				
Read	2.2	1.6	2.2	1.7
Write	2.3	1.6	2.3	1.6
Execute	10.0	4.5	10.3	4.1

From the resource access experiment results, we can see that resource requests have similar completion time regardless of the used authorization protocol. This is expected, since the major overhead of the authorization protocol is related to session creation. In our experiment, a resource request related to read or write will on average take from 2ms to 3ms to complete, while the method execution has a completion time between 10ms and 13ms. It is worth noting that the method execution call is designed to return directly, i.e., there is a minimal amount of internal processing within the resource server related to executing the method.

The resource request times for the high-load scenario consistently outperform the low load scenario in our experiments. A theoretical analysis shows that the resource server handling of access requests is done by pulling working threads from a thread pool. A hypothesis is that in the low load scenario, the threads will be inactive before the next resource request arrives, while when there is a high load, this will not happen, meaning that the thread creation time is increasing the completion time of the low load resource requests. This is however out of the scope for this article to investigate, and is therefore not further explored.

6.3 Results on Different Token Expiry Times

Before an access token expires, the client of the session has the option to fetch a new AuthZ token from the authorization service, and use that to refresh the session. As demonstrated, session activation is a relatively expensive operation,

the impact of token refresh is however not evaluated. By lowering the session expiry time, the amount of session refresh calls are increased. However, measuring the completion time for a session refresh explicitly is not very interesting, as it is executed during the time a session is already open, i.e., it does not directly affect session establishment.

The token expiry time may however impact resource requests. To investigate this, the read-resource experiment is repeated, but this time using sessions with different token expiry times. The client is configured to automatically refresh the tokens. Results from 1000 runs for three different expiry times are summarized in Table 3, for the low-traffic scenario. Additionally, we also report the maximum time to completion.

Table 3: Results for read resource experiment with different token expiry times. μ is the average value and σ is the standard deviation.

Token expiry time	μ	σ	max	min
8s	4.6ms	9.7ms	139ms	2.0ms
28s	3.3ms	3.1ms	60ms	1.9ms
96s	3.1ms	1.8ms	45ms	1.5ms

Based on the gathered results, we can conclude that the expiry time does not affect the average value considerably, but the standard deviation is substantially increased with shorter expiry times. When a resource request coincides with a session refresh call to the resource server, the request may be delayed until the refresh is completed. The likelihood for such a coincidence is to a large extent depending on the expiry time, with a higher risk for a lower expiry time.

6.4 Result on Different Token Sizes

It is most likely the case that the different token sizes have impact on the session activation time, because of the cost for encryption and decryption of the token. To evaluate how the token size affects the connection time, the connection experiments are repeated, but with the issued AuthZ token having different sizes. A claim with configurable size has been added, after which the total token size was calculated (i.e., after encryption). For each token size, the connection experiment has been repeated 100 times. Results for average connection time in relation to size of the token are provided in Fig. 5. Please note that the session activation time remains stable (at around 220ms) until token size reaches 4000 bytes, after which the time to connect increases proportionally.

6.5 Results on Lowering the CPU Clock Frequency of the Resource Server

CPU utilization of the resource server seems to be a determining factor for the completion time for at least the session establishment part of the protocol. The

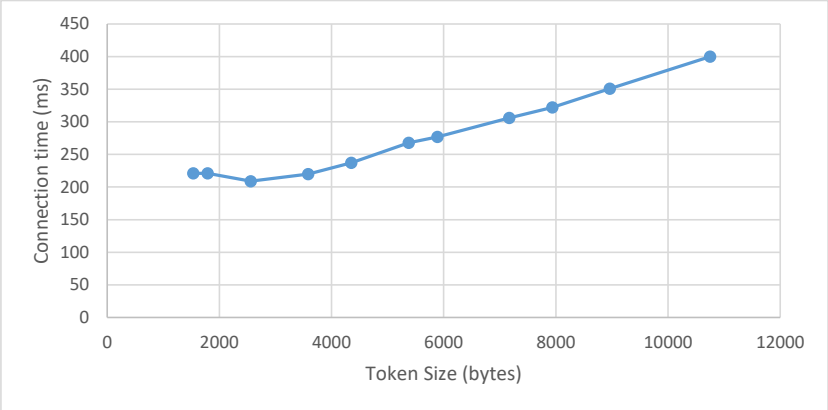


Fig. 5: Session activation time vs. token size

experiments so far have been done using a 1.5GHz processor. In the following we repeat some of the experiments, but with the clock frequency of the CPU of the resource server lowered to 600MHz (the lowest supported frequency of the Cortex A72), i.e., to 40% of the nominal performance. The aim is to get results comparable to the ones of an CI845, an industrial hardware platform developed by ABB used for various control service applications. Connection experiments results are given in Table 4.

