Executive Summary

Previous deliverables described the requirements needed for realising the vision of the IoT@Work project towards introducing IoT technologies into automation plants and in particular enabling the secure Plug&Work of devices and services in such a domain. They described the internal specification of crucial support technologies needed for virtualizing the network infrastructure into separate network slices, thus isolating applications from others that may misbehave, for enabling a secure event-based communication pattern, for continuously checking the validity of the connected devices, etc.

This deliverable takes a different view of all these technologies – that of the application developer who needs to know how to use these technologies effectively. It thus presents their high-level architectural interconnections, to make their interactions clearer, then presents their application programming interfaces (API) and how they can be used in a specific complex scenario, so as to present all the previous information from a specific practical context.
## Document History

### Version History

<table>
<thead>
<tr>
<th>Version</th>
<th>Status</th>
<th>Date</th>
<th>Author(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Initial Document</td>
<td>25/01/2012</td>
<td>C. Kloukinas, City</td>
</tr>
<tr>
<td>0.2</td>
<td>Draft</td>
<td>27/04/2012</td>
<td>M. Comolli, F. Corvino, S. Piccione, D. Rotondi, R. Tarli, TXT</td>
</tr>
<tr>
<td>0.3</td>
<td>Draft</td>
<td>04/05/2012</td>
<td>C. Kloukinas, City, J. Claessens, EMIC, A. Houyou, J. Walewski, Siemens</td>
</tr>
<tr>
<td>0.4</td>
<td>Draft</td>
<td>30/09/2012</td>
<td>C. Kloukinas, City, J. Claessens, EMIC</td>
</tr>
<tr>
<td>0.5</td>
<td>Draft</td>
<td>19/11/2012</td>
<td>C. Kloukinas, City, J. Claessens, EMIC</td>
</tr>
</tbody>
</table>
| 0.6     | Draft  | 10/12/2012 | A. M. Houyou, H.-P. Huth, K. Fischer, Siemens, L. Dürkop, J. Imtiaz, inIT,
|          |        |            | M. Comolli, F. Corvino, S. Piccione, D. Rotondi, R. Tarli, TXT            |
|          |        |            | C. Kloukinas, City, J. Claessens, EMIC                                   |
| 0.7     | Draft  | 17/12/2012 | All                                                                       |
| 0.8     | Draft  | 17/01/2013 | All                                                                       |
| 0.9     | Draft  | 25/01/2013 | A. M. Houyou, Siemens, M. Comolli, F. Corvino, S. Piccione, D. Rotondi, R. Tarli, TXT |
| 1.0     | Draft  | 01/02/2013 | A. M. Houyou, C. Kloukinas                                               |
| 1.1     | Final  | 08/03/2013 | C. Kloukinas, City, A. M. Houyou, Siemens                                |
| 1.2     | Final  | 04/07/2013 | A. Houyou, Siemens                                                        |
Summary of Changes

<table>
<thead>
<tr>
<th>Version</th>
<th>Section(s)</th>
<th>Synopsis of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>All</td>
<td>Initial Document structure</td>
</tr>
<tr>
<td>0.2</td>
<td>2</td>
<td>Initial description of Event Notification Service</td>
</tr>
<tr>
<td>0.3</td>
<td>All</td>
<td>Document restructuring</td>
</tr>
<tr>
<td>0.4</td>
<td>All</td>
<td>Document restructuring, initial bootstrapping scenario</td>
</tr>
<tr>
<td>0.5</td>
<td>All</td>
<td>Initial architecture functions</td>
</tr>
<tr>
<td>0.6</td>
<td>2, 3</td>
<td>Detailed architecture functions and scenario, first version of management</td>
</tr>
<tr>
<td>0.7</td>
<td>All</td>
<td>Further detailed functions, scenario, management</td>
</tr>
<tr>
<td>0.8</td>
<td>All</td>
<td>Consolidation and corrections</td>
</tr>
<tr>
<td>0.9</td>
<td>All</td>
<td>Integrating changes from internal review</td>
</tr>
<tr>
<td>1.0</td>
<td>All</td>
<td>Formatting and editorial last check</td>
</tr>
<tr>
<td>1.1</td>
<td>All</td>
<td>Formatting and editorial last check</td>
</tr>
<tr>
<td>1.2</td>
<td>-</td>
<td>Some minor formatting corrections.</td>
</tr>
</tbody>
</table>
## Contents

1 Introduction ............................................................................................................. 10
   1.1 Scope of WP2 and Task 2.4 ........................................................................... 10
   1.2 Objective of this Deliverable ....................................................................... 10
   1.3 Relationship with other tasks and WPs ..................................................... 10
   1.4 Structure of this document ......................................................................... 11

2 IoT@Work Architecture: Functions under Analysis and Dependencies .................. 12
   2.1 Components used/integrated in the pilot scenario: .................................... 12
      2.1.1 Communication Plane ......................................................................... 12
      2.1.2 Application configuration/orchestration plane ..................................... 12
      2.1.3 Security Plane ..................................................................................... 15
   2.2 System Pre-configuration Parameters ........................................................ 16
      2.2.1 IoT@Work Native Devices .................................................................... 17
   2.3 Security Pre-configuration Parameters and Mechanisms ............................ 18
      2.3.1 Interface for Issuing Capabilities ....................................................... 18
      2.3.2 Device Credentials .............................................................................. 19
   2.4 Application Engineering and Planning ....................................................... 20
      2.4.1 Interface for the Directory Service .................................................... 20
         2.4.1.1 RESTful Interface ..................................................................... 21
         2.4.1.2 Interfacing OPC UA devices with the Directory Service .......... 21
      2.4.2 Interface for Configuring Event Namespaces ....................................... 25
      2.4.3 Automation Application ....................................................................... 25
   2.5 System Bootstrapping and Dependencies .................................................. 25
      2.5.1 Network bootstrapping including slice initialization ............................. 25
      2.5.2 Bootstrapping of native device ............................................................ 28
      2.5.3 Application Bootstrapping ................................................................. 28
      2.5.4 Bootstrapping of the Domain Controller .......................................... 29

3 IoT@Work Reference Bootstrapping Scenario ................................................... 31
   3.1 The High Level Business Scenario ............................................................. 31
   3.2 Bootstrapping Security Components ........................................................ 32
      3.2.1 Access to the Maintenance Application ............................................ 32
      3.2.2 Obtaining Initial Network Access ..................................................... 34
   3.3 Application Configuration and Supporting Services .................................... 37
      3.3.1 Connecting to the Directory Service ................................................ 37
      3.3.2 Registering with the ENS ................................................................. 38
         3.3.2.1 Connecting to the ENS .............................................................. 38
3.3.2.2 Publishing events ................................................................. 40
3.3.2.3 Subscribing to events ............................................................ 42
3.4 Example: Energy Monitoring Device ......................................... 44

4 Application Management Services .............................................. 46
  4.1 Event-Based System Architecture ............................................ 46
  4.2 Event-Driven Applications ...................................................... 46
    4.2.1 Publishing Events from Rules – Choosing Namespaces ........ 48
    4.2.2 Subscribing to Events – Choosing Namespaces .................. 49
  4.3 Events for Run-time Monitoring ............................................. 49
  4.4 Minimal Event Information .................................................... 50

