MASTER'S THESIS | LUND UNIVERSITY 2018

Generic Event Integration in Video Management Software

Julius Barendt, Kim Fransson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2018-37

Generic Event Integration in Video Management Software

Julius Barendt
jur12jba@student.lu.se

Kim Fransson
dat13kfr@student.lu.se

August 8, 2018

Master's thesis work carried out at Axis Communications.

Supervisors: Pontus Evertsson, pontus.evertsson@axis.com
Niklas Hansson, niklas.hansson@axis.com
Boris Magnusson, boris.magnusson@cs.lth.se

Examiner: Ulf Asklund, ulf.asklund@cs.lth.se

Abstract

Today physical security systems consist of many different devices such as door controllers and cameras. It can be hard to monitor these devices by hand, and to make this easier these devices can send out events each time something interesting has happened. These events would then be displayed in a log where an observer could read them. The problem today is that there are many different ways hardware could be changed or upgraded, and these modifications could add new types of events. Each time this happens, the software log will also need an update to display the events correctly. This master's thesis will analyse how the events could be structured so that software could present them directly without making changes to their code, and what others features could be added to make events even more useful in a security system. In the end, several desired features were identified, such as filter the events or trimming the data down to only relevant information. An event structure with an accompanied API was developed to allow all this generically.

Idag består många säkerhetssytem utav flera antal hårdvaruenheter, så som kameror eller passersystem. Med många enheter kan det vara svårt att hålla koll på allt som händer. För att göra detta lättare så kan enheterna skicka ut meddelanden varje gång något intressant har skett. Dessa meddelanden kan sen visas i en log där de lätt kan läsas. Problemet som uppstår idag är att när hårdvaran uppdateras så kan nya typer av meddelanden läggas till, då måste även mjukvaran som presenterar loggen uppdateras för att kunna visa meddelandena på ett korrekt sätt. Detta examensarbete går ut på att identifiera ett sätt där meddelandena skulle kunna visas i loggen utan att behöva uppdatera mjukvarans kod. Olika sätt att få meddelandena att bli mer värdefulla för loggens läsare skall också identifieras. I slutändan hittades flera olika sätt att göra meddelandena mer användbara, så som att kunna filtrera dem eller att skära ner dem så att bara relevant information visas. Ett sätt att strukturera meddelande så de kan visas generiskt utvecklades också tillsammans med ett API som en logmjukvara kan implementera.

Keywords: Protocol design, API, surveillance systems, Video Management Software

Acknowledgements

We would like to express our gratitude to Boris Magnusson and Ulf Asklund at LTH for their academic guidance and support.

We would also thank Axis Communications for making this thesis possible by allowing us to perform the work at their company as well as providing us with knowledge and necessary hardware.

Contents

Chapter 1
Introduction

Security systems today often contain a large number of devices such as cameras and door access controllers, specialised software is also often used to make it easier to monitor these systems, called Video Managment Software or VMS. This software acts as a hub where all devices can be controlled, and each camera feed can be viewed in real time. It is, however, almost impossible to watch 100s of cameras at once, so even with a VMS, it can be hard to keep track of everything that is happening. In this case, it would be handy if the VMS could display everything that happens in some form of event-log, where information is more concentrated and can be processed in retrospect by the person monitoring the system.

What this thesis aims to do is to investigate how events should be structured for a log to be possible, as well as to identify additional features that would make an event log event more useful to an end user.

1.1 Problem description

Today the development of AXIS devices and VMS is intimately connected. This close connection is usually a good thing, and it means that the VMS is tailored to handle everything the different devices throws at it, this could be features such as turn on a LED on the camera, or event information, e.g., a door has been opened. AXIS also develops their own VMS called Axis Camera Station (ACS), but this is not the only VMS alternative and ACS coexists with many different VMSes developed by other companies.

The main drawback of the system today is that when new features and events are added to the device, the VMS must also need to be updated to accommodate the new features and events. These changes includes a way to activate the new features or making sure events can be presented to a person in a useful way. There are many different ways devices can be changed, a new sensor is released, a new application is installed on the device or the firmware is updated and this is a real hassle when the VMS needs to be updated each time.

For a big company like AXIS, this is a pain, but it is doable. However, for smaller VMS vendors the time and money needed are just not worth it, and the new features are left unsupported. Obviously this is not ideal, and it is bad for the VMS developer but also for AXIS when end users end up without acces to the latest features.

AXIS wants to find a way to make the changes to devices less of a burden for VMS developers. Ideally, the device should be able to explain its features to the VMS which then can display them without having to be updated each time a new feature or event is added.

1.2 Current solution

The solution to the updating problem is to come up with a way of separating the VMS development and the development taking place on the device. On the device the development could include changes to the hardware itself, updating of the firmware or installation of new applications running on the device, called ACAPs[17]. For features that can be controlled with a button in the VMS such as toggle a light or activate loitering detection[1] this is straightforward enough, and Axis has already come up with a way this could be done called "On-Screen Controls"[15], referred to as OSC for the duration of this report. OSC is based on a simple concept, illustrated in figure 1.1, where the VMS sends a request to the device asking it what features are available. The device will then answer with a list containing information about each feature, such as names and information about how to activate it.

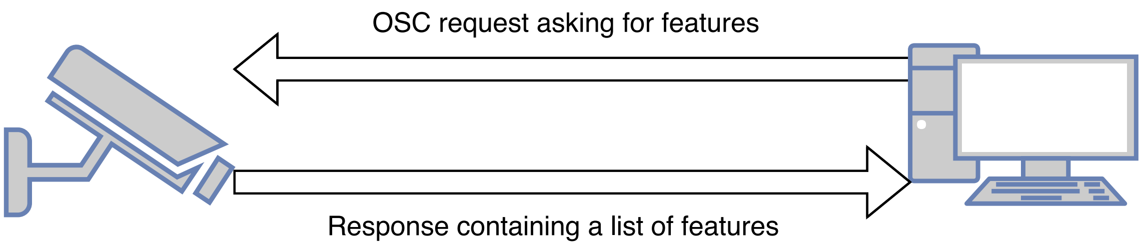

OSC request asking for features
Response containing a list of features

Figure 1.1: OSC workflow

This solution works because the device features can be triggered by sending VAPIX or ONVIF request[16], which the devices support. The device includes the request-URL used to trigger them in the response. The VMS is then able to create a button, with the features name as text, which uses the provided URL when pressed. The VMS is now ready to let the end user use the new features without having to make changes in the VMS. OSC also works when the VMS wants to know what languages are supported by the device.

The VMS developers only need the one-time commitment to implement OSC. After that when a new feature is added to the device, it is up to the device developer to update the list of responses with the new feature, when that is done the VMS will be able to handle it just like any other feature. The device feature- and VMS-development has now been effectively separated. However, OSC is only used for button based features and AXIS now wants to find a similar solution for generic presentation of events.

[1]A feature that detects if people are lingering in an area for too long.

1.3 Thesis outline

Chapter 1 - Introduction This chapter will introduce the main problems that VMS and device development face today.

Chapter 2 - Background and related work This chapter will discuss related work and what has already been done by AXIS.

Chapter 3 - Problem identification methods The different methods used to identify what problems needs to be solved will be presented here.

Chapter 4 - Problems and Approach This chapter will further discuss the problems found, as well as explained the approach taken to solve them.

Chapter 5 - Prototyping To test the solutions prototypes were made, this chapter will explain these prototypes.

Chapter 6 - Evaluation and Results The results obtained from the prototype testing of the approaches will be presented here. As well as a description of how the results were evaluated.

Chapter 7 - Discussion This chapter will discuss the results in more detail.

Chapter 8 - Future work Will explain what needs to be done in the future.

Chapter 9 - Conclusion Summary and comments on the thesis as a whole.

Abbreviations

ACAP Axis Camera Application Platform

ACS Axis Camera Station

IPC Inter Process Communication

ONVIF Open Network Video Interface Forum

OSC On-Screen Controls

PACS Physical Access Control System

REST Representational State Transfer

TDD Test Driven Development

Terminology

end user is the person who actually uses a particular product.

event2 A system that handles event declaration and creation on AXIS devices.

external event is an event that is sent outside the device. Such an event is interesting for external consumers such as the ACS.

internal event is an event that is sent inside the device. Such an event is interesting for internal processes inside the device.

nice name A string describing data and is intended to be understandable to a human.

Chapter 2

Background and related work

This chapter will be an introduction to the related work and background that will be used as a foundation for this thesis. As well as give insight into different key-parts needed to understand the problem and what limitations exist that needs to be solved.

2.1 Event

In the context of this report, event refers in general to something that has happened on a device, or on a peripheral connected to the device. An event could be movement detected on a camera, or a person scanned their ID-card on an access control system.

Technically an event is a collection of different key-value pairs. For example, if a user scans a key-card, the scanned card event would have a key "CardNumber" with the key-cards number as the value, or if a door is opened the event could have a key "DoorStatus" with the value "Opened". The number of key-value pairs an event has will differ from event to event, depending on how much data is relevant to it.

Events can be further divided into two categories, State less and State full, and all events can be placed in one of these two camps.

A *stateful event* is the name for events that represent some form of state on the device. A lot of events are stateful, motion detected is either true or false, a door is either open or closed, and LED-lights are enabled or disabled. Events like these will be possible to get information about even when nothing has happened to change a status.

Stateless events cannot be associated with a state and will only be created when something happens and then disappear, it's not possible to get the status of these events. Examples of these types of events are when a person scans their ID-card or when a door is forced open.

Regardless of whether events are stateful or stateless, they are created in the same way on the device. They are also sent to and received in the same way on the VMS part of the exchange.

2.1.1 Metadata

Metadata is data used to describe or add additional information about other data. The association with events are not clear-cut in this case, but the type of event-data treated as metadata in this report is mainly two things; when the event happened, e.g. a timestamp and on which device it happened, e.g. the device-ID. Some events may have more metadata.

2.1.2 Values

Today the biggest issue with event reporting is to make them human readable, for just the event name or description it would be simple enough to attach a nice name in plain text and send it as an additional field of data with the event. However, for the events with changing values, this is not as simple. The values are represented as they are in code, with boolean values being either 0 or 1. If values are presented to a human in this way, it could be confusing. The 0 or 1 also could mean different things in different scenarios. In the context of a "door open" event the 0 is "closed", and 1 is "open", but another event may want the 0 to be "denied" and 1 to be "granted". Because of this a VMS cannot make a simple mapping saying that 0 is "false" and 1 is "true", the event itself needs to be able to describe the presentation of the values.

2.2 Onvif

Open Network Video Interface Forum or ONVIF for short is an open protocol with the purpose of standardizing communications between network-connected security devices[10]. ONVIF is divided into several profiles, and the one most relevant to this thesis is profile C designed for Physical Access Controll Systems(PACS)[9].

The ONVIF standard contains a list of features and events that devices need to implement. All the events need to have *topics*, it can be made out of many topics but needs to have a minimum of one. During this thesis events will be classified by the combination of topics it has. When displaying the topics they are often separated by a "/" character as seen in listing 1. For example, an event describing if a door is open or not will have three topics and look like listing 1. From this its clear that the incoming event will be about a door, it will contain a state and the state will be its physical state. These topic combinations are used to filter the events quickly and group them by relevance.

```
Door/State/DoorPhysicalState
```

Listing 1: Topics from a door status event

To have a standard to conform to will ease the work quite a bit as an assumption about how the events will look can be made on the VMS side.

2.3 VMS

Video management system(also called video management software, or video management server) is a central component of a security system mainly consisting of cameras. However, it is possible to integrate many other security devices to a VMS, such as access control systems or speakers. So the video in VMS can be misleading. In this thesis, VMS refers to a system which is integrated not only with cameras, but to many different device categories.

A VMS is used in general as a hub controlling all devices connected to it, and it is also used to collect data from the different sources. For example video from cameras. It can record and store this data for future inspection and provides an interface for the end user to interact with the data, such as view the video both live and offline.

Usually, devices have a lot of features that can be used and needs to be integrated with the VMS. There is many different ways to do this, and Axis has come up with a solution to generically implement these features, without having to update the VMS, called on-screen controls, which is described in section 2.4.4.

Devices can send events to event consumers such as a VMS. The VMS will have to handle these events by itself, having to make specific actions for each event. One way to handle them is to display all the events for the end user. The end user then needs to search and filter through the events. Another way is to implement a rule engine that executes actions when different events occur, such as start to record on a camera when a door is opened.

AXIS develops their own VMS called Axis Camera Station(ACS), and it is essential to dig in deeper into how the ACS handles events to get a better understanding of how a commercial VMS handles events. This investigation will shed some light on what limitations exist, and how to work around them.

2.3.1 Event handling on ACS

The core event handler in the ACS is its rule engine. The rule engine consists of several rule entities. Each rule has its own set of triggers and actions. The real event consumers in the ACS is the triggers. The triggers consume events from different sources. Actions specify what the rule should do when one or more triggers are activated. For example, there exist in the ACS a MotionDetected-trigger which consumes motion detection events from cameras that support motion detection and a Send-Email action. A rule can say that; when a MotionDetected-trigger has consumed a motion detection event an email will be sent to the end users email. Thus notifying that motion has been detected.

To configure an event rule in the ACS, the end user is first prompted to add a trigger (see figure 2.1). The end user can select many different events. All of these events (except Device Event) are pre-defined in ACS and can be configured using specialised dialogues tailored for these events. The pre-defined events are of no interest in this master thesis. Device Events is the collective term for all events specified by a device, and these are not pre-defined. This process is also referred to as subscribing to events. The configuration itself can be refereed as a subscribe description.

When the end user selects to have a device event as a trigger, the user needs to specify which connected device the event will come from. In figure 2.2 the AXIS A1001 is chosen. When it is time to chose what kind of device event to use, the ACS will contact the device and ask for what kind of events it produces. A1001 will send back a list of events supported by the device.

When an event is chosen the end user is shown different filter options. These filter options are fetched from the device event. However, these filters are hard to interpret. For example, in figure 2.3 what exactly does *AccessPointToken* ,*CredentialType*, *CredentialHolderName* and *CredentialToken* mean, and what kind of values are valid for these filters?