Table 4: Results on connection experiment using downgraded resource server CPU. All numbers are provided in milliseconds, μ is the average value and σ is the standard deviation. Percentages in parenthesis are compared to the original experiment.

Authorization	None		Token	
	μ	σ	μ	σ
Low load				
Open session	18.4 (+23%)	6.1	19.9 (+32%)	10.0
Request Token	0	0	149.8 (+30%)	31.9
Activate session	163.5 (+63%)	31.4	216.7 (+12%)	33.7
Total	182.0 (+58%)	33.7	386.7 (+14%)	51.1
High load				
Open session	21.6 (+27%)	7.2	33.8 (+81%)	85.5
Request Token	0	0	216.1 (+1%)	84.5
Activate session	196.0 (+57%)	22.0	364.7 (+63%)	58.7
Total	217.7 (+54%)	25.2	614.7 (+35%)	211.7

Compared to the previous experiments in Section 6.1, the total completion time for connection establishment increases between 14% and 60%.

As the completion time for the different experiments related to accessing resources are very similar, only the read experiment using token authorization has been repeated in this setting, for low and high traffic scenarios. For low traffic, the average time for read was 4.4 ms (+38%), with $\sigma = 3.2$ ms and for high traffic average completion time was 3.8 ms (+72%), with $\sigma = 1.8$ ms. The results follow the patterns of the initial experiments, but with completion times increasing on a similar scale as the connection establishment experiment.

This confirm the assumption that the completion time in our experiment is highly dependent on the CPU power of the resource server. The standard deviation also increases significantly, especially in the high-load scenario. This implies that our selected high-load scenario may be beyond the limit of what this configuration of the resource server can handle while staying within predictable operational boundaries.

7 Suggestions on optimizations of session activation

Based on the analysis of the experiment results, we suggest two potential optimizations for the authorization protocol implementation: 1) In the client to keep the session to authorization service(s) open, and 2) to remove the explicit encryption of the access token if it is already being transported using an encrypted channel.

Both suggested optimizations are implemented and then evaluated using the same experimental setup as the initial experiments related to connection establishment. Results are provided in Table 5. As the optimization only affects the session establishment, the experiments for resource requests are not repeated.

Table 5: Experimental results of optimization of session activation. All numbers are given in milliseconds, μ is the average value and σ is the standard deviation.

Improvement	Keep session		Single encryption		Combined	
	μ	σ	μ	σ	μ	σ
Low load						
Open session	14.1	3.0	13.8	5.5	15.0	6.3
Request Token	7.3	4.4	128.5	32.7	7.1	4.1
Activate session	191.0	10.9	142.6	13.6	143.6	15.8
Total	212.5	13.4	285.0	38.3	165.8	20.8
High load						
Open session	15.4	9.8	16.1	15.0	13.1	3.5
Request Token	10.8	7.5	160.3	50.7	8.2	3.4
Activate session	208.3	25.2	151.8	35.5	148.1	12.5
Total	234.4	32.9	328.2	69.0	169.5	15.5

As can be seen, keeping a session to the authorization service provides a large performance increase on the token request part of the protocol, going from an average of 115ms (Table 1, low load req. token) down to 7ms in our experiment. Removing the double encryption enables a significant gain during the session activation phase, down to about 50ms in our experiment.

The best performance gain is reached by combining these two approaches, both caching the authorization service session and removing the double encryption. Using this combination cuts the connection time to between 50% and 37% of the initial implementation. This results in the total difference between using no authorization and using the authorization protocol with access tokens to be reduced to about 50ms in our experiment (115.0 ms with no authorization compared to 165.8 ms), and is even lower for the high-traffic scenarios.

Repeating the experiment with different token sizes with the single encryption optimization in place indicates that the size of the token no longer has an impact on the session activation time, i.e., in our experiments all the measurable additional time related to the size of the AuthZ token is related to the explicit token encryption.

8 Discussion

The performed experiments have shown some important and interesting properties of the authorization protocol. Authorization in the way it is implemented in this architecture, will have no measurable impact on individual resource requests, but have significant impact on session establishment. For sporadic resource requests that include establishing a new session to a resource server, a majority of the response time will be related to the session establishment. This is also the case for scenario without using the enforcement architecture.