5 Conclusions ............................................................................... 52

6 References ................................................................................ 54
List of Figures

Figure 2.1 IoT@Work Architecture Key Functions and API ........................................... 14
Figure 2.2 Bootstrapping device credentials ................................................................. 20
Figure 2.3 On demand interaction between OPC UA devices and Directory Service ..... 23
Figure 2.4 On bootstrapping interaction between OPC UA devices and Directory Service .............................................................. 24
Figure 2.5 Activity diagram of network bootstrapping (including slice initialization). (Yellow: OS and “non-IoT@Work” network management performed or initiated by the OS.) .............................................................. 27
Figure 3.1 Capability issuing due to maintenance task delegation ............................... 33
Figure 3.2 NAC Authentication Sequence Diagram (Fig. 5.7 of D3.2) ......................... 35
Figure 3.3 Device Booting for IoT@Work-aware Devices and native Applications ... 36
Figure 3.4 Device Booting for IoT@Work-aware Devices and non-native Applications ............................................................................................................. 37
Figure 3.5 Enquiring the Directory Service ................................................................. 37
Figure 3.6 Lifecycle of an ENS communication session (ENS client standpoint) ....... 40
Figure 3.7 ENS session usage (publish) ...................................................................... 41
Figure 3.8 EMD connection to the ENS for publishing .............................................. 42
Figure 3.9 ENS session usage (subscribe) .................................................................. 43
Figure 3.10 EMD bootstrapping summary ................................................................. 45
List of Tables

Table 4.1. Event rule F - Filter measurements ......................................................... 47
Table 4.2. Event Rule C - Exerting control ............................................................. 47
Table 4.3. Event Rule P - Valid publisher ............................................................... 49
Table 4.4. Event Rule S - Valid subscriber .............................................................. 49
## List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMQP</td>
<td>Advanced Message Queuing Protocol</td>
</tr>
<tr>
<td>APP</td>
<td>Applications ranging from mobile applications to service mashups and single process non-distributed applications</td>
</tr>
<tr>
<td>CEP</td>
<td>Complex Event Processing</td>
</tr>
<tr>
<td>CSI</td>
<td>Communication-service interface</td>
</tr>
<tr>
<td>DS</td>
<td>Directory Service</td>
</tr>
<tr>
<td>ECL</td>
<td>ENS Client Library</td>
</tr>
<tr>
<td>ENS</td>
<td>Event notification system</td>
</tr>
<tr>
<td>EMD</td>
<td>Energy Monitoring Device</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hyper Text Transfer Protocol</td>
</tr>
<tr>
<td>I/O Device</td>
<td>Input/Output Device with digital to analogue and analogue to digital converters</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>IoT@Work</td>
<td>Internet of Things at Work (FP 7 project)</td>
</tr>
<tr>
<td>NAC</td>
<td>Network Access Control</td>
</tr>
<tr>
<td>NCS</td>
<td>Name space configurator, offers a GUI to define ENS name spaces, and push them to the publishing elements.</td>
</tr>
<tr>
<td>OPC UA</td>
<td>Object Linking and Embedding for Process Control – Unified Architecture: A new set of standards that incorporates all of the functionality of the former OPC standards (and more), but does so using cross platform web services and other modern technology.</td>
</tr>
<tr>
<td>PLC</td>
<td>A Programmable logic controller is a control system that has a user-programmable memory for storage of instructions to implement specific functions such as I/O control, logic, timing, counting, report generation, communication, arithmetic, and data file manipulation.</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>SM</td>
<td>Slice manager</td>
</tr>
<tr>
<td>SEP</td>
<td>Slice Enforcement Point</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Scope of WP2 and Task 2.4

As described in the IoT@Work project’s DoW, the objective of WP2 is to enable Plug&Work in factory automation, “from the application level down to the Plug&Play network”. For this goal it aims at providing communication services for industrial environments that can adapt on-demand, based on the underlying heterogeneous communication infrastructure and the changing application needs. Along with these, it aims at minimising the manual steps required for network and application configuration, while being able to support different industrial services and applications and their respective requirements.

Task 2.4 of WP2, of which this deliverable is a part of, concentrates on cross-layer support and the middleware infrastructure. It considers the interfaces to and from applications for making use of the different components and services that have been developed in IoT@Work.

1.2 Objective of this Deliverable

This deliverable presents the overall architecture of the various components and services that have been developed in IoT@Work from an application’s point of view, describing exactly how an application can interface and interact with each one of them.

It provides the technical specifications of the various APIs and the ideas behind them so that they can be used productively. Then it shows how these APIs can be applied in the setting of a particular scenario – a reference bootstrapping scenario that considers the implementation of a complex business task using IoT@Work functionalities, and how management application services can be structured and employed to take advantage of the underlying services and aid in rendering the system more robust, without reducing its overall agility.

1.3 Relationship with other tasks and WPs

Earlier deliverables concentrated on the technical specification of different components and services to enable the application of the IoT vision on a factory automation context, and in particular of the secure Plug&Work goal for devices and services used in an industrial setting. These deliverables and tasks, e.g., D1.2 [13], D2.2 [5], D2.3 [6], and D3.2 [3], considered the problem from the viewpoint of a developer of such support technologies – someone who would like to produce an industrial-strength implementation of the technologies prototyped within the IoT@Work project and understand the reasons behind some of the decisions taken in their development. That viewpoint is however very far from the viewpoint of the person who needs to use these technologies in order to support a specific automation plant – what we may call the “end programmer”. While it is important to describe how the slice system or the event notification system can be implemented internally, it is equally important to describe clearly how one can use these effectively in order to develop robust components, suitable for use in automation plants. This deliverable aims to fill this void, taking a developer through each of the different technologies and how these can be used. It does so both by providing the application programming interfaces (API) of each technology and their architectural assumptions and by presenting their use through a specific complex scenario mirroring a real-world situation whereby three different organizations are using the technologies developed within the IoT@Work project to collaborate effectively and securely within the same automation plant.
1.4 Structure of this document

The second section of this document concentrates on describing the overall architecture of the components and services developed within IoT@Work from an application’s point of view. It describes their interfaces, interaction patterns and assumptions, so that application developers can use them productively.

The third section shows how the services and components described in the second section can be used in a specific real-world business scenario that involves multiple stakeholders (factory owner, factory equipment manufacturers, external equipment maintainers), and processes, in order to make it easier for readers to understand how the technologies described in section two can be applied in practice.

Finally the fourth section considers the high-level services for managing applications and services through the mainly event-based approach followed within the IoT@Work project, while section five concludes the document.
2 IoT@Work Architecture: Functions under Analysis and Dependencies

2.1 Components used/integrated in the pilot scenario:

The IoT@Work architecture could be depicted in a set of key functions that support the management of the “Internet of devices” found in the factory field. Whereas, within the earlier deliverables D1.2 [13], D3.2 [3] and D2.3 [6], we dived in some of those respective functions and their specification, we only take into this deliverable those functions involved in dealing with the programmability of the infrastructure for the needs of applications. The interfaces between the application programmer and system administrator are explored in this section in more details.

2.1.1 Communication Plane

The communication plane integrates mainly communication and network management functions that interface with the slice management system (specified in D2.3) and those components dealing with providing a semantically programmable message bus – the Event Notification Service (ENS) introduced in D1.2. The way these communication services are orchestrated upon configuration of different applications is demonstrated through the bootstrapping process already discussed in D2.2 [5] and D2.3.