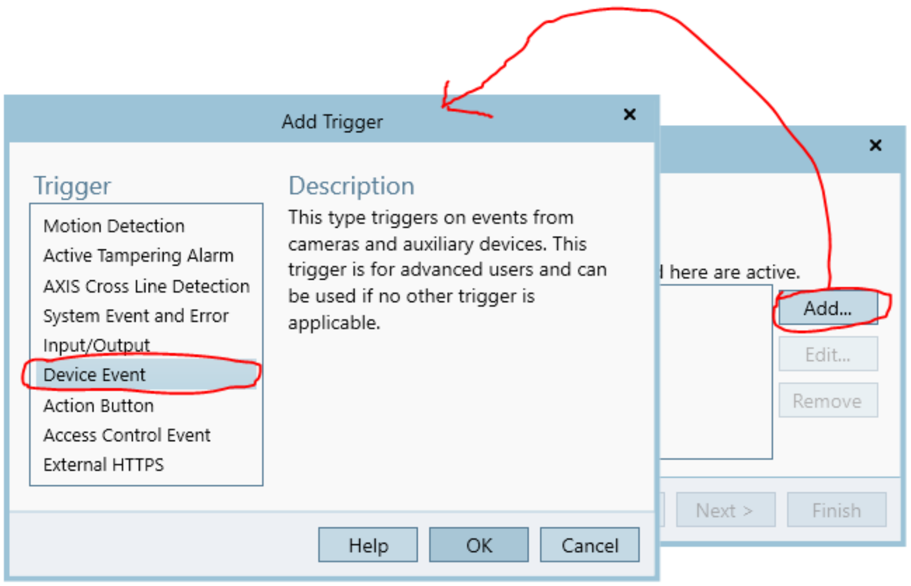


Figure 2.1: User dialogue in the ACS for specifying a trigger

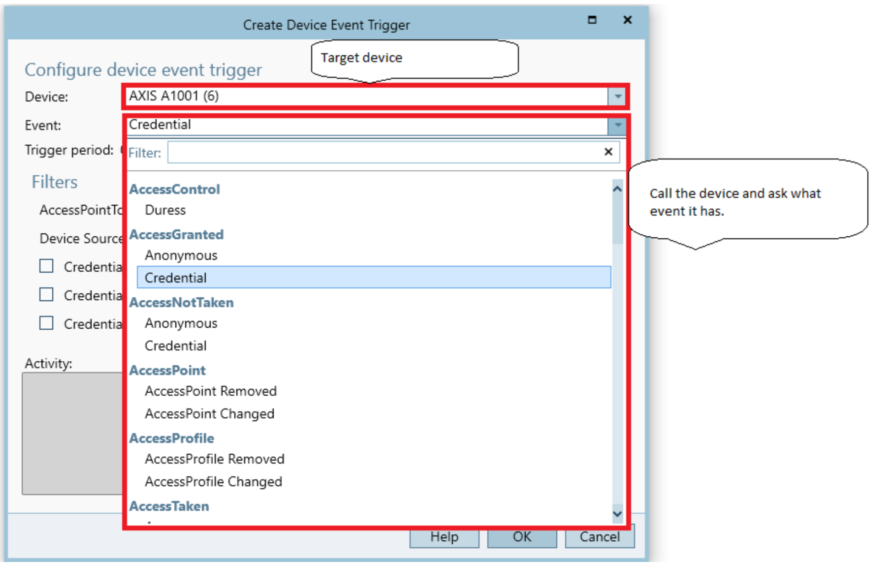
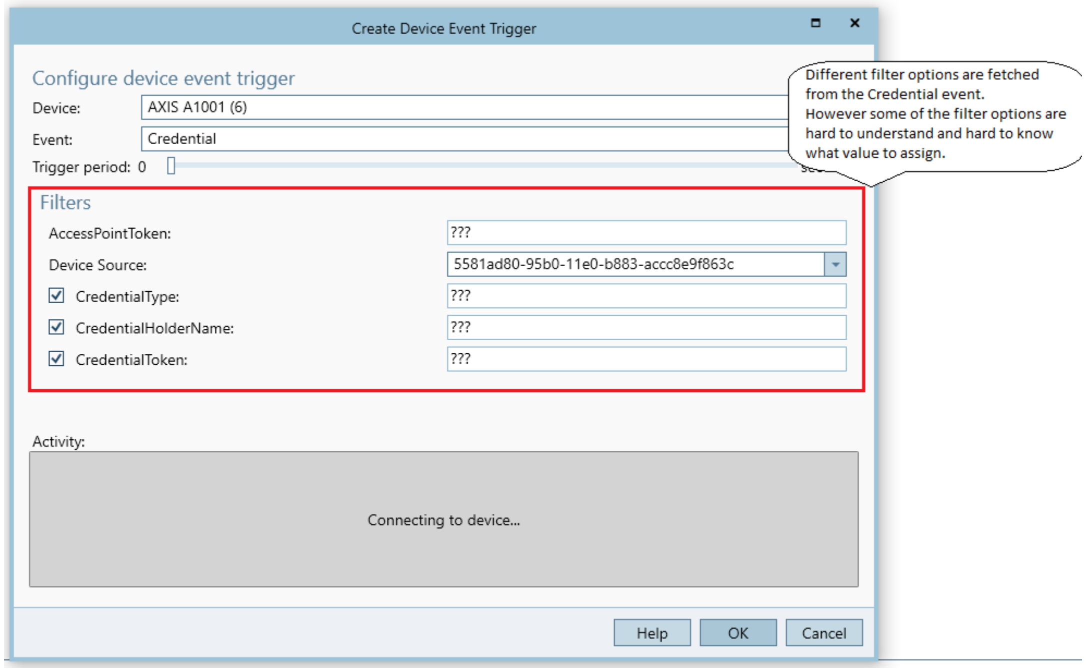


Figure 2.2: User dialogue in the ACS for specifying which device event type the trigger should generate.



Create Device Event Trigger

Configure device event trigger

Device: AXIS A1001 (6)

Event: Credential

Trigger period: 0

Different filter options are fetched from the Credential event.
However some of the filter options are hard to understand and hard to know what value to assign.

Filters

AccessPointToken:	???
Device Source:	5581ad80-95b0-11e0-b883-accc8e9f863c
☑ CredentialType:	???
☑ CredentialHolderName:	???
☑ CredentialToken:	???

Activity:

Connecting to device...

Figure 2.3: User dialogue in the ACS for configuring which filter values should be used.

After the trigger configuration, the user can specify which action should be taken when an event occurs. These details are not necessary for this section. Once a valid rule configuration is set up, the ACS will instantly try to contact the device and subscribe to the event.

The rules configuration are stored in the ACS database, and this configuration is used as a filter for polling events from connecting devices. One problem that may occur here is when a device has been reset or upgraded and no longer supports a specific event description that may be used in a rule configuration saved on the ACS database. If this happens, the ACS will no longer be able to get any triggers when polling events from the event stream. To fix this, the administrator needs to remove or to modify the configuration manually until it becomes valid.

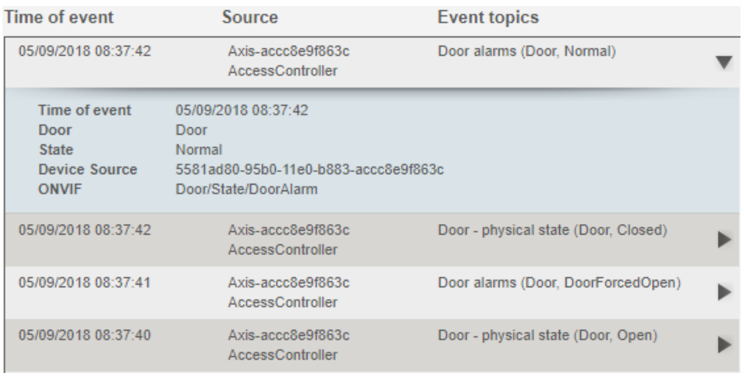
2.4 Axis devices

AXIS develops many different devices which in the context of this paper are any AXIS developed and network connected hardware product that runs an embedded system capable of reporting events. The primary focus throughout the report will be on the device A1001 that will be explained in more detail in section 2.4.1.

2.4.1 A1001

The A1001 which is a network door controller[14]. Which runs a custom Linux distribution and development is mainly done by adding so-called *daemons* to the system. A daemon is an application running in the background without user interaction[1]. These daemons can be developed independently from each other and are written in C.

The A1001 itself produces a lot of events some of them are tampering detection, scheduled events and changes in configuration[14]. It is also possible to attach peripherals, for example, a card scanner or a door contact switch[1]. When an event occurs on one of these peripherals, it will notify a daemon running on the A1001 device, and this daemon will in turn produce an appropriate event on the A1001 as well. Today an end user can view a complete log of all events using a web browser by accessing a website hosted locally on the A1001, this log can be seen in figure 2.4.



Time of event	Source	Event topics
05/09/2018 08:37:42	Axis-accc8e9f863c AccessController	Door alarms (Door, Normal)

Time of event	05/09/2018 08:37:42
Door	Door
State	Normal
Device Source	5581ad80-95b0-11e0-b883-accc8e9f863c
ONVIF	Door/State/DoorAlarm

Time of event	Source	Event topics
05/09/2018 08:37:42	Axis-accc8e9f863c AccessController	Door - physical state (Door, Closed)
05/09/2018 08:37:41	Axis-accc8e9f863c AccessController	Door alarms (Door, DoorForcedOpen)
05/09/2018 08:37:40	Axis-accc8e9f863c AccessController	Door - physical state (Door, Open)

Figure 2.4: The event log hosted on a A1001 device

This log shows the time when the event occurred, source is which device the event happened on and the event topics column displays a short text describing the event.

The weblog works but is not ideal. When displaying the events in the log every one of them needs a describing name to be understandable. To get these names, the device has to fetch them from a long and hard-coded list. If new events are added during run-time, they would then not have a human-readable name unless the list was also updated. This list is also internal and readable only by the device. An outside source, such as a VMS, is not currently able to get a hold of it.

2.4.2 Event2

The event2 structure is a way to handle and control the flow of both internal events and external events produced on AXIS network products, it will be referred to as event2 during this report. Internal events are events that will not be sent to a VMS, but rather only used internally by the device. Likewise external events are the ones that the end users will see. The event2 system consists of three component types, Event Producers, Event Consumers and Event switch (see figure 2.5).

Before any events can be sent, they need to be declared, and the role of the Producer is to declare events to the Event switch. It is also responsible for sending events to Consumers.

Consumers subscribe to events and receive them from Producers. They can also request updates of stateful events.

[1]a device that uses magnets to determine if a door is open or not

Finally, the heart of the event2 system is the Event switch. Its role is to store event declarations from producers, storing subscriptions from consumers and match event declarations with subscriptions and finally sets up connections between producers and consumers.

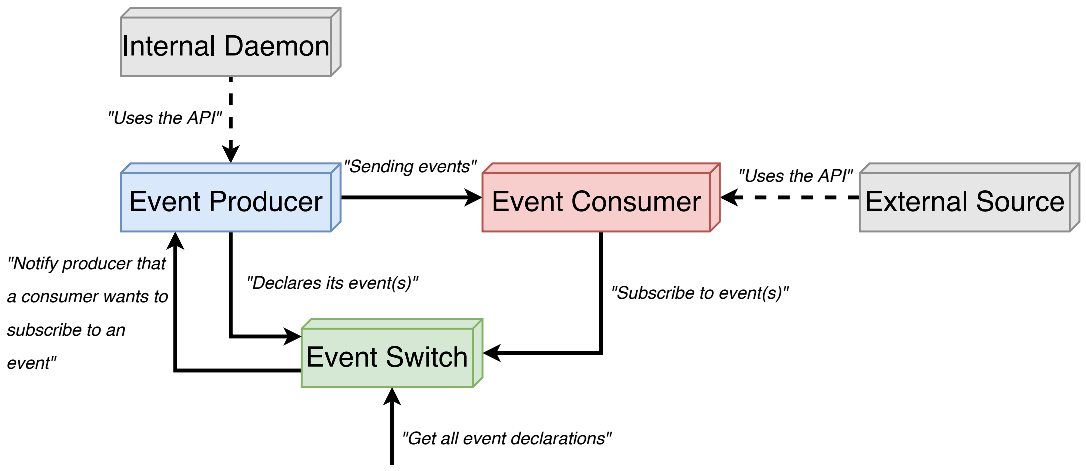


Figure 2.5: Simplified model of the event2 system

Figure 2.6 explains the sequence how a producer declares an event to the event switch.

First the event must be declared either by constructing an XML representation (see listing 2). To construct the representation using XML is also referred to as statically declaring which can be useful if an application does not need to change the event declarations during their lifetime. However if the application indeed will change its event declarations under its lifetime, the preferable way is to dynamically construct it in the C-code as seen in listing 3, for simplicity, only the most common methods are shown. The methods are self-explanatory, and no further deeper details are needed for this example.

DBus is a system for IPC and is used to communicate between process [18]. It is not necessary to go more into detail about DBus for our problem. The DBus object can be seen as the address to the producer and will be needed for communications between the switch and producer.

1) In the message "Declare" the event declaration is sent as the event_declaration parameter, and serialized as a byte array, together with the DBus object in the event producer and a human-readable name representing the event producer.

2) When the process of declaring the event is completed the event switch will send back an integer identifier serving as an acknowledgment. This identifier also known as GlobalDeclareId is used to access the event declaration in the Event switch when a producer wants to create an event.

Figure 2.7 explains how a consumer subscribes and consume an event E. First, the consumer needs to specify a subscribe expression, representing the event(s) the consumer is interested in. This expression is serialized as a byte array. This expression is the same as the event declaration. This subscription is sent together with the DBus object in the event consumer, a consumer token representing the subscription in the consumer and a human-readable name representing the event consumer. The DBus object will serve the same purpose as described above. It will be needed for communications between the switch and also necessary for the producers to know where to send the events.

The Event switch then tries to match the expression with an existing event declaration that has been stored earlier. If it is a match, the switch will then forward the subscription to the target producer and set up a connection between the producer and the consumer. All communication regarding sending events are now between the producer and the consumer. However, adding or removing declarations or subscriptions still needs to go through the event switch.

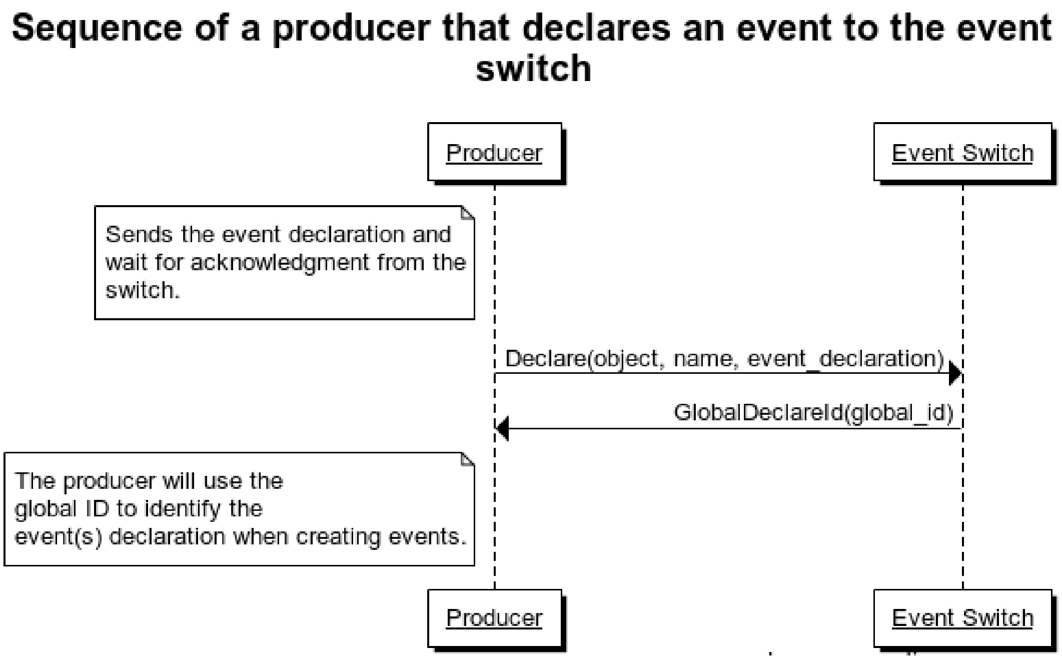

Sequence of a producer that declares an event to the event switch
Producer
Event Switch
Sends the event declaration and wait for acknowledgment from the switch.
Declare(object, name, event_declaration)
GlobalDeclareId(global_id)
The producer will use the global ID to identify the event(s) declaration when creating events.
Producer
Event Switch

Figure 2.6: Simplified sequence diagram of how a producer declares an event

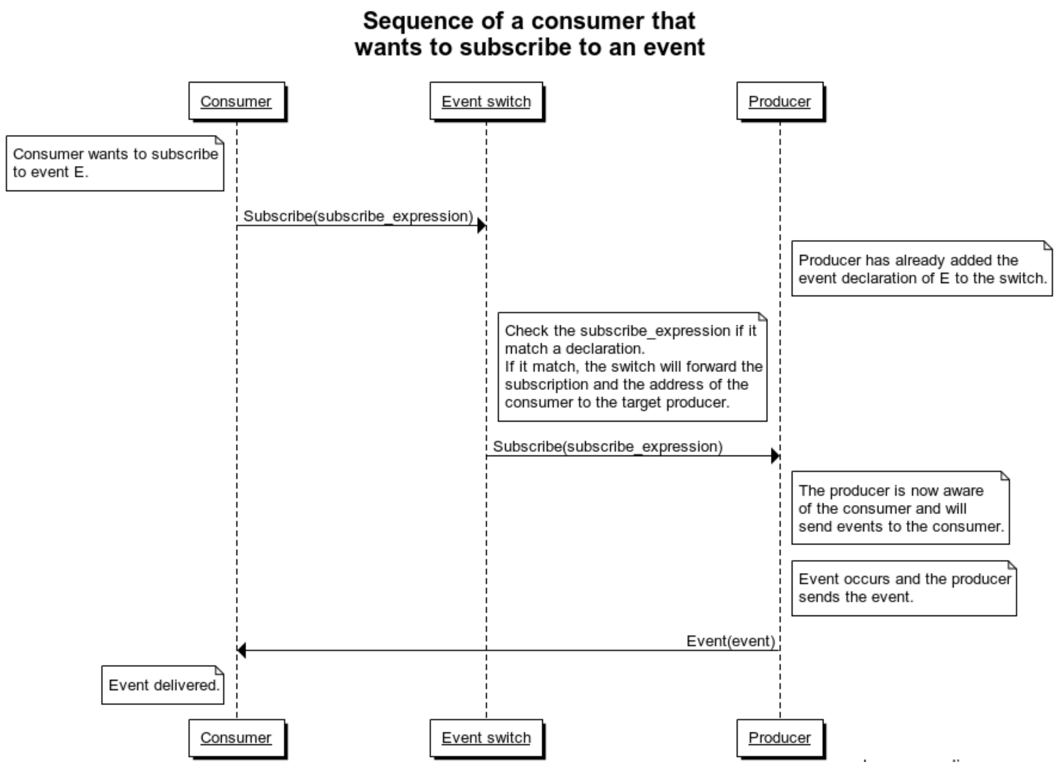

Sequence of a consumer that wants to subscribe to an event
Consumer
Event switch
Producer
Consumer wants to subscribe to event E.
Subscribe(subscribe_expression)
Producer has already added the event declaration of E to the switch.
Check the subscribe_expression if it match a declaration. If it match, the switch will forward the subscription and the address of the consumer to the target producer.
Subscribe(subscribe_expression)
The producer is now aware of the consumer and will send events to the consumer.
Event occurs and the producer sends the event.
Event(event)
Event delivered.
Consumer
Event switch
Producer