The session establishment time is increasing when the traffic load towards the resource server is increasing. In particular, the standard deviation for the high-load scenario is increasing for the connection time when the enforcement architecture is used. This due to the authorization protocol containing several additional steps in which uncertainty is introduced. However, for resource requests, the higher traffic load has no adverse effect on completion times in our experiment. The architecture scales well with regards to resource requests, but may have issues for session establishment. This is even more visible for the results with a lower resource server CPU clock frequency. The completion time increases and are on average in the same order as the CPU performance downgrade, but the standard deviation for connection establishment using the enforcement architecture is almost tripled.

When analyzing the impact of using short-lived access tokens, the results point in a similar direction. The average completion time related to resource requests is close to the initial experiments, but the standard deviation increases as the expiry time is decreased. The risk of a resource request coinciding with the session re-activation call increases with a short token expiry time. For minimizing this impact, the client could be implemented to avoid resource requests while a

session-reactivation is on-going. With a shorter token expiry time, the flexibility of the access control mechanism increases. Generally, using a short expiry time in a large system will generate a lot of traffic both between clients and authorization services, and clients and resource servers.

For the initial experiments on session establishment, the completion time is on average three times higher when using the described authorization protocol. The majority of additional time is spent during token request and session activation. Combining two simple suggestions of performance optimization, the session establishment phase of the authorization protocol is brought to numbers comparable to the baseline scenario when using no authorization.

The first performance optimization is a pure client implementation, and is using the assumption that many resource servers will outsource their policy decisions to the same authorization service. Therefore it is a good idea to keep a session of an authorization service open to be used for subsequent AuthZ token requests. This will have a slight memory consumption hit for the client and authorization service. The exact impact for the authorization service is not further evaluated in this article. However, for a larger systems with many clients there may be negative scalability effects of that approach for the authorization service, especially on memory consumption. On the other hand, establishing a new session with the authorization service for each resource request and client will have a much higher impact on the authorization service CPU utilization.

The second performance optimization is related to encryption and decryption of the AuthZ token. If the client is communicating with the resource server using an unencrypted session, it is important that the AuthZ token is encrypted. Otherwise this provides an opportunity for token to be stolen and misused by a threat actor. However, if the session is already encrypted, there is no obvious need for the AuthZ token to also be encrypted. The default behavior of the .NET stack is to use asymmetric encryption of the AuthZ token, which leads to a huge additional time needed for the resource server to perform decryption of the token. Removing this "double encryption" is possible within the resource server OPC UA configuration. However, using this option may lead to other unwanted behavior of the server, e.g., the same option is used to remove validation of the client nonce on session establishment. This optimization will therefore need some additional rounds of analysis before being used in a real-world system.

If keeping the encryption of the AuthZ token, the size of the token will have a direct impact on the completion time for session establishment, as shown in Section 6.4. Therefore it is desirable to attempt to minimize the size of the token. For dynamic access control this may be a challenging task, as the policy decision from the authorization service will have to be expressed in a very detailed way. Further investigations by continuing the work in [11] are needed to find the right balance between the high granularity and sufficiently compact token encoding. Furthermore, there is a practical size limitation of the AuthZ token in the OPC UA .NET stack is set to 256kb, which should be sufficient for most needs, but may be an actual limitation in more complex scenarios.

8.1 Recommendations

Based on the experimental results and experiences from implementing the enforcement architecture, a few basic advice and recommendations can be provided.

From the client side one should **keep sessions active**, if several resources requests are likely to be performed towards the same resource server. This recommendation is applicable regardless of the enforcement architecture. It provides extra benefits with regards to authorization service sessions as described above.

Avoid double encryption, especially asymmetric encryption/decryption that is CPU intensive, and can cause a high penalty on a low resource embedded device.

If using token encryption, **keep token size small** as time to perform encryption/decryption increases with token size.

Find the **right balance for token expiry times**. A short expiry time will have an impact on scalability properties of the system, including resource request performance, as session refresh can interfere with ordinary resource requests. The longer the expiry time, the higher the risk of outdated permissions being used in the system. In the performed experiments, an expiry time of a few minutes have a rather low likelihood of negatively impacting performance, but this will depend on the size and complexity of the system.

If using the described enforcement protocol in a control-loop with real-time requirements, the completion time should be carefully measured in the target system, to guarantee that deadlines can be met and that the jitter for resource requests are kept at acceptable levels. In such a scenario it is most likely that there is no need for short token expiry times. Please note that there are other potentially more deterministic methods for real-time control in OPC UA, e.g., using the publish/subscribe pattern, which however cannot provide as flexible access control possibilities.