We now consider the functions involved during such an orchestration of a communication service (or slice) for the purpose of a newly “plugged” IoT@Work device. Before the device is allowed to connect or initiate a new application, network connectivity has to be established and provisioned. This means:

- The newly connected device is first auto-configured through layer-2 and layer-3 protocols for link-up discovery, and network IP address configuration.
- The Slice-Enforcement Point (SEP) is started in every network node.
- The detected local-network topology is registered through each SEP with the slice manager (SM); if authentication is needed then SM checks a pre-configured certificate against a public key (network-access control for network nodes is a security plane function).
- The topology data-base in SM is populated with information such as active node interfaces, direct node neighbours, network-node semantics (e.g., which type of switch, scheduler capabilities, etc.).
- The system establishes Slice0 a.k.a. “Guest Slice”, which connects all nodes and enables controlled bootstrapping of higher-layer services and applications.

2.1.2 Application configuration/orchestration plane

The IoT@Work architecture suggests that it is possible to isolate application domains so that they can influence each other neither during their operation nor during their configuration. In comparison with today’s configuration tools that detail for each machine or sensor in the factory a whole range of configuration data, we isolate each application interacting with a given device and define configuration steps separately for each single application on its own. First, the application engineer requests a communication service that is used for operating the application, then the devices interacting with each other within that application introduce further application logic
after having presented valid credentials for this. The aforementioned configuration steps are taken care of by the following components:

- The slice manager, which could be triggered by a slice request from an application orchestration service that is capable of translating an automation planning information into slice specifications. The function depicted in the communication plane in Figure 2.1, is linked to the application plane in that it allows the application programming environment to define a slice request for an application (or aggregates). Such a slice request includes a list of end-nodes, communication requirements, and a mapping to a slice class, as defined in D2.3.

- The interface to the embedded application configuration service allows providing the device with the needed configuration data for the applications, e.g., device names and communication cycle information.

- In the case of a planned device (especially in the case of an engineered production system), we introduce a configuration function which:
  - Requires a service hosted in both device and planning/engineering tools;
  - Gathers device semantics upon connecting a device;
  - Contacts the planning tool to request the device name; and
  - Configures planned device name.

- The planned device, together with the services it provides, is registered with the directory service as a child of the (immediately) containing device or entity (the containment relationship should be intended not only in a physical but also a logical sense: for example the energy monitoring module attached to a production unit should be registered in the directory service as a child of that production unit, and the production unit as a child of the production cell where it is deployed).

- In order to manage access to events collected in the factory, we introduce a configurator service for the ENS namespaces, through which plant engineers can define as many namespaces as required.
Figure 2.1 IoT@Work Architecture Key Functions and API
Examples of supported applications, which are addressed in IoT@Work include:

- ENS client applications (event publishers and subscribers), as introduced in D1.2;
- CEP (Complex Event Processing), whose usage is exemplified in Section 4;
- NAC (Network Access Control) Client, a security-plane application introduced in D3.2; and
- State of the art controller/IO applications (e.g., Profinet, etc.), which are further supported in the IoT@Work architecture.

### 2.1.3 Security Plane

The security plane defines the functions that are involved in the configuration of security related policies, which are used to secure both the infrastructure (embedded security functions) and the configuration and application interactions with that infrastructure (e.g., through capability-based access control). These functions are intertwined with the management actions taking place during configuration and also making sure that the application and infrastructure are protected from security threats. In this document, we are concentrating more on the interdependencies between some of the configuration steps required for performing security checks. The bootstrapping phases and networking functions are designed in a way to also provide mechanisms to enforce security policies. Therefore, when a device is first introduced in the infrastructure, it is assigned to a limited slice. This could be also the fallback slice when an intrusion is detected. The same applies to configuring an application: the capability-based access control for publishing and subscribing to the ENS message bus allows controlling who is listening to or is publishing the data that is produced from the factory infrastructure. This access control envisages also the definition of a Policy Decision Point and a Revocation Service to respectively check the validity and enforce the revocation of the issued capabilities (this has been explained in deliverables D1.2 and D3.2).
2.2 System Pre-configuration Parameters

Pre-configuration parameters are configuration parameters that were either installed on the respective device before power-up (off-line configuration) or those that are loaded from another component following a sequence of configuration steps (See also D2.3).

Off-line configuration is out of scope of this document.

In the case of power-up configuration, configuration parameters stored locally on a device form the basis of a controlled bootstrapping sequence. Dependencies between the configuration parameters may exist. As an example of such a configuration sequence, assume a device-local service which needs to contact some other service on another device. In this case this service may have a pre-configured symbolic name (i.e., URL) stored locally in a configuration file. In order to use this name, the service must be able to resolve this URL into an IP address. For that, the device must already have the address of a Domain Name Service, which is probably not a pre-configuration parameter but a result of the boot of the network protocol stack. The network protocol stack, however, needs a pre-configured MAC address in order to boot. Eventually, additional pre-configured credentials are needed for accessing the network. This example shows that pre-configurations are important but typically form only the basic prerequisites for allowing more automated and sophisticated configuration of different application domains.

The bootstrapping of an IoT@Work native device (see 2.2.1) and applications takes place at the moment the device is able to reach and interact with the architecture functions/services, depicted in Figure 2.1, until the distributed application can start running correctly. For most production-related devices, the application logic and access rights could be defined at the programming or design phases of an engineered system (i.e., as part of a planning tool), in which devices are listed, identified, and programmed using a virtual model only of the real device. As a result of the planning phase, each known device can be associated with a list of configuration information (or virtual representation of the device).

The functions of the IoT@Work architecture offer the possibility to split several key configurations into modular and hierarchical configuration data that are loosely coupled with each other.

The bootstrapping addressed in IoT@Work is the sequence of configuration steps, whose final result is the correct and secure functioning of the applications as they have been programmed. For the bootstrapping steps to occur correctly, we require both device and system functions to be ready for the bootstrap. This means:

1. The planning of the system includes some pre-configuration at function level. The results of this pre-configuration include configuration files, scripts, data elements, all of them related to functions within the IoT@Work architecture. An example for such functions are ENS name spaces and capabilities, which are pre-defined for each sensor or actuator offline and which can be transferred to the device during bootstrapping.

2. There exists at least one method that allows the planning tools to automatically associate a unique ID to each device instance once the latter is physically connected to the network. For instance, the configuration of the device ID could be based on a distinctive and unique profile for each planned device. This offline pre-configured profile can be compared with the context of the real devices as connected to the system. This automatic gathering of the device context, will allow a comparison between real context and the expected device profile. Once the right profile is found, the device can be assigned the unique ID associated with that profile.
3. The device receives throughout a sequence of hierarchical information the bootstrapping steps. This hierarchy includes at the lowest level hardware- and network-near information (IP address, device name, digital-hardware configuration, etc.). Other information, in ascending order, are ENS names, access tokens (or capabilities), slice information, and application logic.

In this section we are mostly concerned with the pre-configuration parameters that are needed at the system management side and the interfaces that are needed in order to transfer each configuration piece of information to an IoT@Work-native device. For this we start by explaining what a native device actually is.

### 2.2.1 IoT@Work Native Devices

For the sake of this discussion, a native IoT@Work device can be seen as a HW/SW platform plus IoT@Work-specific embedded functions and interfaces. Herein we concentrate on the IoT@Work required pre-configuration parameters. Furthermore, the IoT@Work-specific parts can be structured into basic systems functions, such as the device-local SEP plus the communication-service Interface, as well as the device credentials and the ENS. Optional application-specific IoT@Work functions, such as the Profinet-OPC configuration service, may complement the IoT@Work-native device.