Figure 2.7: Simplified sequence diagram of how a consumer subscribes and consumes an event.

```
{
<declare>
   <expression>
      <keyvalue
        key="topic0"
        value="Device"
        value-nice-name="Device"
        name-space="tns1"/>
      <keyvalue
        key="topic1"
        value="SystemMessage"
        value-nice-name="System message"
        name-space="tnsaxis"/>
      <keyvalue
        key="topic2"
        value="ActionFailed"
        value-nice-name="Action failed"
        name-space="tnsaxis">
        <tag tag="client-event"/>
      </keyvalue>
   </expression>
</declare>
}
```

Listing 2: Statically declared events using XML

```
{
event_declaration_add_key_value(declaration, "topic0",
    "Device", VALUE_TYPE_STRING);
event_declaration_add_key_value(declaration, "topic1",
    "SystemMessage", VALUE_TYPE_STRING);
event_declaration_add_key_value(declaration, "topic2",
    "ActionFailed", VALUE_TYPE_STRING);

event_declaration_set_name_space(declaration, "topic0",
    "tns1");
event_declaration_set_name_space(declaration, "topic1",
    "tnsaxis");
event_declaration_set_name_space(declaration, "topic2",
    "tnsaxis");

event_declaration_set_nice_names(declaration, "topic0",
    NULL, "Device");
event_declaration_set_nice_names(declaration, "topic1",
    NULL, "System message");
event_declaration_set_nice_names(declaration, "topic2",
    NULL, "Action failed");

event_declaration_add_key_tag(declaration, topic2,
    "client-event");

}
```

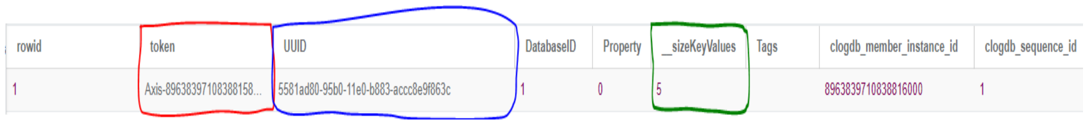
Listing 3: Dynamically declared events using C, simplified

A short explanation of how the events are stored. Both event declarations and events are stored in an SQLite3 database stored on the network device. One table *Declaration* (see figure 2.8) where the over all structure of the event is stored. The *token* column is a unique key for identifying the event declaration, the *UUID* column is the device that produces the event, and lastly, the *sizeKeyValues* column states how many keys (also referred to event properties) the event declaration has.

Another interesting table is the *Declaration_KeyValues* table where the event properties are stored. In figure 2.8 we see that the event declaration has 5 event properties. In figure 2.9 we see that we have 5 rows with the same token value. This value refers to the corresponding event declaration that has these properties. Important columns is *KeyValue_key*, *KeyValue_value* and *KeyValue_Tags*. The two first is self explanatory but the third column KeyValue_Tags stores some more specific characteristics of the property, such as nice names. The tags are stored in the form of:

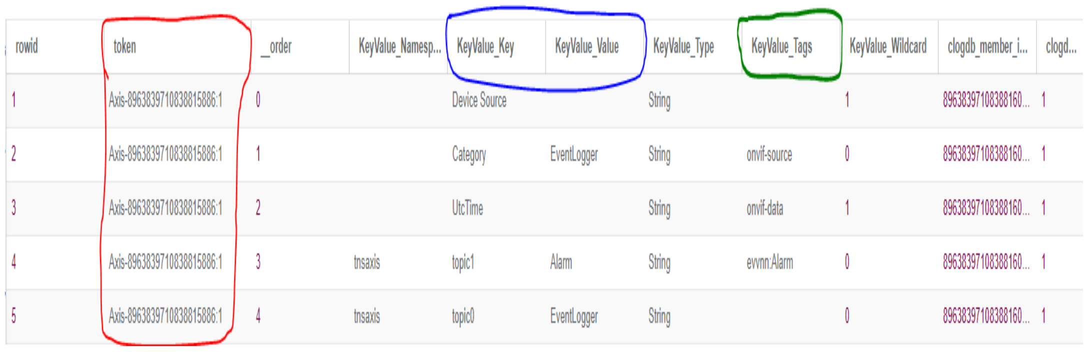
$$tag1; tag2; ...; tagN$$

For example in figure 2.9 we can see on row 4 the tag `evvnn:Alarm`, the evvnn stands for event value nice name.



rowid	token	UUID	DatabaseID	Property	__sizeKeyValues	Tags	clogdb_member_instance_id	clogdb_sequence_id
1	Axis-89638397108388158...	5581ad80-95b0-11e0-b883-accc8e9f863c	1	0	5		8963839710838816000	1

Figure 2.8: Table of how the event declarations are stored on a device



rowid	token	_order	KeyValue_Namesp...	KeyValue_Key	KeyValue_Value	KeyValue_Type	KeyValue_Tags	KeyValue_Wildcard	clogdb_member_i...	clogd...
1	Axis-8963839710838815886:1	0		Device Source		String		1	89638397108388160...	1
2	Axis-8963839710838815886:1	1		Category	EventLogger	String	onvif-source	0	89638397108388160...	1
3	Axis-8963839710838815886:1	2		UtcTime		String	onvif-data	1	89638397108388160...	1
4	Axis-8963839710838815886:1	3	tnsaxis	topic1	Alarm	String	evvnn:Alarm	0	89638397108388160...	1
5	Axis-8963839710838815886:1	4	tnsaxis	topic0	EventLogger	String		0	89638397108388160...	1

Figure 2.9: Table of how the event key and key values (event properties) are stored

2.4.3 VAPIX

VAPIX is the name for AXIS own REST-API. This API is used for many different things including getting lists of and setting parameter values on devices, but also to activate different features. The API is easy to use and gives the users control over the device, but the most important part is that it makes it easy to write software that can modify devices. An example use of this API was shown in section 2.5

VAPIX is an essential way that VMS software communicates with different devices.

2.4.4 On-Screen Controls

On-Screen Controls or OSC is the name used by AXIS to describe their way of implementing button based features on their VMS generically without having to make changes to existing code for each new feature. This section will explain how it works and what limitations exist preventing it from being an ideal solution to this thesis problem.

The main idea behind OSC is to use the HTTP/HTTPS post method[15]. The user query by posting JSON formatted data and will get a response containing data relevant to the query. The following is a simple example used to get the supported languages on the device. Listing 4 displays the JSON structure a VMS could send a device, the critical part here is the *method* parameter specifying what type of data should be returned, this is followed by *params* used to send additional data to the method *languages*.

```json
{
  "apiVersion": "1.0",
  "context": "",
  "method": "languages",
  "params": {
  }
}
```

Listing 4: JSON data used to request languages

When a device receives this request, it will return the data seen in Listing 5. It is similar to the request but contains the *data* field, and the different languages have now been obtained.

```json
{
  "apiVersion": "1.0",
  "context": "",
  "method": "languages",
  "data": {
    "languages": [
      {
        "language": "English",
        "locale": "en-us"
      },
      {
        "language": "Svenska",
        "locale": "sv-se"
      },
      {
        "language": "Deutsch",
        "locale": "de-de"
      }
    ]
  }
}
```

Listing 5: JSON response from the device with languages

Similar to this other methods exists used to list all available features on the device. The responses data field would then look more like Listing 6.

Here a response containing a feature called "SpeedDry" was received. The VMS will parse this response and get all necessary information needed to use the feature. Visually the response includes the nice name and a short description, and both are human readable

```
"data": {
    "en-us": {
      "name": "SpeedDry",
      "niceName": "Speed Dry",
      "info": "Remove water from the camera dome.",
      "requestType": "GET",
      "response": false,
      "requestURL":
        "/axis-cgi/com/ptz.cgi?auxiliary=speeddry"
    }]
    "features": []
},
```

Listing 6: JSON response from the device with features

and ready to be displayed to the user. The VMS will also obtain everything it needs to know to use the feature, included are a VAPIX URL used to activate the feature as described in section 2.4.3.

This approach works just fine for button-based features because they are few and usually do not change that often. However, OSC in its current state is unfit for taking care of event generalisation, for each button based features there could exist, perhaps, ten events, each needing their nice name and values. The values that make up the event are often also not human readable and could change during run-time adding another layer of complexity that OSC does not support.

2.5 RESTful web services

The VMS needs to be able to communicate with the different devices in order to get events. One way of doing this is to use RESTful web services, which is a way for machines to interact through the internet. It uses HTTP request such as GET and POST to provide users with functionality. GET is most often used to retrieve data, and POST is usually used to manipulate data on the server, such as updating lists or modify variable values. These kinds of web services are often used as an API where computer programs can send HTTP request to a server hosting a web service, and the server responds with data relevant to the request. Each functionality are their own URL so to access a specific feature; the request has to be sent to the right URL.

RESTful APIs are *stateless* meaning that neither the client or the server needs to know anything about the other[11]. The request contains all data needed for the server to produce an appropriate response, and the response contains everything needed to be useful. In short, this means that when a client sends a request, the server responds to it and then forgets about it, no session information is stored.

AXIS has a RESTful API of their own called VAPIX, an example use of this API is shown in listing 7. Here a POST request is sent to set a variable called *audio* to 1 thus enabling audio on the device.

```
http://IP-to-device/axis-cgi/audio/receive.cgi?&audio=1
```

Listing 7: REST API example to enable sound on an axis device

2.6 XML and JSON

To have the VMS display the events in a log a way of sending event-data from the device to the VMS is needed. The two most common ways of representing data are XML and JSON. Both are human readable and easy for machines to parse and generate. They are popular ways of sending data between applications because they are platform independent, and support exists in most programming languages. JSON is the more popular option as it is more readable and more lightweight(requiring fewer characters, compare listing 2.10a and listing 2.10b which is shows an example of how a person could be represented.)

```
{
  <person>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
    <age>24</age>
  </person>
}
```

(a) XML representation

```
{
  "firstName": "John",
  "lastName": "Doe",
  "age": 24,
}
```

(b) JSON representation

Figure 2.10: (a)XML and (b)JSON representation of a person

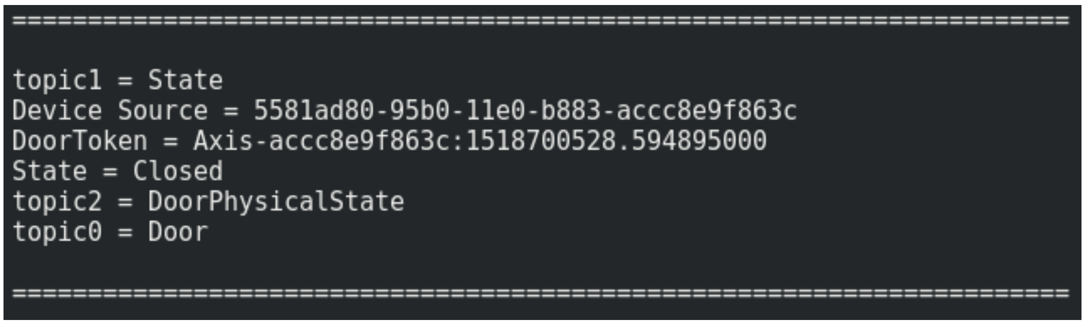
Chapter 3
Problem identification methods

Before improvements to the event system could be made, the problems with it needs to found, and this chapter will describe the methods used when identifying the problems. The first thing done was to get a better understanding how events works on a device, and to get this an investigation was done on an A1001 device. This investigation revealed some problems and secondly to identify further problems, feedback from experienced engineers was requested.

3.1 Investigation of events on the device

To get a better understanding of the events, a custom daemon was developed and run on an A1001 device. The purpose of this daemon was to use the event2 structure and subscribe to all events on the device, each time an event occurred the daemon printed all the data about the event so it could be observed and studied further. Each event contains several keys, and each key has a value. The daemon printed the event by first printing the key followed by " = " and then the value. The functionality to extract data from the events was provided in the event2 library. This daemon was used throughout the project as a base to test the different theories and solutions.

Figure 3.1 shows the raw data contained by the event produced when a door is closed. The event contains three "topic" keys as required by the ONVIF standard. The "Device Source" key is to identify from which the event came from, and the "DoorToken" key tells what door the event refers to. "State" is perhaps the most interesting one as this is the key that actually tells us the door has been closed.

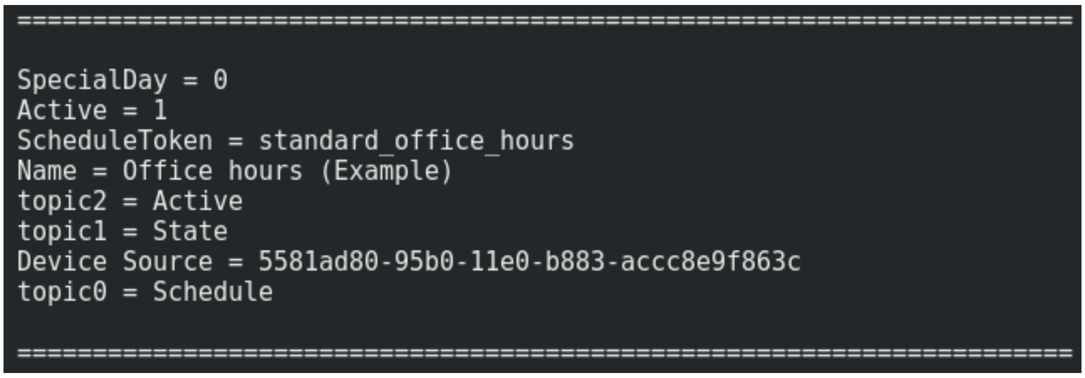


```
======================================================================

topic1 = State
Device Source = 5581ad80-95b0-11e0-b883-accc8e9f863c
DoorToken = Axis-accc8e9f863c:1518700528.594895000
State = Closed
topic2 = DoorPhysicalState
topic0 = Door

======================================================================
```

Figure 3.1: The old door closed event

This is a lot of useful information, the different topics make it easy to filter events and sort them into different categories and it is good to know what door the event refers to. However, for a human this is hard to read and it is not immediately obvious that a door has been closed.