The typical use-case for this enforcement architecture would be for high level control and supervision, i.e., on the communication between an operator HMI and (several) resource servers, or for the workflow orchestration part of, e.g., modular automation, as described in [9]. In these use-cases the cycle-times are often not that high and may even be event-based, while the access control policies are more dynamic in nature, based on the currently executing workflows in the system.

8.2 Limitations and impact

The precise results of the described experiments are limited to the materialization and implementation of the hardware and software components used. For different system with other components, the completion times most likely will differ. However, we believe that the results provide good indications of the completion times for the described phases of the authorization protocol, especially the relative performance of the protocol compared to the baseline.

A drawback of using .NET to measure quality metrics is its lack of real-time characteristics, as e.g., memory management is out of control for the programmer. A garbage collection may occur at any time, which can have huge effect

on a particular measurement. Furthermore, operating system overhead may be larger and more unpredictable when using Windows (for the client) and Ubuntu for the resource server, as compared to using real-time operating systems. To counter these issues, we have repeated the experiments a large number of times to decrease the impact of sporadic disturbances.

As mentioned, the inference time of the authorization service is set to a minimum in the experimental setup. In reality, this inference time will of course be larger and have an impact on the performance of the session establishment. Measuring this impact is out of scope for this work, but may be interesting to look at as part of a future experiment.

9 Conclusions

Dynamic fine-grained authorization is a requirement for the future industrial automation and control systems, which will be network-centric, dynamic and flexible, using a Zero-trust security model. Very few technologies are available which can provide such characteristics for industrial systems. The OPC UA protocol is currently the best option, if using the authorization flow based on outsourced policy decisions to an authorization service.

In this work, we have analyzed, implemented and evaluated several aspects of this variant of access control enforcement architecture, something that is not previously done.

The experiments show that for resource requests there is no measurable additional cost of the authorization protocol. However for short AuthZ token expiry times, which will result in frequent session refresh calls, the standard deviation of the resource request is increasing. The expiry time is therefore one important design decision which is a trade-off between the level of dynamicity of the architecture and the standard deviation of resource requests.

There is a large difference in the connection establishment times between using fine-grained access control and no authorization, which is further impacted by increasing traffic load. However, two suggestions for optimizations are provided that limit this impact. One is related to the session handling in the client, while the other is related to avoiding double encryption of the AuthZ token. Combining these two optimizations more than halves the average connection establishment time compared to the initial approach.

Limitations. The experimental evaluation required a number of implementations to fully support the described authorization protocol. These implementations clearly have an impact on the results of the evaluations and any misinterpretations of the standard, or bad design decisions can limit the validity of the results.

In this article we aim to have an experimental scenario were the resource server and client are similar to what could be expected in an industrial system. However, the required implementation was not possible to run on an embedded real-time system. To partly counter this, the CPU-, memory- and network-utilization are measured during the experiments, leading to the realization that

the CPU-frequency is the deciding factor. This was confirmed by repeating some of the experiments with the CPU of the resource server clocked down to 40%, leading to results with equally longer completion time.

Future Work. The inference time of the Authorization Service is not accounted for in our evaluation, as the authorization service always returns the same claims, encoded in a correct token. Therefore, it is our plan to include the inference time for different variants of policy decision mechanisms in the authorization service in our future work.

Once there exist commercial or open-source implementations of the authorization flow, we would like to repeat the evaluation using a resource server running a real-time operating system on an embedded device.

Detailed threat modeling and analysis is not covered in this work, but is another important future investigation.

Acknowledgements

This work is supported by ABB AB; the industrial postgraduate school Automation Region Research Academy (ARRAY), funded by The Knowledge Foundation; and the Horizon 2020 project InSecTT. InSecTT (`www.insectt.eu`) has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 876038. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Sweden, Spain, Italy, France, Portugal, Ireland, Finland, Slovenia, Poland, Netherlands, Turkey.[8]

References

1. A. Sigov, L. Ratkin, L. A. Ivanov, and L. D. Xu, "Emerging enabling technologies for industry 4.0 and beyond," *Information Systems Frontiers*, pp. 1–11, 2022.
2. K.-d. Thoben, S. Wiesner, and T. Wuest, "Industrie 4.0 and smart manufacturing – a review of research issues and application examples," *Intl. Journal of Automation Technology*, January 2017.
3. Y. Lu, "Industry 4.0: A survey on technologies, applications and open research issues," *Journal of Industrial Information Integration*, vol. 6, pp. 1 – 10, 2017.
4. C. Zanasi, F. Magnanini, S. Russo, and M. Colajanni, "A zero trust approach for the cybersecurity of industrial control systems," in *2022 IEEE 21st International Symposium on Network Computing and Applications (NCA)*, vol. 21, pp. 1–7, 2022.
5. B. Leander, B. Johansson, T. Lindström, O. Holmström, T. Nolte, and A. V. Papadopoulos, "Dependability and Security Aspects of Network-Centric Control," in *28th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE, 2023.
6. J. Saltzer and M. Schroeder, "The Protection of Information in Computer Systems," in *proceedings of the IEEE*, vol. 63, pp. 1278–1308, September 1975.
7. S. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero Trust Architecture," tech. rep., National Institute of Standards and Technology, Gaithersburg, MD, aug 2020.