While the following list is applicable to both native IoT@Work network devices (i.e., switches, routers) and native end-devices, the specific content of each item may vary due to different roles.

Pre-configuration parameters for these components include:

- Network access control: secure device credentials (see Section 2.3.2)
- CSI/SEP:
  - Address of a SM, in case a discovery service for auto-detection of the SM is not used.
  - The ID of a management slice that connects all SEPs with the SM and shall enable contact during the joining phase of a device.
  - ID of the device. For this a secure device ID can be applied.
  - List of the device’s network interfaces that shall be managed by the SEP.
  - MAC address of each device managed by the SEP.
  - IP address under which the SEP can be accessed. This only applies in case auto-configuration (e.g., IPv6 auto-configuration or DHCP) is not used.
  - Default policies that are valid until some other policies are defined by the SM. This defines the desired behaviour for the time in the bootstrapping when the slice system is not yet available.
  - Optional symbolic name. This may re-use the symbolic device name used by the OS or SNMP.

---

1 This allows to build gateways between slice clouds and legacy networks, i.e., by telling the SEP to use only certain interfaces.
• Event namespaces: definition of the event namespaces concerning the events dispatched by the ENS and involving the device.

• Capability-secured resources: issuing of the capability tokens authorising the device to access secured resources (e.g., ENS, network slices, etc.)

• ENS: configuration files of a device acting either as ENS publisher or subscriber (see Section 3.3.2.1)

• Embedded service Application Configuration (e.g., Profinet I/O and controller configuration). This service requests the necessary configuration data that enables the device to use an application-specific communication protocol. This data is provided by the application configuration service.

Furthermore, a native device needs a permanent storage for the above parameters and also means for securing access to this storage. This will be achieved by established OS functionalities. The OS on the device controls bootstrapping and thus also needs IoT@Work-specific pre-configuration tasks to do so. This can for example be scripts called in the OS boot sequence to start the SEP and eventually configure the ENS Client module or other components with the desired pre-configuration parameters.

2.3 Security Pre-configuration Parameters and Mechanisms

2.3.1 Interface for Issuing Capabilities

We described a functional component for the creation of capability tokens through a GUI in D1.2 Section 4.3.6.5. That GUI is arranged as a wizard: the user is led through a sequence of predefined steps because the issuing of a capability is complex and requires the provision of many data (see D3.2 Section 3.1.3 for an overview of the data reported in the capability schema and how these, e.g., the revocation services URIs, are used [12]).

The creation of capability tokens follows the sequence below:

IF capability is ROOT THEN
  INSERT capability issuer
  INSERT revocation services URIs
  INSERT URI of the resource to grant access
  INSERT access rights and flag at least one of them as delegable
  INSERT validity time frame
ELSE
  LOAD the parent capability token
  INSERT capability issuer
  INSERT capability subject
  SELECT the access rights to be authorised and eventually flag them delegable among the ones granted by the parent capability token
  INSERT validity time frame
ENDIF
SELECT digital signature data to be used to sign the capability
CREATE capability token
The wizard does not allow moving to the next step if the data entered in the current step are invalid. The constraints that capability tokens must comply with in order to be considered valid have been presented in D3.2 Section 3.1.4.

An application/service that needs to access a resource secured with a capability-based access control needs to load the capability token granting the required access rights and submit it to the resource manager (see D1.2 Section 4.3.6.2 for the definition of resource manager). For this reason, the application/service must be provided with the location of the capability (or set of capabilities) it owns such as a path in the file system or an URI.

### 2.3.2 Device Credentials

Before an IoT@Work native device is granted full access to the automation network, the device needs to be authenticated and authorized during the bootstrapping and network connection phase. For this purpose the device needs to be pre-configured with device-specific credentials also representing the secure device ID. These credentials are the base for the authentication step and the integrity check (see Ref D3.2).

In D3.2 we compared different approaches to bootstrap these kinds of credentials and analyzed the applicability for IoT@Work scenarios based on a specific set of Key Performance Indicators. As conclusion we identified that the manufacturer-based approach is most applicable for IoT@Work and at the same time providing a high security level.
As depicted in Figure 2.2, the devices are equipped with security credentials during the manufacturing step. These credentials are issued from a trust anchor of the manufacturer. This trust anchor must be provided to the Authentication Authority/Radius Server of the plant factory once to enable authentication of devices including complete path verification.

2.4 Application Engineering and Planning

In the following sections we present the pre-configuration activities involving middleware applications.

2.4.1 Interface for the Directory Service
2.4.1.1 RESTful Interface

The Directory Service is a RESTful service hence each entity it manages (e.g., the production unit as a whole, the services it provides, its administrative information) can be accessed through a URL and can be represented in several formats according to the media type indicated by the clients. This RESTful interface can be accessed through HTTP. The supported HTTP methods are:

- GET, for accessing the whole or a portion of the production unit semantic profile and enquiring to filter the devices in a specific subset (i.e., category).
- POST, for adding new production unit profiles as well as services to the existing profiles.
- PUT, for updating the existing profiles.
- DELETE, for deleting an entire profile or specific portions.

A refined description of the DS architecture is provided in D1.3.

2.4.1.2 Interfacing OPC UA devices with the Directory Service

A native IoT@Work device provides an OPC UA address space that consists of an OPC UA information model. Such an information model describes metadata and profile about the device and its application process. The model is based on hierarchical arrangements of nodes and their relationship to each other. Nodes are representing types, characteristics, configuration and behaviour of the device. It is essential to provide a mapping of device information model to the DS data model for seamless propagation of device application services throughout the system.

The DS defines its own data model annotated with semantics. This model is inspired by:

- The uCode Relation Model [15] because the attributes of the device profile are modelled as subject-relation-object triples. The Subject is normally an IoT@Work Thing, the Object can be another IoT@Work Thing or a primitive value, and the Relation is the profile attribute. The substantial difference between the two models is that the DS does not use uCode for naming data elements;
- The RDF data model [17] because each attribute of the device profile is modelled as a (set of) RDF statements (i.e., subject-predicate-object triples).

The DS data model, therefore, is a connected and directed graph in which nodes (vertices) represent entities, both physical and virtual ones, or primitive elements, and edges store the relationships. All concepts in the DS data model have been taken from ontologies.

The DS may act as a proxy to an OPC UA device in the sense that it may expose (a subset of) the OPC UA device profile, enriched with semantics according to predefined mappings. This mapping process can be performed in two ways:

- On demand, when the DS is enquired for data reported in an OPC UA profile. The OPC UA client module of the DS connects to the respective OPC UA server at the production unit. It browses through the information model using OPC UA native service primitives (see getOPCUAInformationModel operation in Figure 2.3). Finally, an Address Space Profile is generated and returned to the client. Such OPC UA profile is semantically annotated and formatted according to the media type specified in the initial request. Finally, the formatted semantic profile is returned to the DS enquirer. This approach can employ a cache where the annotated profiles are stored: cached profiles need be updated only when the OPC UA profile changes.
- **On bootstrapping**, during the device configuration phase (Figure 2.3). Configuration agents detect the presence of a new device and asynchronously activate the DS Client. This client, through its OPC UA client, enquires the newly plugged device to gather its Interoperable Profile (i.e., a profile compliant with an ad-hoc XML schema that defines an information model shared between the OPC UA Information Model and the DS Data Model). That profile is submitted to the DS where it is annotated semantically and stored in its own NoSQL database.