Figure 3.2 highlights another problem with the events, some values are not human readable. This event is triggered at a set time and the two topmost data fields are SpecialDay = 0 and Active = 1. It could be displayed like this to a user but they would probably find it much more useful if the SpecialDay field said "No" or "New Years Eve" and Active could be "True" or "False"

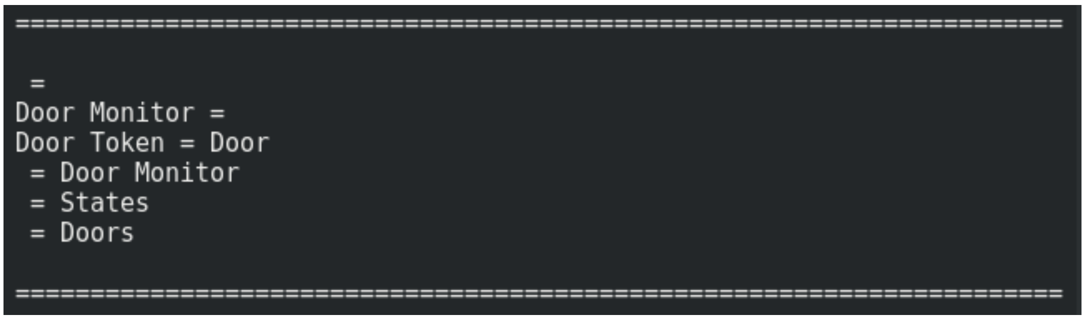


```
======================================================================

SpecialDay = 0
Active = 1
ScheduleToken = standard_office_hours
Name = Office hours (Example)
topic2 = Active
topic1 = State
Device Source = 5581ad80-95b0-11e0-b883-accc8e9f863c
topic0 = Schedule

======================================================================
```

Figure 3.2: The old scheduled day event

A string describing data in a human readable way is sometimes called a *nice name*, and this term will be used throughout this report. The values needs nice names to help the VMS side to display the values in a way useful to humans, and today the event2 system has support for this. It is possible to tag both keys and values with a nice name.

To investigate the potential of this nice name tag the daemon was modified and instead of the key and value, the corresponding nice names was printed instead. For the door closed event in fig 3.1 the corresponding nice names are shown in figure 3.3.



```
===============================================================

 =
Door Monitor =
Door Token = Door
 = Door Monitor
 = States
 = Doors

===============================================================
```

Figure 3.3: The old door closed events corresponding nice names

Some nice names are not that useful, and some keys and values do not have any nice names at all. If this nice name tag contained useful strings for all values it could be displayed by the VMS instead of the actual value, but this is not the case. Another problem is that the event2 system only allows the nice name tag to be set during the event declaration stage, making it hard to set useful nice names for values that change during run time, such as the special day key's value.

It would also be useful if the entire event had a collective nice name, as mentioned previously the door closed event could have the nice name "Door has been closed". This tag is only for the individual keys and values making up the event. But none of them can be used to represent the event in its entirety they can only be used by the VMS when displaying the event details. There exists no support for setting the nice name for an entire event.

3.2 Further problem identification

Each Friday at the department which this thesis was carried out, a seminar was held. This seminar is used to present what has happened during the week and to let the entire team sync up.

After the investigation of events on the device was done, the problems were presented to the entire team of about 50 engineers at one of these seminars. These engineers could then come with feedback as well as stating other problems previously not thought of, for example, one problem could be reduced readability in a log if events had too much irrelevant data attached to it, or the desire to aggregate some events together creating a larger more useful one and the need to filter events.

Chapter 4
Problems and Approach

The previous chapter described the methods used to identify problems. This thesis will focus on four of these problems, making the events readable to a human, enable filtering of the events, aggregation of events and the final problem is to make sure the solutions are easy to use so that companies will actually use them.

4.1 Human Readable Events

To display the events in a log, and for this to be useful, the events needs a human readable name. This name should let the end user immediately know what has happened for the event to have been triggered. This is done today by some VMS developers by having a hardcoded list with descriptive names for each available event on the device, and this list would then have to be updated each time a new event is added. To avoid this a way of describing the events automatically and generically is needed.

Another unwanted aspect of the events is the amount of data presented. A massive amount of information is not easily digested, and it takes time to read it all and understand what is going on. Many events from the device will include a lot of data that's not useful for a human and it will clutter the log with unwanted data and in turn reduce readability.

4.1.1 Approach - Event descriptions

As previously mentioned an event consists of different keys each with a value. Each key and value can have nice names on the device, but there exists no support for the collective event to have a nice name.

This is solved by having the events require a "NiceName" key with a nice name representing the entire event as value, as seen in listing 8. The VMS side will then replace the part within the curly brackets with the value of the Key with that name, in this case, the "State" key from the example in figure 3.1.

```
"Door has been {State}"
```

Listing 8: Example of a NiceName keys value

But this only solves half the problem as the value needs a nice name of their own, to avoid the 0 and 1 situation described previously. It would be good if the nice name tag of the values could be used. However, the event2 system has the limitation that nice names need to be set at the event declaration stage. After that, the names cannot be changed. And that eliminates the possibility to set nice names for values that will change during runtime.

One solution to the changing value problem is simply to extend the event2 system to include the ability to change nice names after the event has been declared. It makes it possible to have custom nice names for each of the values.

Another solution is to add another mandatory key to the events called "ValueMap" and the value should be similar to listing 9.

```
"Status=0:Opened;1:Closed;2:Forced Open"
```

Listing 9: Example value map string

This string will have the name of all applicable keys[1] followed by the possible values and a nice name for the value. This string will then be parsed, and a value map can be constructed and associate values with nice names just as shown in figure 4.1. When constructing the events nice name, the key's value can then be swapped for the values nice name from the value map, an example how this is done is illustrated in figure 4.2.

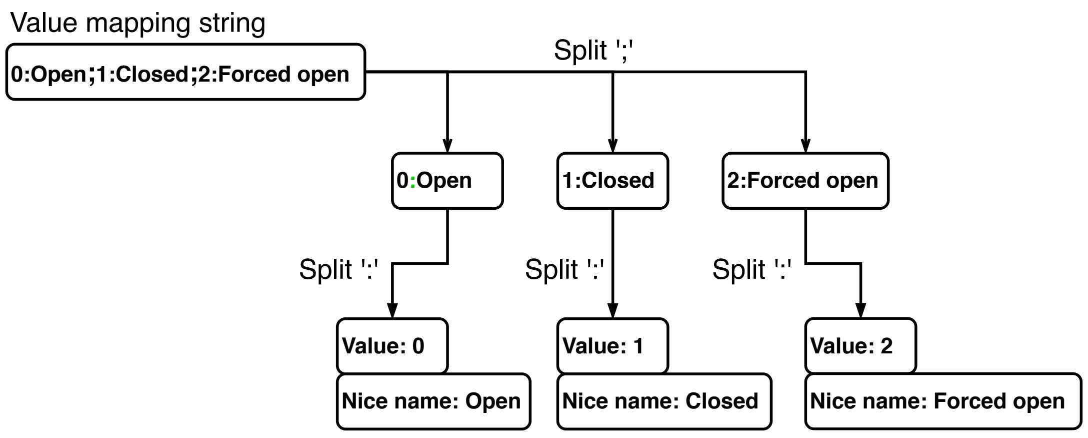


Figure 4.1: Value map parsing steps

[1]The one used by the NiceName key

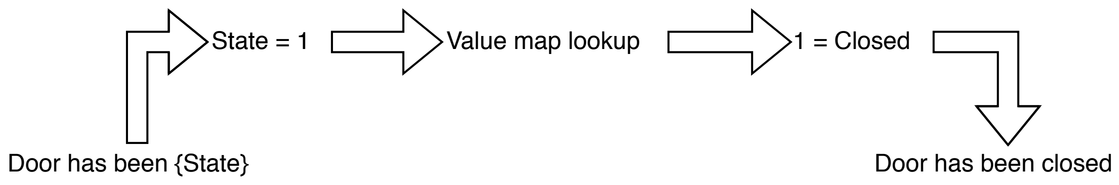


Figure 4.2: Nice name generation using a value map

4.1.2 Approach - Irrelevant data reduction

As the need and skill level of each end user is different, it's impossible to know which of the events keys is relevant to display. Two basic methods were however developed to try and make a good enough system for evaluating the events. One way is to send only the keys that have an explicitly set nice name for the value. Because nice names need to be set by hand in the code on the device by a human, it will probably be useful to some other human on the VMS side if that effort has been made. This method has the drawback that events with no nice name values will be skipped, a card number has no nice name because it's just a number. But a user may want to see it anyway.

The second way is to introduce another key to some events, called "Advanced", and the value includes the name of keys deemed not useful for the average user. All the events keys are sent to the VMS, but now the VMS will know what keys are considered advanced and can choose not to display those keys. But the VMS can have the option to display all keys if a user wants to. The only drawback to this is another key need to be attached to some events. And there is no way to tell if a key should be advanced or not, it's up to each developer.

4.2 Filtering

When it comes to filtering the events, the fact that every event needs to comply with ONVIF as discussed in section 2.2 can be used. What this means is that each event needs to implement one or more topics. These topics can then be used to classify what type of event it is and this enables a VMS to filter based on these topics.

There's nothing wrong with filtering in this way, but just as with the nice names with events, VMS developers has to hardcode lists with the different topic combination for each event. It would be better if the VMS could ask the device which events it has, and get a list of all the topic combinations for these. Another level of complexity to this problem is that filters also need to have nice names. If a filter is displayed to the user in the raw format as a combination of topics, it's not clear what the event is about. If instead the filter was presented as, for example, "Door - physical state" the user would immediately know what the filter did.

In short, each event needs a filter associated with it, but, in most cases, the filter nice name cannot be the same as the name displayed in a log. Figure 4.3 illustrates the difference. The difference between the nice names is that for a single event instance, the name changes based on the values of some keys, but the filter nice name is used to describe the general event, it is not dependent on values and thus will never change.

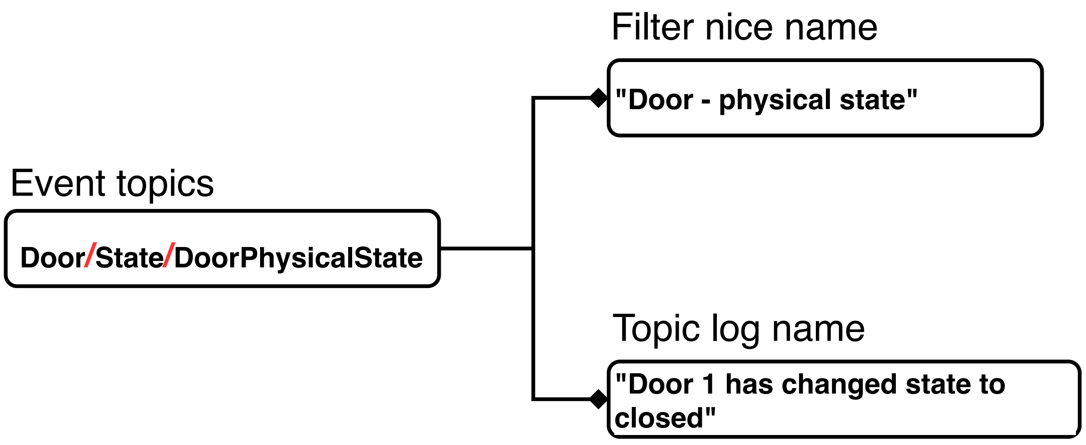


Figure 4.3: The different nice names needed

To have filters for each event is great, but it would be even more useful if the user could filter events based on values too. The physical state event used in previous examples has three different values, seen in figure 4.1. If a user wants to only view the times the door has been forced open, the filter for the specific event can be used. But this will also display every time the door has been opened and closed normally as well. Support for further filtering eliminates this problem.

If a VMS has a time-line view of a video feed or something similar, then perhaps they would want the events to show up on the time-line, or have the time-line jump to a specific point when selecting an event from the log, or simply wants to sort events based on a time-interval. For this to be possible the ability to filter events based on the time they are created would be required.

4.2.1 Approach - Filter and filter names

During the event declaration stage, all keys needs to be declared as well. This means that when the event has been declared it's topics are already defined. This makes it easy to have the device maintain a long list of all event topic combinations. Whenever the VMS user wants to apply a filter, a request is sent to the device which will respond with the said list.

Because filters and topics will not change during runtime, they can be saved and added to the list during the event declaration stage. The proposed solution is to modify the event2 system and add support for events to have the key *filterNiceName* with the filter nice name as value. When events are declared this value will be saved in a long list alongside the topics as seen in listing 10.

```
"Door/State/DoorPhysicalState: Door - physical state"
```

Listing 10: Event topics with a corresponding filter name

The topics are separated by a "/" character and a ":" character marks the beginning of the suggested name of the filter. All left for the VMS to do is to parse the string and associate the topic combination with the supplied name for easy filtering.

4.2.2 Approach - Filter on values and time stamps

The issue with filtering on values as well as the actual filter is a bit harder. The device needs to provide every possible value each key can have to the VMS for it to be possible. But today the device doesn't know what different values are possible, certainly not at runtime, when some values have never even appeared.

The value map solution discussed in section 4.1.1 could potentially also solve this problem. It suggests that all of the events possible values should be present in a value map string at the declaration stage. Because this string includes both the value and a nice name for that value, it could be used for filtering as well. If the value map string is concatenated to each filter in the list, the VMS could then parse it in conjunction with the filters and then know all the possible values, enabling further filtering. This also has the bonus of obtaining the nice name of the values, making it possible to display that part of the filtering directly without modifications.

Each event already contain a time-stamp of the time they are created. This time-stamp could be used as a filter if a user wants to display events that has happened at a certain time, or interval.

4.3 Aggregation of Events

To show what aggregation of events is and the problem it can solve, the best way is to begin this section with a user case, illustrated in figure 4.4. In the figure we have a person, "Person A" and a door leading into a building. The door is equipped with a access control system (not shown in the figure). The access control system is connected to a VMS, which the events, produced by the control system, are sent to.

Person A wants to access the building. To access the building, Person A needs to scan its key card on the scanner (the thing on the door). When Person A scans its card an event is sent to the VMS.

On the right side of the figure is a time line of every event sent and its origin (what caused the event). For a simple action "Person A access the building", the access controller produce seven events to describe this action. On the VMS log after receiving these events it could look messy, especially if events from different sources has been received during the duration of the action.

The user case is made up, however the A1001 produce approximately the same amount of events, as the access controller did. It is therefore not far from what the reality looks like and that the problem do exists.

Figure 4.5 illustrate a example of this, the log is not from an actually VMS, but reflects the reality that can occur in a real log. In the log two actions has been done, Person A access the building and another person has also accessed the building simultaneously using another door to the building, connected to the same VMS. In the figure, events coming from Person A:s action is marked for readability for the reader.

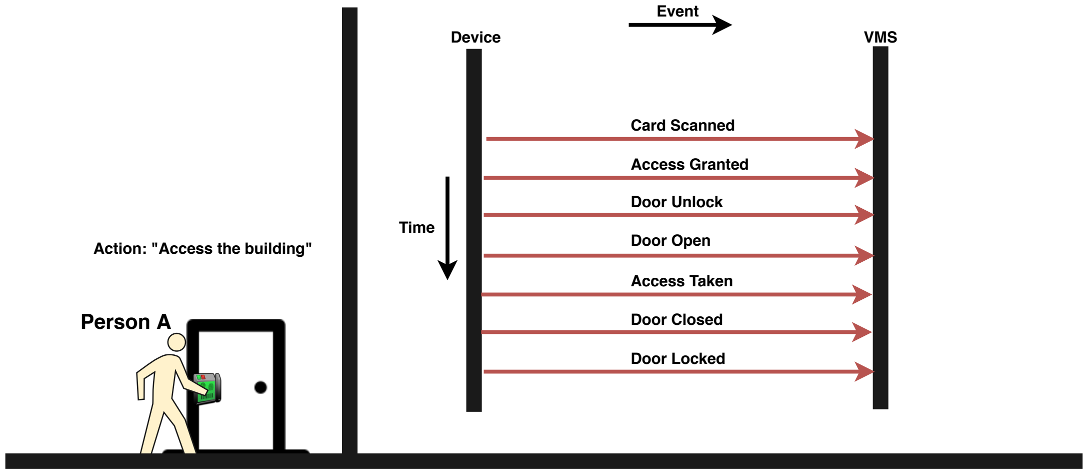
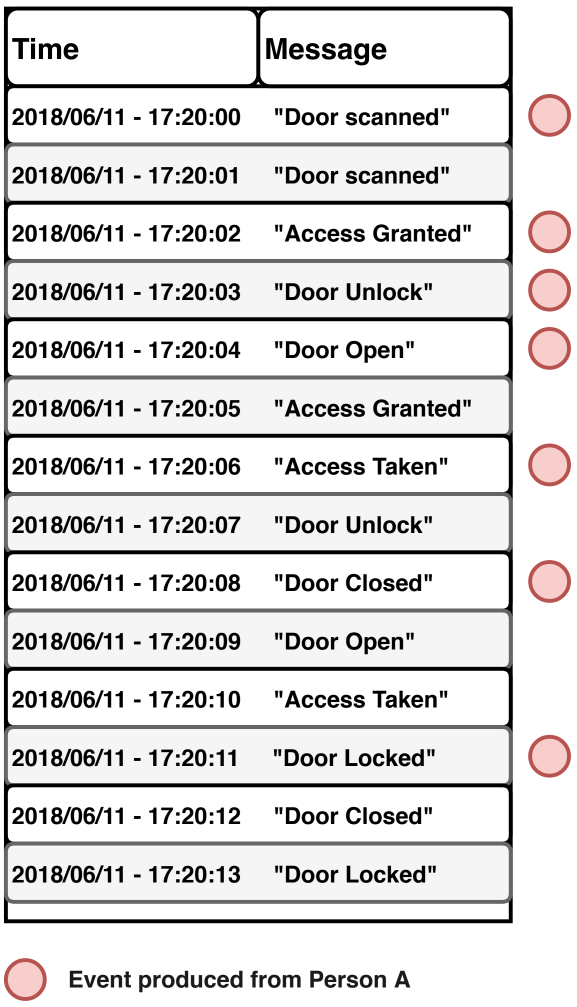


Figure 4.4: Illustrated example of a Person accessing the building, resulting in a chain of events sending to the VMS.