[8] The document reflects only the author's view and the Commission is not responsible for any use that may be made of the information it contains.

8. R. Sandhu, K. Ranganathan, and X. Zhang, "Secure information sharing enabled by trusted computing and PEI models," *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security, ASIACCS '06*, vol. 2006, pp. 2–12, 2006.
9. B. Leander, A. Čaušević, H. Hansson, and T. Lindström, "Toward an ideal access control strategy for industry 4.0 manufacturing systems," *IEEE Access*, vol. 9, pp. 114037–114050, 2021.
10. K. Knorr, "Dynamic access control through Petri net workflows," *Proceedings - Annual Computer Security Applications Conference, ACSAC*, vol. 2000-January, pp. 159–167, 2000.
11. B. Leander, A. Čaušević, T. Lindström, and H. Hansson, "Access control enforcement architectures for dynamic manufacturing systems," in *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, pp. 82–92, 2023.
12. "IEC 62541 OPC unified architecture," standard, Internation Electrotechnical Commission, Geneva, CH, 2016.
13. C. Alcaraz, J. Lopez, and S. Wolthusen, "Policy enforcement system for secure interoperable control in distributed Smart Grid systems," *Journal of Network and Computer Applications*, vol. 59, pp. 301–314, 2016.
14. F. Martinelli, O. Osliak, P. Mori, and A. Saracino, "Improving security in industry 4.0 by extending OPC-UA with usage control," in *15th Intl. Conference on Availability, Reliability and Security*, ACM, 2020.
15. J. Park and R. Sandhu, "The UCON$_{ABC}$ usage control model," *ACM Transactions on Information and System Security*, vol. 7, no. 1, pp. 128–174, 2004.
16. S. Cavalieri and F. Chiacchio, "Analysis of OPC UA performances," *Computer Standards and Interfaces*, vol. 36, no. 1, pp. 165–177, 2013.
17. F. Kohnhäuser, N. Coppik, F. Mendoza, and A. Kumari, "On the Feasibility and Performance of Secure OPC UA Communication with IIoT Devices," *Lecture Notes in Computer Science*, vol. 13414 LNCS, pp. 189–203, 2022.
18. M. Silveira Rocha, G. Serpa Sestito, A. Luis Dias, A. Celso Turcato, and D. Brandao, "Performance Comparison between OPC UA and MQTT for Data Exchange," *2018 Workshop on Metrology for Industry 4.0 and IoT, MetroInd 4.0 and IoT 2018 - Proceedings*, pp. 175–179, 2018.
19. "MQTT Version 5.0," OASIS Standard, March 2019. Edited by Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta.
20. A. Burger, H. Koziolek, J. Rückert, M. Platenius-Mohr, and G. Stomberg, "Bottleneck identification and performance modeling of OPC UA communication models," *ICPE 2019 - Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pp. 231–242, 2019.
21. D. Silva, L. I. Carvalho, J. Soares, and R. C. Sofia, "A Performance Analysis of Internet of Things Networking," *Applied Sciences*, vol. 11, no. 4879, pp. 1–30, 2021.
22. M. Ladegourdie and J. Kua, "Performance Analysis of OPC UA for Industrial Interoperability towards Industry 4.0," *IoT*, vol. 3, no. 4, pp. 507–525, 2022.
23. "eXtensible Access Control Markup Language (XACML) Version 3 . 0 Plus Errata 01," OASIS Standard incorporating Approved Errata., July 2017. Edited by Erik Rissanen.
24. V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone, "Guide to Attribute Based Access Control (ABAC) Definition and Considerations," tech. rep., NIST, 2014.
25. J. Bengtsson, F. Larsson, K. Larsen, P. Pettersson, and W. Yi, "UPPAAL - a Tool for Automatic Verifictation of Real-Time Systems," DoCS Technical Report Nr 96/97, Uppsala University, January 2016.
26. M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)." RFC 7519, May 2015.