The first integration option requires an OPC UA Client module on the DS and ensures that the OPC UA-related data are always up-to-date. As the DS manages semi-permanent information, it might be awkward to enquire the OPC UA server every time a request is received, even if the usage of a cache can mitigate this issue. The second integration option is in line with the semi-permanent data storage features provided by the DS and does not require an OPC UA client on the DS. It requires that, every time the OPC UA data are updated, also the device profile on the DS must be updated. Furthermore, the second approach introduces an XML Schema defining an *interoperable model*, to support the translation from a subset of the OPC UA Address Space Profile into the DS Data Model.
Figure 2.3 On demand interaction between OPC UA devices and Directory Service
Figure 2.4 On bootstrapping interaction between OPC UA devices and Directory Service
2.4.2 Interface for Configuring Event Namespaces

The events managed by the ENS are arranged in *namespaces* that can be configured through the Namespace Configurator Service (NCS). NCS is a web service providing the following functionalities:

- Access to the list of the existing namespaces;
- Access to and editing of namespace data (e.g., adding new vertices, removing vertices, etc.);
- Namespace deletion; and
- Management of the trusted subjects for each namespace, i.e., the subjects of the root capabilities authorised to access the namespace. Indeed, access rights on the namespace are granted to all capabilities derived from such trusted capabilities.

This web service can be accessed through two interfaces:

- A RESTful interface supporting the following HTTP methods:
  - GET, to retrieve namespace list, namespace data, list of children of a specific namespace vertex, data about the trusted subjects of a specific namespace;
  - PUT, to edit namespace or vertex data;
  - POST, to add new namespaces, new children to a specific vertex, new trusted subjects to a specific namespace; and
  - DELETE, to delete namespaces, vertices and trusted subjects;
- A web application, built on top of the RESTful interface to allow end users (e.g., plant engineers) to define and manage namespaces.

A refined description of the NCS is included in D1.3.

2.4.3 Automation Application

An automation application must define its requirements with respect to the communication infrastructure. These must be stored in pre-defined description files, which are dependent on the communication protocol of the application, i.e., in Profinet the GSDML files of the devices are needed. The application configuration service can request more IoT@Work-specific information from the application. Here an OPC UA information model is used. This model must be defined and afterwards it must be filled with the application data.

2.5 System Bootstrapping and Dependencies

2.5.1 Network bootstrapping including slice initialization

In this section we provide a high-level view of the steps and functions involved in the bootstrapping of an IoT@Work network, i.e., a network that uses slices for network virtualisation and management.

After the network-bootstrapping phase, the network – that is layer-2 and the slice system – is working. IP-layer booting is not described here apart from some aspects relevant for the slice system itself.

As a side note, a system administrator or planner may also wish to use different configurations for the IP layer per slice.
A precondition for network bootstrapping is that the physical devices and links are installed and all relevant information (described below) from the planning phase is available.

It should be noted that network bootstrapping is tightly coupled with the booting phase of single devices. Some aspects of device booting are also included here, in order to enhance the clarity of our description.

The complete system boots in a loosely coupled self-organising manner. This is because strictly controlled boot sequences introduce complexity that is difficult to manage and they also greatly affect the flexibility of the bootstrapping phase itself.

Use cases to which the bootstrapping specification provided in this section shall apply include:

- Complete system boots, i.e., after installation of a new network or after a power outage of an existing network.
- Single device boot, i.e., where the system is up and running but one device has to boot. This can happen for example when installing a new device.
- Single SW component starts, i.e., starting a piece of SW (which then may be part of a larger distributed application or service).

Figure 2.5 summarizes the steps and functions/databases involved when bootstrapping an IoT@Work communication network. This figure comprises a SM activity lane and a native-device lane. Notice that this figure does not represent an implementation viewpoint. Instead it summarizes the logical view of the network bootstrapping. In the following we provide more details on each of the steps, functions, and databases involved.
Figure 2.5 Activity diagram of network bootstrapping (including slice initialization). (Yellow: OS and “non-IoT@Work” network management performed or initiated by the OS.)

Basically, network bootstrapping starts with an arbitrary sequence of devices booting (as in Section 2.5.2), which then establish layer-2 links with their neighbours. Once at least partial communication is possible, and once both native and non-native devices have completed booting, the slice system can start (see Section 2.5.4). An important step here is the registration of SEPs at the SM, a process which also involves capability and topology information upload from a SEP to the SM.
2.5.2 Bootstrapping of native device

When starting the device, the OS is naturally the first to boot. During this phase, the network interfaces of the device are also booted and the capabilities of those devices and the associated links are then available from the OS. At the next step, the OS starts pre-configured services or applications. One of these services is the SEP (see Section 2.2.1). When starting the SEP, it tries to register with the SM. If this fails, the device applies a pre-configured default policy (disable all network access, retry, etc.).

The first step a running SEP performs is to disable all network access for its device hosted processes or applications apart from its own access. That is, after the SEP starts no other application can access the network. This provides a way to enforce any network access. This means for example a switch will not forward traffic during the booting until a slice is created for that specific traffic. However, network devices such as switches or routers only block edge interfaces and not network-internal ones. In end devices, the SEP can simply initialize access to the network devices in a manner so that only the OS and the SEP itself can communicate with other devices.

Once the SEP boot is complete, applications can be started either by OS boot scripts or initiated by a user or some other event. Starting applications may rely on the CSI as an aid for performing the respective SEP commands. The CSI is a set of helper tools and a library that facilitates the dialog with the local SEP instance and is also independent of any implementation details of the SEP. As such, the CSI does not need to boot on its own and does not need its own pre-configuration parameters.

2.5.3 Application Bootstrapping

In this section, applications are viewed as locally running code or a group of locally running processes. Applications that need to communicate with remote instances must use a predefined slice or slices for this communication. That is, any application must be associated with one or more slices. This is done by assigning one or more slice endpoints to this application. So, in the process of starting an application, a slice or slices already need to be established, and then one needs to create end points. These end points then act as interfaces for the applications. As deliverable D2.3 already pointed out, there are two ways to accomplish this:

1. The slice is already there and an endpoint for that slice is already assigned. A slice endpoint from the perspective of an application is a (virtual) network interface, so it can be used in exactly the same way as a standard network interface.

2. The slice has no endpoint in the device where the application runs. In this case, the SEP must construct the local endpoint before the application can use it. This may also involve steps to construct a slice in the system in case it does not exist yet. The SEP can try to initiate a new slice by requesting it from the SM. If the slice already exists in the system, the SEP attaches a new endpoint to that slice with the assistance of the SM.

Since an endpoint constructed by the SEP is a layer-2 network interface, it must be additionally configured for IP layer access. This configuration relies on the various mechanisms available in the network stack of the host OS. For example, the SEP can perform IPv6 auto-configuration through the OS.

Once the required slice endpoint(s) are available, the SEP applies additional access restrictions, so that only authorized applications can access the new slice endpoint. This access restriction also uses mechanisms provided by the host OS. For example, the SEP may change ownership and read/write access of an endpoint in order to allow access from the application.
Now the application can start to communicate. The steps described above can be carried out by an IoT@Work-aware application (see below) or – on behalf of the application – from a management application, a boot script, or an IoT@Work-aware wrapper. To ease and standardize communication with the SEP, an application interface provides a set of tools and methods, which form the Communication Service Interface (CSI, see deliverable D2.3). While the CSI strictly spoken is an optional component, we believe its use will greatly ease the use of slices and it decouples applications from the actual SEP implementation.