Time	Message	
2018/06/11 - 17:20:00	"Door scanned"	●
2018/06/11 - 17:20:01	"Door scanned"	
2018/06/11 - 17:20:02	"Access Granted"	●
2018/06/11 - 17:20:03	"Door Unlock"	●
2018/06/11 - 17:20:04	"Door Open"	●
2018/06/11 - 17:20:05	"Access Granted"	
2018/06/11 - 17:20:06	"Access Taken"	●
2018/06/11 - 17:20:07	"Door Unlock"	
2018/06/11 - 17:20:08	"Door Closed"	●
2018/06/11 - 17:20:09	"Door Open"	
2018/06/11 - 17:20:10	"Access Taken"	
2018/06/11 - 17:20:11	"Door Locked"	●
2018/06/11 - 17:20:12	"Door Closed"	
2018/06/11 - 17:20:13	"Door Locked"	

● Event produced from Person A

Figure 4.5: Illustrated example of how a log could look like after receiving the events from simultaneous actions, that has not been aggregated.

It would be a tedious task for the end-user to find and analyze the events linked to Person A:s action mixed up with similar events. It would be nicer if the events were aggregated together into a event that summarized the whole action, instead that the action would be compiled by seven events.

The idea is when an action is done, an aggregated event is sent from the device (see figure 4.6) and the VMS log could display this aggregated event as illustrated in figure 4.7, where the aggregated event consists of the events produced by the action.

The level of abstraction is raised with aggregated events, hide the smaller events that really does not describe much for the end-user. Making it easier to identify actions that has occurred without piece together the smaller events.

This problem raises the question on how would aggregated events work in Axis event system? What changes needs to be done? Can this be done dynamically or should it be done manually?

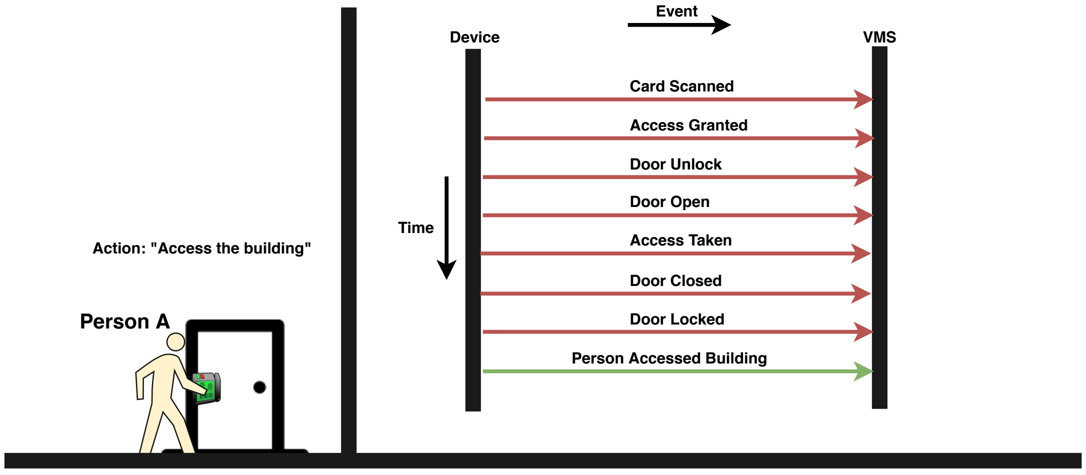
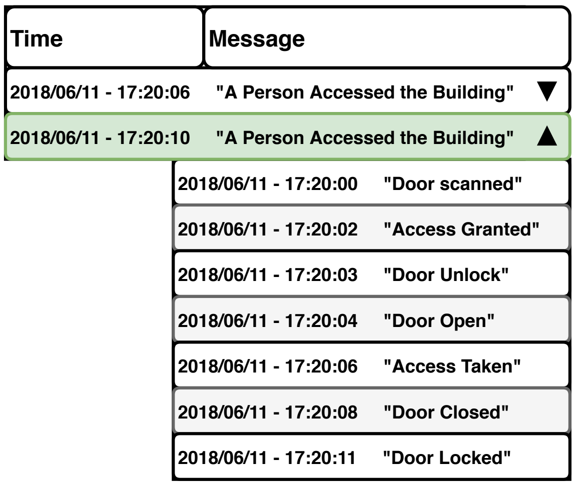


Figure 4.6: Illustrated example of a Person accessing the building, resulting in a chain of events sending to the VMS and an aggregated event that summarized the chain of events.



Time	Message
2018/06/11 - 17:20:06	"A Person Accessed the Building" ▼
2018/06/11 - 17:20:10	"A Person Accessed the Building" ▲
2018/06/11 - 17:20:00	"Door scanned"
2018/06/11 - 17:20:02	"Access Granted"
2018/06/11 - 17:20:03	"Door Unlock"
2018/06/11 - 17:20:04	"Door Open"
2018/06/11 - 17:20:06	"Access Taken"
2018/06/11 - 17:20:08	"Door Closed"
2018/06/11 - 17:20:11	"Door Locked"

Figure 4.7: Illustrated example of how a log could look like after receiving the events from simultaneous actions, that has been aggregated.

4.3.1 Approach

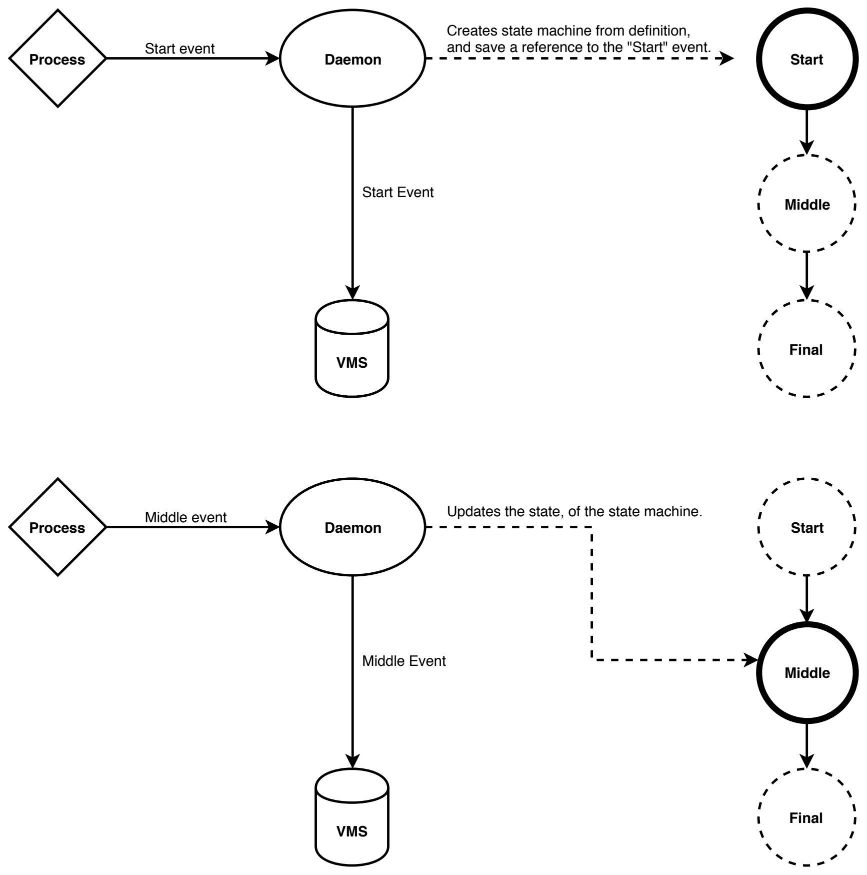
The approach was to gradually developed a concept that could be further developed in the future into a proof of concept. The lack of time was the reason for this and also the overall complexity of the problem made it difficult to cover and find all the use-cases. Three questions was the start of the research, where, when and how should the event be aggregated. These question was a good starting point and building block for the concept, however not a final result.

Where the aggregated should be defined must be on the device. It is logical because all of the other events created by the device are defined and created on the device. What exactly the aggregated event should contain is flexible, the key things however should be a list of reference to the events so that the VMS would know which events will be linked together to the aggregated event and a descriptive name explaining the action.

The definition of the aggregated event should be saved on the device, such as the event definition are now on the devices. A separate process on the device will be needed for creating the aggregated events. It will need to listen to every event produced by the device and check with a list of aggregation definitions if the event is a part of a aggregation. To be able to know when to start creating an aggregated event and when to send it, it assumes that events come in a sequential order with a start event and a final event. The event will still be sent to the VMS without delay, and after the final event will the aggregated event be sent.

For example, the main process will need to listen to all start events of a possible aggregation. When a start event has occurred a new task or process should start only listening to the next event of the aggregated event list, and when the final event has occurred, send the aggregated event. To terminate an aggregated event construction a deadline needs to be set. If an aggregated event construction is ended only when the final event has occurred it is possible that the construction will go on forever for the case when the final event never comes. A deadline is therefore needed, when the deadline has come the construction process of the aggregated event will be terminated. From the previous example with Person A accessing the building, the start event would be the event produced by scanning the card, and the final event when the event produced when the door is locked. A deadline could be 1 to 2 minutes after the start event has occurred before terminating the construction process, confirms that the action will not happen.

Figures 4.8 illustrates a example of how the aggregated events could be created on the device. Every device has internal processes, example from applications running on the device, these processes generate events. With the new daemon constructed in this thesis, it subscribes to all external events happening on the device and send them to the VMS. This is necessary because all events needs to go in one direction. This daemon will check every events and see if it is a start event for an aggregation. If that so a reference to the event is saved, and a aggregation process is started. From the aggregation definition a state machine is created, needed for keep track of the next event in the aggregation. When the next event in the aggregation has occurred the Daemon updates the state machine and saves the reference to the event. When the final event has occurred, the aggregated event is created with the references and a descriptive name from the definition and is sent to the VMS.


Process
Start event
Daemon
Creates state machine from definition, and save a reference to the "Start" event.
Start
Start Event
Middle
VMS
Final
Process
Middle event
Daemon
Updates the state, of the state machine.
Start
Middle Event
Middle
VMS
Final

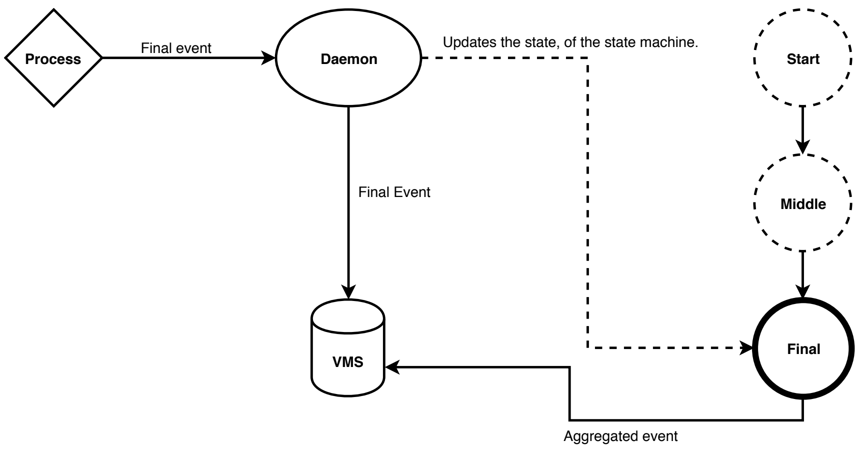


Figure 4.8: 1: State machine created by the daemon when start event of an aggregation is received, 2: State machine updated when next event in the aggregation is received, 3:When the last state is set, the aggregation is sent to the VMS

The state machine will be on a another process, and is created for every new start event that occurs, so it will be possible that several state machines are alive simultaneously. When the deadline has occurred or when the final event has occurred that process will be terminated. However it will be necessary to make choices depending on when an aggregated event is to be sent if that aggregated event is a sub part of another aggregated event definition. Figure 4.9 illustrate an example of this.

We have two aggregated event definitions. The first one is an aggregated event about a person not taking access to a building (A1), the second one is the aggregated event of a person accessing the building (A2). As seen in the figure, A1 and A2 both consist of the same start event and the three succeeding events. When the start event ”Card Scanned” two state machines are created (not shown in the figure). Lets say that the action is ”person access the building”, when the four first events has occurred A1 will be created and sent to the VMS. However when the three last events has occurred A2 is also created and sent to the VMS. Now the VMS has received two aggregated events and they describe two different actions, and the actions also contradict each other. A person that did not go into the building, but did go into the building.

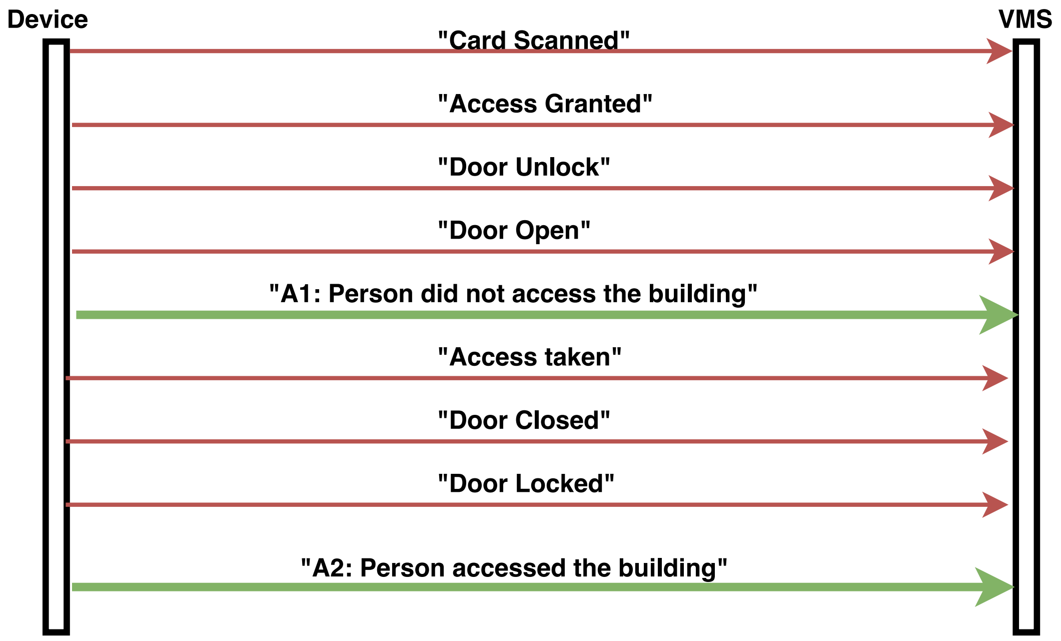


Figure 4.9: Illustrated example of when a aggregation definition is a sub part of another aggregation definition.

First thing to solve is not to have two different state machines alive when there exists several definitions with the same start event. One solution is to merge the state machines together into one. For our example we would have a state machine looking something like this (see figure 4.10). However which of the aggregated event should be sent? Should it be a shortest match and send A1, or longer match and send A2 instead. If shortest match is applied it would still create incorrect information. When the first four events has occurred, A1 is sent and the process is terminated. However the last four events do occurred but A2 is never sent. The VMS will have information of a person not accessing the building but in reality the person did accessed the building. This is illustrated in figure 4.11. This results in that longest match should be applied in such situations.

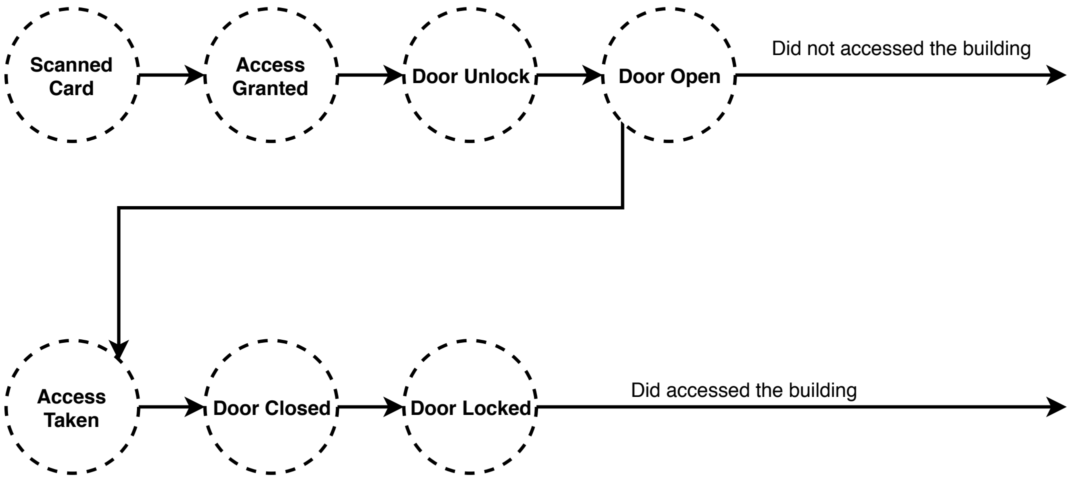


Figure 4.10: When two or more aggregation definition has the same start event, it would be possible to merge the state machines.

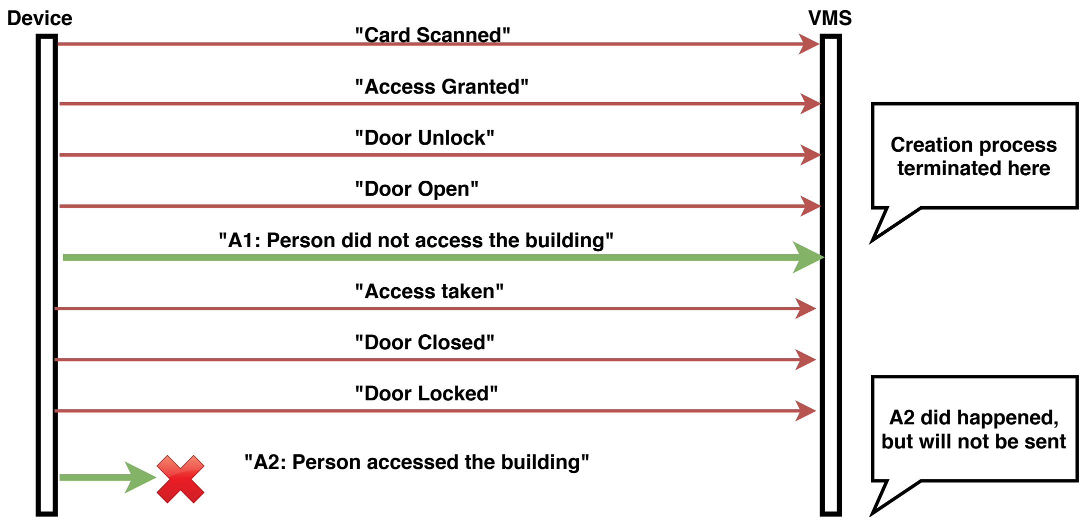

Device
VMS
"Card Scanned"
"Access Granted"
"Door Unlock"
"Door Open"
"A1: Person did not access the building"
"Access taken"
"Door Closed"
"Door Locked"
"A2: Person accessed the building"
Creation process terminated here
A2 did happened, but will not be sent

Figure 4.11: When shortest match is applied the termination will end to early because of A1, resulting that A2 is never sent.

There exists still a situation that may occur with longest match. In figure 4.12 we have two fictional actions A1 and A2. A1 is sub part of A2, however the deadline differs, A1 has a deadline of five seconds and A2 a deadline of one day. The two first events has occurred, because of longest match A1 will not be sent, it must be verified if A2 should be sent. Next event occurred and there exists still a chance that the final event will occur. If the final event will not occur, the state machine will terminate sending A1 with a long delay.

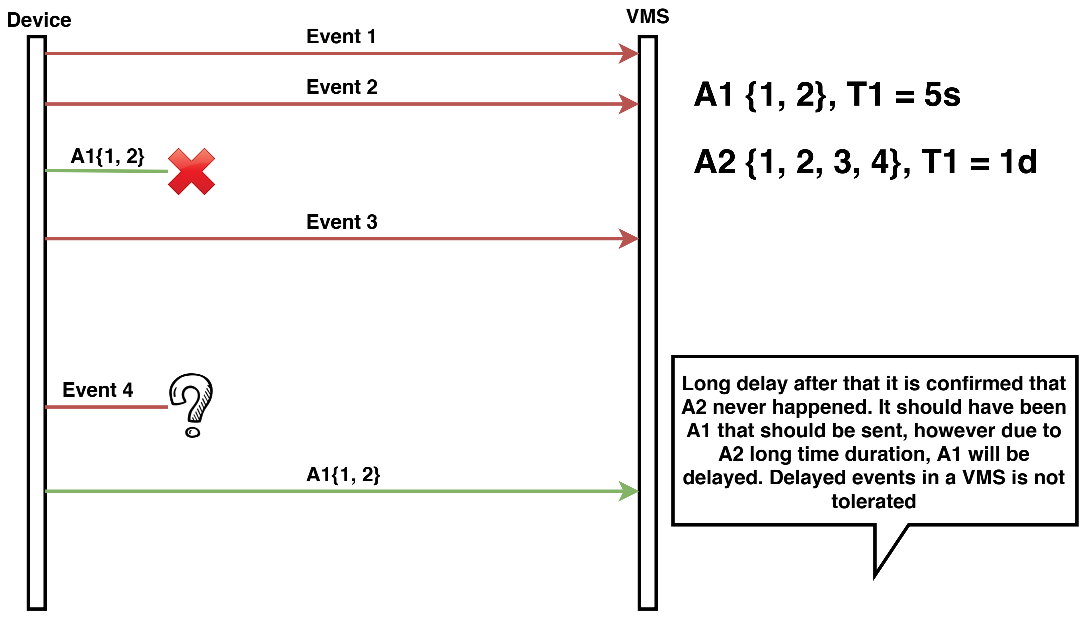

Device
VMS
Event 1
Event 2
A1{1, 2}
Event 3
Event 4
A1{1, 2}
A1 {1, 2}, T1 = 5s
A2 {1, 2, 3, 4}, T1 = 1d
Long delay after that it is confirmed that A2 never happened. It should have been A1 that should be sent, however due to A2 long time duration, A1 will be delayed. Delayed events in a VMS is not tolerated

Figure 4.12: A problem that may occur if longest match is applied, delaying A1 before knowing if A2 should be sent.

4.4 Ease of Use

The final problem is that the system needs to be easy to use and implement. If a VMS company wants to use the new event system to create an event log in their software, but the implementation time is too high, no VMS company will want to do it, and the whole new event system is useless.

There are problems that exists on the device side too, mainly that the way that nice names, filters and value maps need to be formatted is now very complex. This is not only hard for the developers to use, but requiring specific structures is also very prone to human errors. If the new event structure is too complex and hard to use, it could increase the development times and in turn could increase the time to market which will potentially hurt the sales numbers for the company(Axis)[12].

4.4.1 Approach - Device development

To ease development on the device, programming functions needs to be added that will perform the formatting of nice names, filters and value maps automatically, making the developers life easier as well as preventing any potential human error. For example, the value map string could be generated by calling a EVENT_GENERATE_VALUEMAP_STRING function with the in parameters being a key name followed by possible values and their nice names. For the example value map in listing 9 it would look like listing 11.

```
event_generate_valuemap_string("Status",
                    0, "Open",
                    1, "Closed",
                    2, "Forced Open");
```

Listing 11: Example use of the GenerateValueMap function

4.4.2 Approach - VMS development

To ease the development for VMS developers, an API will be provided. This API will take care of all the heavy lifting concerning event handling, such as parsing and filtering, see figure 4.13. This will lead to faster implementation times for VMS developers as they don't have to worry about any event specific code, allowing the developers to simply call a function and then only have to worry about applying their own styling to the event data provided.

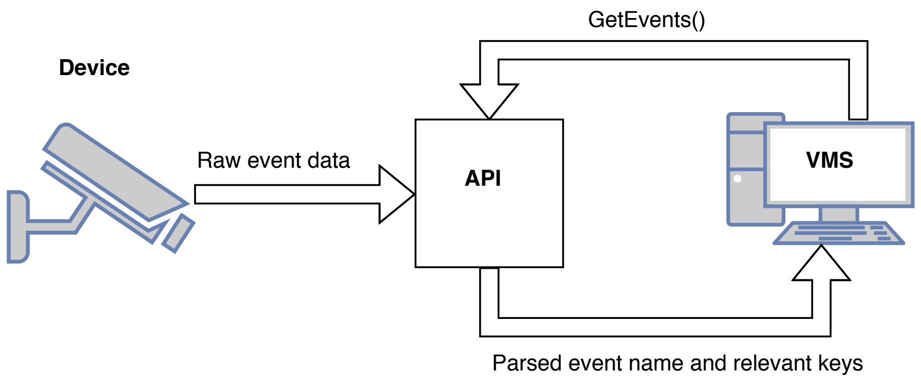


Figure 4.13: Event API

There are two ways this API could be made, the first way is to develop a C# API and the second is to create a REST API.

Chapter 5

Prototyping

To enable testing and evaluation of the proposed solutions a prototype system was made. The prototype consisted of three parts, seen in figure 5.1. One part of the system was modifications to the embedded system on an A1001 device, most of which is isolated to a custom daemon running on the device. This daemon had the purpose to collect and send events to the VMS. The second was a VMS side API that receives events and is responsible for parsing and extracting useful data from them. Lastly, a mock GUI was made that implements this API to ensure functionality and to verify everything was working as intended.

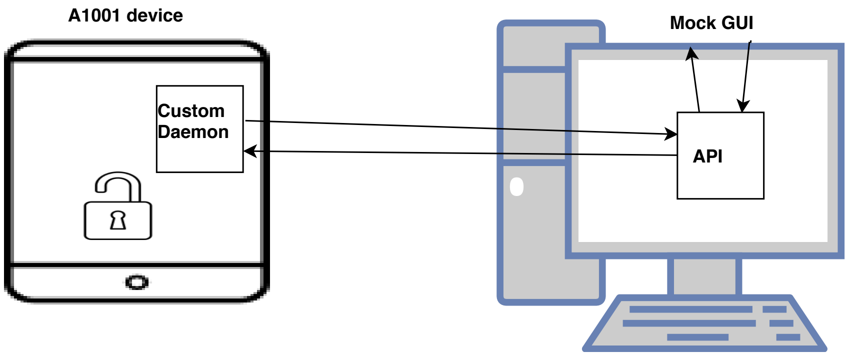


Figure 5.1: The prototype system

This chapter will go through how each part of the prototype was constructed, and motivate some design choices made.

5.1 Device Software Extensions

The prototype development on the device can be split into two parts. The most significant part is the construction of a custom daemon. Smaller modifications to the event2 system were also required and will be explained as well.

5.1.1 Daemon

The custom daemon used as a part of the prototype system was built on the same one used when investigating how the events were structured on the device. This meant that the code for subscribing and receiving events was reused. However, additional features were needed.

The existing code base for the embedded system is very large, and to avoid having to make changes to it as much logic as possible was isolated to this daemon. This meant that some basic functionality had to be redone by the daemon, such as allowing a VMS to connect to it.

To enable communication with a VMS, code were written to allow a VMS to connect to the device via a TCP-socket, and this connection was used when transferring data. The choice to used TCP-sockets was made based on the fact that there already existed support for it in C. So using this would reduce development times.

When an event happened on the device the daemon looped through all key-value pairs, and concatenated them together in a large string, separating each pair with the ";" character to enable easy parsing, see listing 12.

```
"key1=value1;key2=value2;key3=value3;timestamp"
```

Listing 12: How events are structured before being sent to a VMS

Instead of this custom data format, XML or JSON structures could be used instead see listing 5.2, this was also tested.

```
{
  <event>
    <key1>value1</key1>
    <key2>value2</key2>
    <key3>value3</key3>
    <timestamp>timestamp</timestamp>
  </event>
}
```

(a) XML representation

```
{
  "key1": "value1",
  "key2": "value2",
  "key3": "value3",
  "timestamp": timestamp,
}
```

(b) JSON representation

Figure 5.2: (a)XML and (b)JSON representation of an event

When all key-value pairs had been processed a time stamp was added, and the resulting string were sent to the VMS through the TCP-socket.

One problem with this method is that during the time it takes to send the event through the socket new events could have been triggered. To combat the real-time programming problem that the new events are editing the string currently being sent a *mutex* was used. This mutex protects the string from being edited by other threads[5].

To enable clients to retrieve the list with filters the VMS could connect to another port. The daemon listens to the connection, and if a client connect, the daemon then sends the list with filters and filter names, and then terminates the connection. This system could potentially be extendable in the future, and the device could receive data from this connection as well and send other responses, e.g., if a '2' is received only the filter names could be sent instead of the whole filter.

The daemon also used the publisher part of the event2 system to create events on its own. The daemon will generate an event each time a client connects to it. This event was made to be able to have total control of it, from the declaration stage to sending it to the VMS without modifying too much of the device's software, thus isolating as much as possible of our changes to this daemon. These events were used as a base when testing different event structures, such as adding additional keys to events.

5.1.2 Event2

The event2 system needed to be modified to be able to support some solutions. The edits were however small and simple. The following functions were added

1. **event_set_value_nice_names** - Allows the developer to edit nice name of a value after the declaration stage.
2. **events_get_filter_list** - Retrieves all events filters from a list on the device.
3. **event_generate_value_map_string** - Generates a value map string for a provided key and values/nice names.

The first function only edits the nice name property of the event, and will be used to set values nice names dynamically during run time, to help the VMS display it better to end users.

The second function returns a list of all filters along with corresponding nice names. This list is a local string and is updated each time an event is declared. When the event is declared, the event's topics along with the new *filterNiceName* key is added to the end of the string.

The third function was previously described in section 4.4.1.

5.2 API

To aid with the development surrounding events and to hopefully enable faster development times for VMS vendors, an API was developed. This API will take care of everything event related and provide ready to use data to the VMS, such as parsed nice names. It makes it easier for VMS developers as they don't need to understand the event system and are able to only use the API functions that they need.

One challenge to consider for the API is to design it in a good way and make it easy to use and possibly extend. Because when an API is distributed, you can only add functionality, never remove as that would displease a lot of customers. Because of this, it is good practice to only include the most basic required functionality to avoid later regrets[6]. To ensure a good design and that only functionality that's really needed were added, ideas from Test Driven Development(TDD) was used. The idea behind TDD is to write test cases before the actual code, and this forces the developer to think everything through and avoid unnecessary "just in case" code and get a better understanding of the requirements, thus achieving better code[4].

Using this idea as a starting point pseudo code was written on paper by the authors of this thesis. The objective of the code was to implement a log from a VMS developers perspective. This code took the role of the test cases in TDD and allowed the authors to get an understanding of what features should be included in the API. The most desired features were the following.

1. ConnectToDevice - Connects to a deivce.
2. GetAllEvents - Retrieves all events received from the device.
3. GetEvent - Get a single event.
4. GetFilterList - Retrieves a list of available filters.
5. ApplyFilter - Applies a filter to only get certain events.