2.5.4 Bootstrapping of the Domain Controller

The slice management fulfils three main functions during the bootstrapping phase. First, it registers all device SEPs. Second, it initiates network discovery, during which it accesses (part of) the link-information databases in the native devices (see Section 2.5.2). The results from this discovery are stored in the topology database. Third, it manages slices: create, tear down, and configure capabilities. It also configures the network devices in order to meet the service requirements of each slice.

We assume that the network is already operational in some default mode, i.e., it must at least support best-effort IP traffic in the local domain. A richer definition of that default mode is contingent on the underlying network technology. For plain Ethernet, this may entail that STP has been settled, LLDP info is available, and, eventually, edge devices are configured for blocking all incoming traffic. If this assumption is not true upon SM start, it will take longer until the following steps can be completed. Notice that we assume that, at a minimum, the network consists of “managed devices” in the network. These devices are equipped with SNMP and they support LLDP.

All devices that support slices must have an SEP.\(^2\)

When the SM starts, it performs the following steps:

1. Open a registration port waiting for the SEPs to register.
2. Open a management port.
3. Start collecting topology information. This step may rely on pre-configuration (i.e., a planning tool may already have produced a network topology description), or it may rely on additional, active topology detection (i.e., SNMP scan). The SEP reports the information collected locally.
4. Read the SM configuration information, which may include the definitions of initial slices.
5. Create the first slices: wait until the topology is complete and all relevant SEPs have been registered. After that, create the slices that are already defined (from the previous step or on demand from the already registered SEPs).

In order to start planning and configuring slices, the SM needs a gapless path for all SEPs participating in this slice. Other parts of the network may still be unknown since the process of SEP registration and topology detection can take some time. Since

\(^2\) Notice that a SEP is a functionality, it does not necessarily mean that a dedicated agent is running on that device. In many cases SNMP or command-line access may be sufficient for implementing basic SEP functionality. In such cases, the SM can directly manage the device. However, for enhanced performance and robustness, a dedicated SEP agent running locally on a device is the best choice.
SEPs and new topology information will continuously become known to the SM during network start up, a SM may wish to delay the process of setting up the first slices until the topology information is sufficiently stable and complete.
3 IoT@Work Reference Bootstrapping Scenario

3.1 The High Level Business Scenario

The following description, involving the IoT@Work partners as actors, is a realistic representation of a typical situation that may occur at the FIAT group plants and that will be reproduced, in its more relevant parts, in the laboratories of Centro Ricerche FIAT for validating the project outcomes.

FIAT has installed a production cell with laser-welding robotic equipment provided by the robotic supplier Comau. When Comau delivered the equipment, FIAT provided Comau with a set of capabilities, to allow them to perform maintenance tasks. Later on, Comau decided to outsource maintenance to TXT.

TXT runs within its infrastructure a remote maintenance service capable of tracking the status of the equipment at the FIAT installation. Besides, in case of a needed intervention, TXT can send a human operator to the FIAT facilities. While there, the operator will be connected to the remote service but would also need to access diagnostic information of the robot in question within the FIAT facilities, always in a controlled manner.

As described in D4.1 and D4.2 the IoT@Work architecture deals with three main concerns that are solved through middleware services. It supports deployment of applications that enable flexible and dynamic access to the production infrastructure. The main concerns include:

1. Supporting the secure and controlled access to the infrastructure by multiple tenants (or multiple users) for both configuring new services and allowing access from third party players to the production infrastructure.
2. The provision of network services to both planned and dynamically added or modified processes and devices in the production system. Examples for the former are automation applications and data-centric IoT applications.
3. Allowing the configuration of both devices and applications in a quasi-automatic manner (i.e., with little manual configuration). This is supported through proposed orchestration services that allow a certain level of decoupling between the planned application logic and the physical production system.

One use case addressed in IoT@Work, which poses a major challenge to the established automation technology, is the delegation of a role of support/maintenance to a third party. All this has to be done in a flexible manner. This use case entails three system concerns, which are supported by the IoT@Work architecture:

1. Allowing a secure and controlled access of the third-party maintenance application (supplied by TXT) to the robot owned by FIAT. This is predicated upon “electronic delegation” of the maintenance role from Comau (as an example of a robot supplier) to TXT agents. The granularity in controlling access to the production machine is achieved through the capability system.
2. Providing communication services that support both legacy automation applications that have strict requirements, and allow a variety of new IoT applications to access well dimensioned and isolated network slices that run and communicate in parallel to these legacy automation processes and share the same network. The network is divided into virtual slices (viz. network overlays), which permit both the dimensioning of the network resources among different applications and the enforcement of access control. The
example IoT applications in question include event-driven applications using the semantic event-notification system (ENS), which allow multiple users (subscribers) to query and register to relevant event streams (e.g., robot relevant status events).

3. Allowing a more automatic plug and play of both applications and devices. This includes the introduction of two IoT@Work devices to an already existing production infrastructure: (a) an energy monitoring device to provide events to the ENS, which could be accessed by the maintenance application provided by TXT, and (b) a TXT-owned handheld device to run the monitoring application in a TXT network slice deployed within the FIAT-owned network.

Both devices within this use case, although having different roles in the factory, are assumed to be IoT@Work devices, which means they have the following functionalities:

- OS – Device’s Operating System that supports several processes at one time.
- NAC – Network Access Control Client.
- SEP – Slice Entry Point.
- CSI – Communication Service Interface.
- IoT@Work-aware applications:
  - Legacy applications that communicate through a slice-enforced mechanism.
  - ENS Client module (configuration data and capabilities included, see Section 3.3.2.1).
  - BT – a boot script to set up the slices required by the applications.

Not all components listed will be demonstrated in the business scenario. They are described here so as to show a complete architecture and allow us to describe the complete process and interactions among various services.

3.2 Bootstrapping Security Components

3.2.1 Access to the Maintenance Application

The delegation of maintenance tasks starts from the capability issuing process shown in Figure 3.1.
Figure 3.1 Capability issuing due to maintenance task delegation
System pre-configuration entails FIAT issuing capabilities to Comau so that they can connect to FIAT’s Configuration Slice and through that to the welding robot that they have provided to FIAT for maintenance purposes. Comau itself uses this capability to issue a new capability for TXT to be able to connect to the slice, as it has outsourced the maintenance of the robot to TXT. A similar issuing of capabilities is carried out for the capabilities required for connecting to the ENS both for publishing and subscribing (not shown in Figure 3.1).

The newly installed equipment comes with an Energy Monitoring Device (EMD) that needs to have at least two capabilities: one authorizing it to connect to the FIAT Configuration Slice for being able to access the factory network and another one authorizing it to connect to the ENS for publishing events. In addition, the device requires the nodes of the event namespaces to which it has to publish events. Indeed this kind of information might be automatically inferred by the device from the resource ID included in the capabilities granting publishing rights to it.

A TXT employee, in charge of a periodic maintenance activity, enters the vehicle body production line of a FIAT factory and walks to a given welding cell, in order to perform equipment maintenance. He carries a portable tablet that he uses for interacting with the environment. He points the tablet’s camera towards a QR code that is on the cell’s robot, with the intention of obtaining information about the interfaces provided by the robot and its associated devices (sensors, actuators, etc.).

This requires connecting with the FIAT system processes and interacting with them in a secure manner. FIAT needs to ensure that such connections are authorised, so as to avoid unauthorised parties from accessing information about the factory configuration and its operation and potentially modify safety and/or business critical processes.

One of the services that the maintainer requires is the monitoring of the energy consumption of the robot. The maintainer’s tablet needs to receive data from this service, which is done by connecting to the ENS at the appropriate event namespace and by subscribing to a specific namespace subset. The use of the ENS ensures that the tablet will be able to connect to the appropriate namespace even when the energy consumption service is unavailable (e.g., because the respective device is momentarily plugged out of the system). Indeed, the ENS allows a complete decoupling of event message publishers and subscribers, thus increasing the robustness of the system and at the same time facilitating the replacement of system components at run-time.

3.2.2 Obtaining Initial Network Access

An IoT@Work aware device, also called a “native device”, is a device that hosts a slice enforcement point (SEP). Depending on its role and placement, it can be part of the network infrastructure (i.e., a router or a switch) or an end device.

The following description depicts the steps until applications are able to communicate using a slice. The description is simplified; it only explains the good case without discussing error handling and the IP layer configuration of a slice interface has been left out as well. The latter can use any state-of-the-art means such as DHCP/DNS or IP address auto-configuration. More details are available in deliverables D2.1 and D2.3. There is also a significant amount of complex interactions between the SEP and the local operating system, which has also been left out for increased clarity. For a showcase of these issues see the demonstrator SW architecture.

For network devices special pre-configuration parameters must be applied to enable them to communicate in a first step where neither network access control nor the slice system work. These pre-configuration parameters must ensure that network
devices can communicate between each other ("form a network") and can reach important infrastructure service such as the slice manager and the RADIUS service [7]. Furthermore, after that initial step all end devices are blocked but then a default signalling slice is established to allow native devices to contact a slice manager. After the slice manager is available and all SEPs have been registered, the slice manager installs eventually pre-planned slices from a configuration. In the reference scenario, this initial slice is pre-established by the slice manager.

After these steps, the network including the slice system is fully functional and the end devices can boot.

In both cases (a device permanently installed in the network or a mobile visitor), the first step is Network Access Control (NAC), (also discussed in [3], [5]). NAC needs connectivity to the first edge network device (cable, wireless) and is not affected by the blocking step described above. It essentially bypasses the slice system (however the system administrator may optionally decide to construct a separate slice for NAC internal data transport). In the reference scenario, NAC is simulated by means of WLAN access procedures for the mobile devices and it has been left out for the fixed devices (servers).

![Network Access Control Diagram](image)

**Figure 3.2 NAC Authentication Sequence Diagram (Fig. 5.7 of D3.2)**

Next, the SEP in IoT@Work aware devices must register at the slice manager (Figure 3.3 and Figure 3.4 shows the device boot). Note, as the previous step assured device integrity, the system operator in a production system may decide to do without additional authentication between SEP and slice manager and rely on the NAC instead. There is a slight difference in the threat posed by a device capable of connecting at the field level, e.g., the energy monitoring device, and the third-party handheld device connecting to ENS as a client application. The former should be both checked in terms of authentication and in terms of device integrity, while the latter only requires authentication.
Now applications can be started, e.g., by means of auto-start mechanisms\(^3\) found in the host operating system or by manual start.

IoT@Work aware applications can request access to a slice on their own. To simplify the processes of setting up a local slice interface and attaching to a slice, former deliverables already decided to provide a set of functions to support the slice signalling for applications. This is the "Communication Service Interface" (CSI), which exists both as a traditional programming library and as a set of executable tools. The CSI is an implementation of an interface to the slice system, not an independent component. The CSI however uses the SEP as a proxy, so there is no additional communication channel needed beside the normal SEP/Slice Manager channel. The reference scenario assumes pre-defined and pre-established slices, so the application just needs to attach to that slice.

If the application is not IoT@Work aware but the device is (has its own SEP), then the steps above can be done by another application (i.e., a wrapper script or boot script) on behalf of that legacy application (Figure 3.4).

If the device does not have its own SEP, a network device at the edge must provide the functionality.

---

\(^3\) Modern OS may also support hot-plug behaviour, i.e., starting an application once a slice becomes available.
3.3 Application Configuration and Supporting Services

3.3.1 Connecting to the Directory Service

Once the monitoring device has access to the network, it can contact the Directory Service (DS) to enquire about properties of the system and its components. An application (or a middleware service) sends a request to the DS (i.e., an HTTP GET to the URL that identifies a device or a piece of information on a device). When the request is accepted, the DS responds with the semantically annotated (portion of) device profile. Figure 3.5 shows the sequence used for enquiring the DS.
The sequence diagram above omits the validation of the credentials presented by the directory service client because the DS is secured by the standard authentication mechanisms defined for HTTP.

In the application scenario a TXT maintainer walks to a manufacturing cell, points a smart-phone/tablet device to a production unit and obtains the URL that represents the unique identifier of the device in the DS. The DS client running on the mobile device performs a HTTP GET on that URL. This request is handled by the REST interface manager of the DS that, through its query manager, retrieves the profile of the production unit from its NoSQL storage. This semantic profile is formatted according to the media type specified in the initial request. The result formatted semantic profile is returned to the DS client.

The monitoring application running on the mobile device uses the information reported in the profile to connect to the energy monitoring event stream provided by the production unit (if any).

### 3.3.2 Registering with the ENS

#### 3.3.2.1 Connecting to the ENS

In D3.2 Section 3.1.7.1 we described the ENS authorisation workflow focusing on the interaction between ENS clients and ENS Authorisation Service in both subscribing and publishing scenarios. In this section we describe how ENS clients can register with the ENS in order to publish or subscribe to asynchronous event messages.

The ENS Client Library (ECL) is a software layer that hides the complexity of handling the lifecycle of a communication session with the ENS and abstracts from unnecessary AMQP features and functions. The ENS session lifecycle flows through the following sequence:

1. Configuration of an ENSClient instance – the outcome of this step is an ENSPublisher or a ENSSubscriber instance according to the kind of operation specified in the configuration;
2. Session setting up (i.e., registration with/connection to the ENS);
3. Session usage (specific to the type of ENSClient instance created in step 1);
4. Session closing (i.e., deregistration/disconnection from the ENS).

The session closing is performed automatically when the session deadline is exceeded or the capability granting access right on the ENS has been revoked. Indeed, the ENS Authorisation Service, when determining the session deadline during the initial registration, ensures that the session deadline is not greater than the expiration time of the presented authorising capability.

The configuration data to be passed to the ECL are indicated below:

- Identifier of the application/service being registered to the ENS (hereafter referred to as subject identifier of subject ID): the subject of the authorisation capability token and the owner/subject of the digital signature data must match this identifier. If the capability is assigned to a device, its secure ID will be the subject ID.
- Identifier of the target namespace and the relating publishing/subscribing pattern: this pattern is a string identifying the subset (e.g., a branch) or the single node of a specific namespace where the events should be published or the event sources should subscribe to.
- Kind of operation to be performed (i.e., publish or subscribe).
• Location of the set of capabilities assigned to the client.
• Digital signature data required to sign the authorisation request.
• Parameters to connect to the ENS Access Request Broker, i.e., the ENS broker dedicated to the collection and dispatching of ENS authorisation requests and responses. This message broker decouples the ENS Clients and the ENS Authorisation Service.