This information was used as a base when considering what functionally to actually include in the API and what is appropriate to have the VMS developers implement themselves, such as apply their own styling when presenting the data.

5.2.1 C# API

The API made was done in C# and provided as a separate *dll*[3] file that other C# projects can use by just importing it. This choice was made on the basis that ACS is developed in C# and implementation into this software is desired for the future. The API was thus powered by *.net*[8] which proved a valuable tool because it provided a lot of functionality that helped with parsing and network sockets.

Communication with the custom daemon on the device was done through TCP sockets. Two different connections were made possible, the first one only connects and listens to the event stream and nothing more. The second one has a two-way communication and is used to retrieve the list with filters and their nice names.

When an event is received the API will construct the events nice name. There are two ways to do this, as desribed in section 4.1.1. Both of these approaches, value map and changing the event2 system, was implemented and tested.

Filtering was done as described in section 4.2.2. The API connected to the device and recieved a list of filters. The list was parsed and when a filter was applied, only events with a topic combination matching the filters, were provided when a VMS requested events.

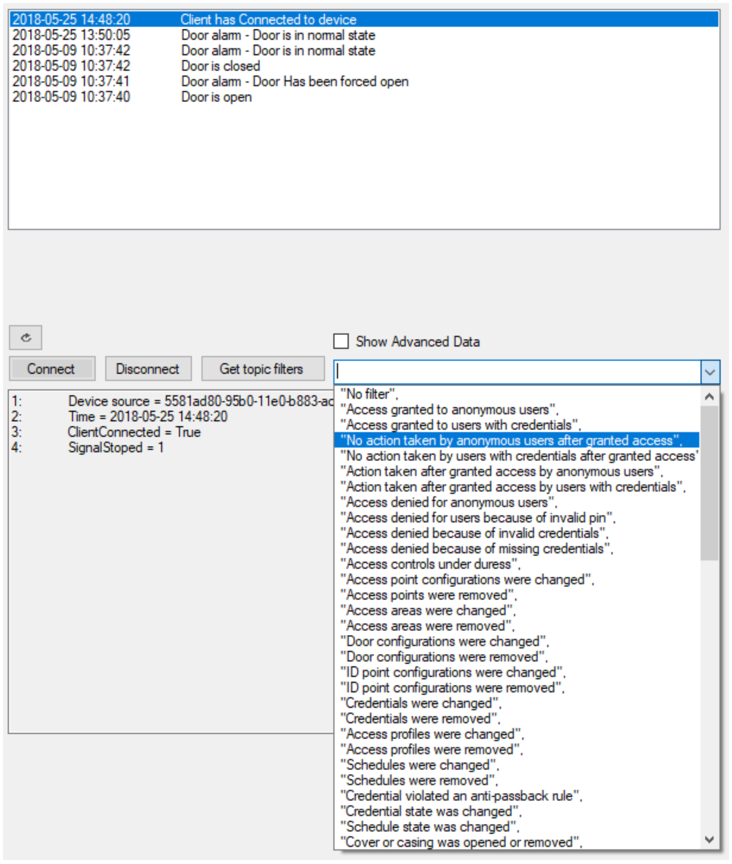
When an event has been received and parsed, it will be saved in a local SQL database. The API takes care of everything from saving to the database and retrieving the events again. When a filter is applied, only the events with the correct topics are retrieved, and if a value filter is also used, the events are further filtered by the API before being displayed.

5.2.2 REST API

An alternative to using a native C# API is to use a REST API, just as OSC does. The device already implements VAPIX, and the idea would be to extend this API to support the new event system. For this thesis no REST API was made, but the pros and cons of a REST API is discussed in chapter 7.

5.3 Mock UI

An event log is the perfect way to test that the solutions were working, and was able to present events in a way useful to humans. However, it was deemed that it was too much work not related to this thesis to implement this log in ACS. Instead a mock GUI was made that implemented the C# API and used this to create an event log. The resulting event log is displayed in figure 5.3



```
2018-05-25 14:48:20    Client has Connected to device
2018-05-25 13:50:05    Door alarm - Door is in normal state
2018-05-09 10:37:42    Door alarm - Door is in normal state
2018-05-09 10:37:42    Door is closed
2018-05-09 10:37:41    Door alarm - Door Has been forced open
2018-05-09 10:37:40    Door is open
```

```
1:    Device source = 5581ad80-95b0-11e0-b883-ac
2:    Time = 2018-05-25 14:48:20
3:    ClientConnected = True
4:    SignalStoped = 1
```

```
"No filter",
"Access granted to anonymous users",
"Access granted to users with credentials",
"No action taken by anonymous users after granted access",
"No action taken by users with credentials after granted access'
"Action taken after granted access by anonymous users",
"Action taken after granted access by users with credentials",
"Access denied for anonymous users",
"Access denied for users because of invalid pin",
"Access denied because of invalid credentials",
"Access denied because of missing credentials",
"Access controls under duress",
"Access point configurations were changed",
"Access points were removed",
"Access areas were changed",
"Access areas were removed",
"Door configurations were changed",
"Door configurations were removed",
"ID point configurations were changed",
"ID point configurations were removed",
"Credentials were changed",
"Credentials were removed",
"Access profiles were changed",
"Access profiles were removed",
"Schedules were changed",
"Schedules were removed",
"Credential violated an anti-passback rule",
"Credential state was changed",
"Schedule state was changed",
"Cover or casing was opened or removed",
```

Figure 5.3: Top pane: List of events presented in a human-readable format. **Lower pane:** List of all the events keys and their values.

The GUI has three buttons. One to connect to the device, the device IP is hardcoded as only one device was running the custom daemon necessary for the GUI to work, but it can easily be extended to allow the user to enter their own IP address. Once connected another button is used to disconnect and close the connection. Lastly, there is a "Get topic filter" button that uses the API to fetch all available filters from the device.

A dropdown list is also present, and when the user clicks on this, the GUI will use the API to fetch all available filters from the device just as with the "Get topic filter"-button, these are then shown in the list. When a filter is selected the log is cleared, and the API will fetch all events with that filter's topic combination from the API database, only these events are then shown in the log. The get filter button was used to test functionality during development of the GUI, and is not actually necessary as clicking on this dropdown list does the exact same thing.

There GUI also has two textboxes. When connected to a device, each time an event is triggered on the device it will be displayed in the upper one. It is presented in this textbox first with the time at which it triggered and then the parsed nice name.

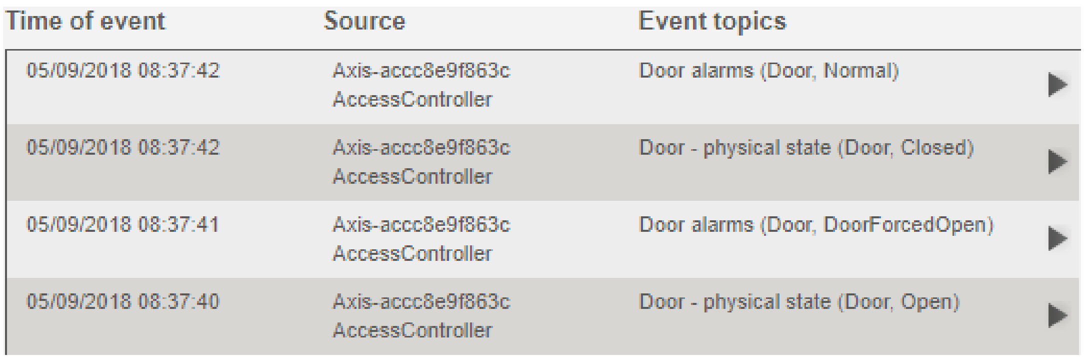
The user can click on an event to select it. Once selected the event's keys along with the key values, is shown in the lower textbox. Keys that had been deemed advanced are not shown unless the "Show Advanced Data"-box has been ticked.

Chapter 6
Evaluation and Results

This chapter will go into detail about how the different parts of the project was evaluated. The results from said evaluations will also be presented but will be discussed further in chapter 7.

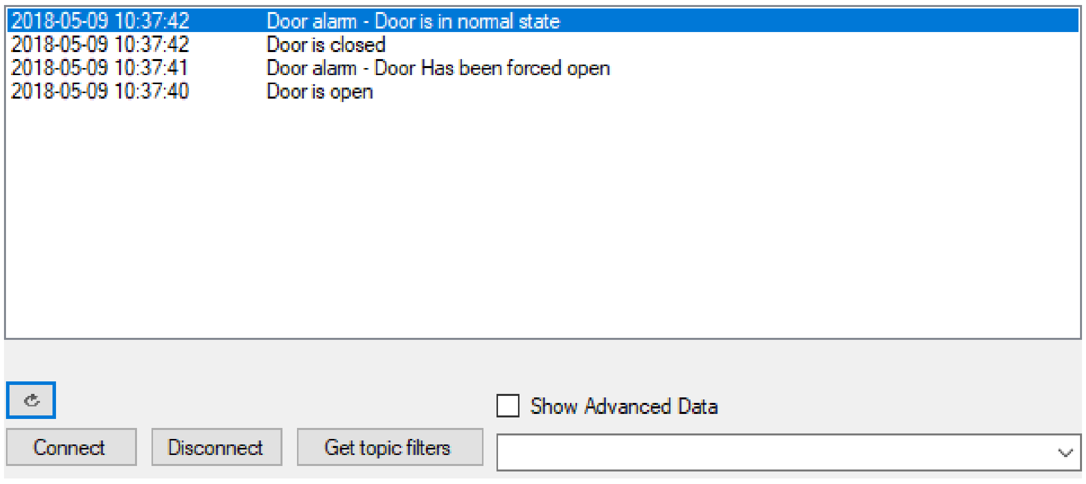
6.1 Generic event descriptions

The first thing tested was if the event could be displayed in a log, with generically generated nice names, and still be useful to a human. Figure 6.1 shows the event log displayed locally on a A1001 device, this log has hardcoded event descriptions. The A1001 log was compared to the mock GUI log shown in figure 6.2.



Time of event	Source	Event topics
05/09/2018 08:37:42	Axis-accc8e9f863c AccessController	Door alarms (Door, Normal)
05/09/2018 08:37:42	Axis-accc8e9f863c AccessController	Door - physical state (Door, Closed)
05/09/2018 08:37:41	Axis-accc8e9f863c AccessController	Door alarms (Door, DoorForcedOpen)
05/09/2018 08:37:40	Axis-accc8e9f863c AccessController	Door - physical state (Door, Open)

Figure 6.1: The old event log on an A1001 device. With hard-coded event names



```
2018-05-09 10:37:42     Door alarm - Door is in normal state
2018-05-09 10:37:42     Door is closed
2018-05-09 10:37:41     Door alarm - Door Has been forced open
2018-05-09 10:37:40     Door is open
```

Figure 6.2: The mock GUIs event log with generically generated nice names.

This was achieved by adding a new key to all events, called "NiceName" and this key's value was a short descriptive string. The main problem with this was if an event meant different things depending on what value some keys had. To combat this problem two methdos were tested, first the use of a value map. The second was to use the existing nice name tag but modifying the event2 system to allow this tag to be changed during runtime.

6.1.1 Results

When comparing figure 6.1 and figure 6.2, the generically generated nice names are just as descriptive as the hardcoded ones, and this part of the thesis wasa deemed a success.

In the log, it made no difference what solution was used, value map and nice name tag both produced the same result on the VMS side. The difference between them were more notiacble on the device side, as will be discussed in chapter 7.

6.2 Event filtering

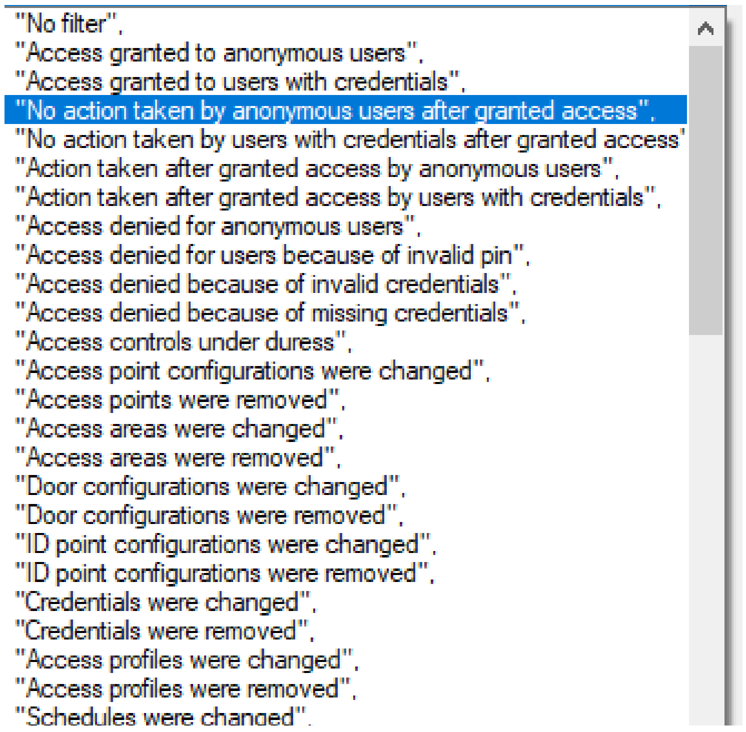
Another requested feature was the ability to filter events, and to have these filters displayed to the user in a understandable way. This was done using the ONVIF topics do distinguish between events and filter them based on this. The device would also generate list of all the topic combinations possible together with a nice name for each combination, and the VMS could use the prototype API to request this list.

The mock GUI was used to evaluate the solution. First the GUI connected to the custom daemon on the device, then a card was scanned and a door was opened and closed. After this a filter was applied to only get the door opened event.

6.2.1 Results

Good results were obtained and the mock GUI used the API to filter events without any problems. The filters, seen in figure 6.3, had descriptive names and as a user it is not hard to understand what the filter does.

Using the value map solution described in section 4.1.1 also enabled to further filter events based on values. Each event also came with a timestamp allowing to filter based on time as well.



```
"No filter",
"Access granted to anonymous users",
"Access granted to users with credentials",
"No action taken by anonymous users after granted access",
"No action taken by users with credentials after granted access'
"Action taken after granted access by anonymous users",
"Action taken after granted access by users with credentials",
"Access denied for anonymous users",
"Access denied for users because of invalid pin",
"Access denied because of invalid credentials",
"Access denied because of missing credentials",
"Access controls under duress",
"Access point configurations were changed",
"Access points were removed",
"Access areas were changed",
"Access areas were removed",
"Door configurations were changed",
"Door configurations were removed",
"ID point configurations were changed",
"ID point configurations were removed",
"Credentials were changed",
"Credentials were removed",
"Access profiles were changed",
"Access profiles were removed",
"Schedules were changed",
```

Figure 6.3: The mock GUIs filter list

6.3 Event aggregation

The main goal of the work was to come up with a concept for aggregating the events on a Axis device. It was a complex task to accomplish, and no complete concept was developed. This was due to first of all lack of time. The thesis work contained a lot of different problems and the aggregated event problem was low prioritized and the problem was underrated by us.

A good starting point was analyzed, which resulted in the emergence of important and relevant questions that needed to be solved or taken into consideration when developing the aggregation concept. The three main questions was ”where should the aggregation take place?”, ”when should the aggregation take place?” and ”how should the aggregation be done?”.

Different use-cases was made up for illustrate the need and use off aggregation, and the problem that could occur. Our concept consisted of that the device should do the aggregation, it was logical because of all events are produced by the device and we also wanted to make the VMS thinner and have the logic on the device.

The events should be changed providing some sort of identification, so that the aggregated event can consists of reference values to the events that the aggregated event is made from. This would mean that the VMS can connect the events to the aggregated event.

Different processes on the devices needs to be added, one process that will need to listen to all events, knowing when the start event of an aggregation occur. The process needs knowledge of all definitions of aggregated events, and needs to handle parallel creation of these events. It was suggested that the process creates state machines for keeping track of the state of the aggregated event.

Problems was introduced on how to handle definitions of aggregated events that was a sub part of other definitions. The work has resulted in a concept in the making, and can be used as a starting point for future development. More details on future work will be discussed in section 8. More discussions of the results will be discussed in section 7.

6.4 XML vs JSON vs Custom notation

Today the standard way to send data through the internet is to either use JSON or XML. But this thesis custom daemon uses its own custom structure to send the events, described in section 5.1.1. To evaluate what way is the most effective one, a test program was written in C#. To parse JSON an extern library was used called JSON.net, which is free for commercial use as well as one of the fastest[7]. C# which is powered by .net, has support for XML parsing, this native XML parser was used. To parse the custom notation the same parsing code used when developing the API was used.

The event used is the same as used throughout the report, shown in figure 3.1. The event data was converted into XML and JSON compatible format. The program parsed each format while measuring the time it took, it then printed the times for easy comparison between them.

6.4.1 Results

The results are shown in table 6.1. Not surprisingly it shows that the custom structure was a lot faster. This performance is because the parsing of the custom format only consists of splitting strings when the other formats need to deserialise the data into objects. The amount of data sent over the network is also less when using the custom format.

Table 6.1: Benchmarking results

Notation	Elapsed time(ms)	Characters
XML	9	584
JSON	216	469
Custom	<0	163

Chapter 7

Discussion

This chapter will provide a thorough analysis of the results obtained in the previous section. It will discuss if all the problems in chapter 4 was solved or if future work needs to be done.

7.1 Generic event description

To display events generically was a success and events could be displayed in a log without specific integration for each one. This was achieved both by using a value map and by making changes to the event2 system allowing values to be tagged with a nice name dynamically during runtime. While producing identical results on the VMS side, the difference between them are large on the device side.

The solution thas was used in the end was to have each event require a value map. The reasoning behind this will be explained here.

7.1.1 Value map VS nice name tag

One of the largest problems of this thesis was to chose between adding a "ValueMap" key to all events versus changing the event2 API to allow developers to change nice names during run-time. Both of which produced identical results on the VMS. The difference between them are more noticeable on the device.

Size of events

One thing to consider is how much extra data each event will send to the VMS. The solution that enables the event to have changing nice names will not add anything extra to the keys and values, it will simply only modify an already existing tag. The value map solution will require the events to have two additional keys and a value for each key, making the event package larger than before.

The size difference is however negligible as it is so small, and modern networks will handle it with no problems at all.

Changes to strings

Events can be changed and updated at any time, and new values could be made possible or old ones could become impossible. When this happens, the code needs to be changed. If a value map were used this would be pretty straightforward, the developer only needs to go to one place, the events declaration, and update the value map.

A lot more work needs to be done if the values nice name changed dynamically because changes would be required in different parts of the code, and there is always a risk of missing an entry. When the value map changes it can be guaranteed that all values are updated.

Filtering

To allow a user to filter events based on values, the VMS side needs to know what values are available. To achieve this the VMS needs to be able to get this information from the device. Currently theres no way to for the device to know every value a event-key can have. But this is made possible with the value map solution, because at the declaration stage the value map is also declared, if this map is concatenated on the corresponding event's filter. The VMS could parse this and extract each possible value, with the added benefit of a nice name to this value.

With the nice name tag solution, the value will only have one nice name, but it will be changed during run time. Rendering it impossible to have a list with the values that could be sent to the VMS. And filtering on values would not be possible.

7.1.2 Reduction of irrelevant data

The advanced data tag used to single out certain event keys and simply not display them works well. The problem with this solution is that there is no way of knowing who is an advanced user, and there could be different advanced users with different needs as well. A software developer and a police officer are both advanced user, but may want to view different keys.

It would be nice to be able to separate these users, but it would be impossible. However, if a user is advanced enough to have the need to view all event keys, they can distinguish the unnecessary keys from the relevant ones themselves.

7.2 Ease of use

One problem faced during the development of this new event system was that the solutions should be as easy to use as possible. If a system is too complex to use, no one wants to implement it rendering the whole thing useless and it should be a preferred alternative to make hardcoded event specific changes each time a new event is added.

7.2.1 Device side development

On the device side the event system works largely in the same way as before. The only difference is that a value map key is needed along with a nice name. The structure of these keys are very strict and is prone to human error, making development a bit harder. However, this was solved by providing helper functions that performed the formatting for the developers.

7.2.2 VMS side API

To ease development on the device side an API was developed, and this API took care of everything event related and worked as intended. One problem with the API solution is to decide which kind of API it should be, it can be a REST API or it can be a C# API.

The solution developed in this thesis is based on the use of a C# API that a VMS could implement. Another way is to use a web-based structure such as REST. There is no clear winner here as both the C# API and RESTful services have pros and cons. These pros and cons to both approaches as will be discussed in this section.

Web vs Local

The biggest difference between using REST and a C# API is where the code is executed. When using a REST API the code is run on the device instead of locally on the computer the VMS is running on. What this means is that the VMS developer themselves would have to parse the event and filter them themselves, something the C# API would do for them. This could however be solved by adding another layer and providing a C# API that did this using the REST API.

Another problem is that REST is stateless, meaning that no session information is stored on the device, each API call will result in the same response, no matter when the call was made, or by who. What this means in practice is that each time a VMS wants to get events that has happened from the device, the device has to respond with all events that has happened, even old ones that the VMS already know about, causing a lot of redundant parsing and unnecessary network traffic.

Timing

The main usage of AXIS devices is security systems, making timing a critical factor to consider. The person monitoring the system needs to know what happens as soon as they happen. The solution using the C# API will always have an open TCP connection to the devices. The daemon running on the device will then send events through this connection as soon as they appear on the device, causing little to no delay.

If a web-based solution were used, the VMS would have to poll each device to check if new information is available. Deciding the poll-rate would be hard, as too often would cause unnecessary network traffic and too long would cause a delay in the event presentation.

There exists a way of designing REST APIs allowing clients to subscribe and be notified of changes without polling, called REST HOOKs, which is a REST compatible variant of *webhooks*. The idea behind this is that a client will use HTTP POST to update a list of subscriptions on the server with a *callback URL*. When an event happens on the server, it will POST the event data to this URL[13]. Efficiently creating a subscription system and eliminating the need for polling. The drawback is that this would be a huge commitment for the VMS vendors, because this would create vulnerabilities, such as the VMS could now be the target of a DDoS (Distributed Denial of Service) attack that would force the developers to implement security measures not previously needed[2].

Parsing

Every event has the same structure, a lot of keys with corresponding values, and no event can be structured in any other way. To send an event to a VMS, all that the device need to do is to concatenate these keys and values after each other into a long string. Parsing is then easy, just split the string twice, once for the key value pair and one more time to separate them.

A custom way of representing the data is harder when using a web-based API. Everyone needs to be able to handle the data, and to ensure this a standard response format is often used, such as JSON or XML. Both of these formats are powerful and could easily be used to represent the events. In fact, they could be too powerful for the simple event structure. Packaging the event and then parsing it takes more resources when using JSON/XML than merely sending the events directly. This is usually worth it if the object to be transmitted has many different properties, or if these properties would change. But the event structure is static, and it will only ever be keys followed by values. However, the time difference between the different formats was so tiny that for a human it is not noticeable. The benefit of using a standard format that everyone can use far outweighs the performance penalty, which would not even be noticeable.

7.3 Aggregation of events

The whole idea behind this master thesis was to make the VMS lighter, taking away logic from it and transfer it to the devices, so that the VMS does not need to make specific integration for when events changes or when new events are introduced.

With aggregated events the idea to move the logic of how aggregation are made, to the devices was a logical choice. With this choice the VMS does not need to implement this aggregation logic. However it still needs to make a integration for linking and showing how an aggregated event would look like on the VMS. It is a good trade off because the device give supports for the VMS how to find the events, the aggregated event contains a list of reference to its events. The VMS then needs to search for the events in the log and link them to the aggregated event and display it as it like. This give the VMS more flexibility on how to show the aggregated event, it could be that the aggregated event becomes a drop box containing the events or something else that could be descriptive for the end-user.

It could also be so that the linking of the events to the aggregated event does not happen directly when arriving to the VMS. The device may save these references in a database for retrieving later when asked by the VMS (for example when the end-user clicks the aggregated event). This could however be slow, due to high load on the database, but more performance friendly for the VMS if a lot of aggregated events occur and needs to be displayed directly together with its events, and the VMS needs to iterate several times over the log finding these events.

Another aspect of the master thesis was that everything was to happen dynamically. Finding potential aggregated event definition needs to be done manually by developers, for every device. Also defining them into the device needs to be done manually, giving them a descriptive name. This work may becomes tedious, but it would be interesting to see the possibilities of machine learning in this area. It may be so that the device can learn how events relates to each other and can aggregate them dynamically giving them descriptive names but at the moment it seems to difficult to accomplish.

Chapter 8

Future work

This chapter will explain what needs to be done in the future. What shortcomings the solutions had and will propose ways to fix them.

8.1 Implementation into AXIS products

Throughout this project, a simple mock GUI has been used to test different approaches regarding displaying the events. This mock UI worked great for testing, but it is not a full-fledged VMS, for AXIS to use this thesis solution the API needs to be implemented into ACS. As the API takes care of everything related to events, this is just a matter of applying ACS specific styling and deciding where the log should be shown.

The custom daemon will also need to be distributed onto other AXIS devices. A lot more work then needs to be done, as every event on the devices requires the new "NiceName", "ValueMap" and "FilterNiceName" keys. But once the events are updated the devices and ACS can communicate.

The API and the daemon communicate through a TCP socket, but the standard way for AXIS products to interact is through RTSP, in the future this change may be needed.

8.2 Aggregation of events

Before making a proof of the concept, the concept needs to be more refined and fine-tuned, finding more potential use-cases to identify more possible problems.

A lot of work needs to be done to identify possible actions that can be aggregated on the devices. A good start would be to start with the A1001, identify which actions that could be of use to have aggregated. This task could be tedious, and will probably be done manually.

When the concept is done, a proof of concept may be the way to go before implementing it on the devices.

8.3 REST API

While a C# API was used for this thesis a REST API could alos be made. While there are no clear winner between the two types of APIs, a rest API could be made as an extension to VAPIX. If the event API was included in VAPIX it would eliminate the need for AXIS to maintain two types of APIs, but it would come at a larger commitment for the VMS developers as as they would need to do a lot of the work provided by the C# API themselves.

Chapter 9
Conclusion

This thesis introduced and compared different strategies to separate the development of hardware events and presentation of these events in a VMS. But to only send data was not enough as a machine is generating the events, and this makes it challenging for a human to read and understand. A way of structuring the data was proposed to make sure a human could read it without any problems, as well as other improvements necessary to increase usefulness for a user. Such as filtering of the events, both on type and values, and the ability to hide data that was deemed not important for the average user. To group related events into a larger more meaningful one would also be useful.

A API was developed that provided the necessary functionality to parse and filter events. To test this API a mock GUI was developed. In the end, good results were obtained in the area of events describing themselves, and events were shown in a log on the mock GUI perfectly understandable to a human, without any specific integration for each event. The functionality tp filter and hide unwanted data was also achieved with good results.

No real reliable way to aggregate events was found, but the resulting research revealed some key problems. These problems are hopefully useful as a base if future work was to be done on the subject.

Bibliography

[1] Daemon definition. http://www.linfo.org/daemon.html. Accessed: 2018-04-18.

[2] Ddos prevention on rest based web services. http://ijcsit.com/docs/Volume%205/vol5issue06/ijcsit2014050689.pdf. Accessed: 2018-05-14.

[3] Dynamic-link libraries. https://msdn.microsoft.com/en-us/library/ms682589.aspx. Accessed: 2018-04-26.

[4] Effective tdd for complex embedded systems. http://www.pathfindersolns.com/wp-content/uploads/2012/05/Effective-TDD-Executive-Summary.pdf. Accessed: 2018-05-09.

[5] Gmutex documentation. https://developer.gnome.org/glib/stable/glib-Threads.html#GMutex. Accessed: 2018-05-09.

[6] How to design a good api and why it matters. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/32713.pdf. Accessed: 2018-05-04.

[7] Json.net. https://www.newtonsoft.com/json. Accessed: 2018-05-14.

[8] .net framework class library. https://msdn.microsoft.com/en-us/library/gg145045(v=vs.110).aspx. Accessed: 2018-04-26.

[9] Onvif profile c specification. https://www.onvif.org/wp-content/uploads/2017/01/2013_12_ONVIF_Profile_C_Specification_v1-0.pdf. Accessed: 2018-04-26.

[10] Our mission. https://www.onvif.org/about/mission/. Accessed: 2018-04-26.

[11] Representational state transfer (rest). https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. Accessed: 2018-05-16.

[12] Time to market (ttm) — what it is and why you should care. https://www.arenasolutions.com/resources/articles/time-to-market/. Accessed: 2018-04-26.

[13] What is a webhook? https://webhooks.pbworks.com/w/page/13385124/FrontPage. Accessed: 2010-05-16.

[14] Axis Communications. A1001. https://www.axis.com/files/datasheet/ds_a1001_51155_en_1710.pdf. Accessed: 2018-04-18.

[15] Axis Communications. On-screen control documentation. https://www.axis.com/partner_pages/vapix_library/#/subjects/t10037719/section/t10122383/display. Accessed: 2018-02-16.

[16] Axis Communications. Vapix library (internal). https://www.axis.com/partner_pages/vapix_library/#/. Accessed: 2018-04-16.

[17] Axis Communications. What is acap? https://www.axis.com/support/developer-support/axis-camera-application-platform. Accessed: 2018-04-16.

[18] Alexander Larsson Sven Herzberg Simon McVittie Havoc Pennington, Anders Carlsson and David Zeuthen. D-bus specification. https://dbus.freedesktop.org/doc/dbus-specification.html. Accessed: 2018-01-30.

EXAMENSARBETE Generic event integration in Video Managment Software
STUDENT Julius Barendt och Kim Fransson
HANDLEDARE Boris Magnusson (LTH)
EXAMINATOR Ulf Asklund (LTH)

Generisk presentation av hårdvaruhändelser på mjukvara

POPULÄRVETENSKAPLIG SAMMANFATTNING **Julius Barendt och Kim Fransson**

Idag består många säkerhetssytem utav stort antal hårdvaruenheter, så som kameror eller passersystem. Med många enheter kan det vara svårt att hålla koll på allt som händer. För att göra detta lättare så kan enheterna skicka ut meddelanden varje gång något intressant har skett. I detta arbete analyserades ett system som idag har problem med att visa upp dessa meddelanden på ett konsekvent sätt, på grund av hård kopplad relation mellan hårdvaran och mjukvaran i systemet.

Detta arbetet har tagit fram och utvärderat ett flertal strategier med att separera utveckling av hårdvaruhändelse från utveckling av mjukvaran för presentation av hårdvaruhändelser. Ett sätt att strukturera data föreslogs för att säkerställa att en person kunde läsa den utan några problem, liksom andra förbättringar som var nödvändiga för att öka användbarheten för användaren. Såsom filtrering av händelser, både på typ och värden, och förmågan att dölja data som inte ansågs vara viktigt för användaren. Goda resultat uppnådes i fråga om mänsklig läsbarhet och händelser i en logg som är helt förståelig för en människa utan någon specifik integration av mjukvaran.

Dagens moderna säkerhetssystem kan bestå utav stora kvantiter av hårdvaruenheter, som till exempel nätverkskameror och passersystem. Dessa enheter producerar massvis med information dagligen i form av händelser. Specialiserad programvara existerar för att kunna samla in dessa händelser och presentera dessa för användaren av systemet. Det är av stor vikt att dessa händelser innehåller läsbar information så att användaren kan förstå vad som har hänt. Det är viktigt att dessa händelser enbart kan tolkas på ett sätt och att det är fullständigt klart vad det är för händelse. I ett säkerhetssystem finns det inte rum för misstag och minsta feltolkning kan leda till katastrof.

Lösningen erhölls genom flera olika viktiga processer, med att först identifiera de nuvarande problemen med systemet och analysera dessa. Ett flertal prototyper konstruerades såsom ett imiterad logg UI för att kunna utvärdera händelsernas läsbarhet.

Som en del av examensarbetet har undersökningar om möjligheten till att aggregera events gjorts. Denna undersökning tog upp svårigheterna med aggregation och problemen som kan uppstå vid detta.

Examensarbete genomfördes åt Axis Communications AB, ett globalt marknadsledande företag inom nätverkskameror.