The connection to the ENS consists of the following series of actions:

1. Retrieval of the most suitable and active capability according to the given subject identifier, operation, target namespace, and namespace pattern;
2. Preparation and digitally signing of the authorisation request;
3. Submission of the authorisation request to the ENS Access Request Broker;
4. Analysis of the authorisation response sent by the ENS Authorisation Service;
5. Opening of the connection to the assigned ENS Broker with the least permissions to perform the requested operation.

Thus, after the registration, the client is connected to the selected ENS Operative Broker and can perform the requested operation through the ECL.

The ECL raises an error if:

• No suitable and/or active capability has been found in the given set of capabilities;
• The owner/subject of the digital signature data does not match the given subject identifier;
• Cannot connect to the ENS Access Request Broker to submit the request or to receive the response;
• The response received is an authentication failure;
• Cannot connect to the assigned ENS Broker.

With respect to a generic ENS client, the API provided by the ECL contains:

• The class ENSClient, that represents a generic ENS client;
• The class ENSClientFactory, that instantiates and configures ENSClient objects;
• The class ENSEvent that models the structure of an ENS event (see Section 4.4 for details about event content and structure).

The class ENSClient provides the following methods:

• connect(), for registering (i.e., session setting up) to the ENS;
• disconnect(), for deregistering (i.e., session closing) from the ENS.

The class ENSClientFactory provides the method build(), whose input parameters are the configuration data to be passed to the ECL.
Figure 3.6 presents the sequence diagram relating to the lifecycle of a communication session with the ENS from the perspective of an application/service that uses the API provided by the ECL.

**3.3.2.2 Publishing events**

The ECL provides a very simple publishing API. Indeed, it consists in the class `ENSPublisher`, a subclass of `ENSClient`, which provides the method `publish(ENSEvent)`.
The sequence diagram presented in Figure 3.7 shows the publication process of an already registered application/service.

![Figure 3.7 ENS session usage (publish)](image)

For each event to be published, the event producer (EventProducer in the diagram) passes to the ECL an event object (ENSEvent parameter of the publish method in the diagram).

The ENSPublisher is in charge of:

- Retrieving the name of the AMQP exchange and the namespace pattern (respectively exchange name and routing key in the diagram) from the configuration information included in the initial authorisation response;
- Extracting the event headers and payload from the given event object (see Section 4.4 for details on the ENS event structure).

The ECL includes an AMQP client (represented by an AMQPBasic\(^4\) object) that builds and submits an AMQP message to the selected ENS broker (ENSOperativeBroker in the diagram). If the message submission is successful, it means that it has been collected by the ENS and will be dispatched to the registered subscribers.

An error is raised if the connection to the operative broker is lost, the message submission fails, or the publication is attempted after session expiration.

With respect to the business scenario, the EMU attached to the welding robot is the energy data event message producers/publisher. It utilises preconfigured capabilities for periodically publishing the energy event message at a namespace. Figure 3.8 describes the interfaces and sequence of operations.

\(^4\) AMQP specification version v0-9-1 defines the Basic class that implements the messaging capabilities.
The application configurator (ApplicationConfigurator in the diagram) will invoke the ENS publisher client, which will connect itself with given (predefined) capabilities to the ENS (through the ENS Authorisation Service).

1. After receiving the energy data from the remote device the application configurator running on the EMD generates an ENSEvent object (createEvent(Observed Data) in the diagram) that is wrapping around the observed energy data. It then passes on this event to the ENSPublisher instance that publishes the event on the ENS within the namespace reported in the capability initially presented.

2. The publishing step is periodically repeated until the device is disconnected or there are no more energy data available.

3. Once the device controller senses the disconnection of the device, it should trigger the application configurator which requests the respective publisher instance to disconnect from the ENS.

### 3.3.2.3 Subscribing to events

The subscribing API envisages the following classes:

- **ENSSubscriber**: a subclass of ENSClient, representing an ENS event consumer,
- **ENSEventListener**: represents the callback that carries out the actual event processing (method `onEvent(ENSEvent)`).

The class ENSSubscriber provides the following methods:
- **subscribe()**: starts the subscription to the events relating to a specific namespace subset.

- **addEventListener(ENSEventListener)**: adds the given object to the callback list (i.e., event listener list) and returns the unique identifier of the new item. One event consumer can have more than one event listener, thus reducing the number of consumers to be created.

- **removeEventListener(String)**: removes the callback object identified by the given textual string.

- **unsubscribe()**: it ends the subscription.

The sequence diagram presented in Figure 3.9 demonstrates how an event consumer (**EventConsumer** in the diagram) can use the ECL to subscribe to a specific event namespace subset.

![Figure 3.9 ENS session usage (subscribe)](image)

**ENSSubscriber** is in charge of:

- Handling the list of callbacks to be executed as soon as events are asynchronously received;
• Retrieving the name of the AMQP dedicated queue where the ENS operative broker will deliver events from the configuration information included in the initial authorisation response;

• Creating, registering and unregistering the default AMQP consumer (DefaultConsumer in the diagram) that receives AMQP messages, converts them into ENS event objects and executes the registered callback(s).

Before connecting to the ENS, the event consumer has to define and register the event listener object(s). When it subscribes, the ECL creates an AMQP consumer and registers it with the selected operative broker through the AMQP’s Basic class (AMQPBasic in the diagram); the registration returns the unique identifier assigned to this consumer. The operative broker makes a best-effort to deliver messages to registered consumers. AMQP’s Basic class handles asynchronous message delivery by converting AMQP messages into ENS event objects and executing the registered callbacks. When the event consumer unsubscribes, the ECL deregisters and stops the AMQP consumer, thus the registered callbacks will no longer be executed. If the session expires while the event consumer is still subscribed, the subscription is terminated automatically.

3.4 Example: Energy Monitoring Device

As stated in the business scenario in Section 3.1, an Energy Monitoring Device (EMD) can be installed in the production cell. The EMD is integrated into the whole system by interfaces to several IoT@Work components. The bootstrapping process of the EMD is shown in Figure 3.10. It is a summary of the detailed descriptions of the previous sections.

1) The energy values are measured by an IO component which provides them via a specific communication channel. The application configurator (Appl. Config.) must configure this channel and act as an IO-data proxy between the IO component and the other parts of the EMD. Here the IO component requests a Profinet communication channel. In that sense the IO component can be seen as a Profinet IO device and the application configurator as a Profinet IO controller. The parameterization of the Profinet channel is done automatically by the auto-configuration procedure described in Deliverable D2.3.

2) After the Profinet channel is established, the IO component sends energy values to the application configurator of the EMD periodically.

3) The Network Access Client (NA Client) authenticates the EMD to the Network Access Authority (see Section 2.3.2).

4) When the EMD can access the network, the device’s Slice Enforcement Point (SEP) presents its capabilities to the Slice Authorization Control (SAC). Afterwards the SEP registers itself with the Slice Manager (SM) of the plant’s network (see Section 2.5).

5) The EMD is now able to communicate with other IoT@Work services. The OPC UA server of the EMD presents its pre-configured information model to the Directory Service (DS) (see Section 2.4.1).

6) The EMD’s ENS client registers the EMD to the Event Notification System (ENS). The ENS client receives the energy values from the application configurator and forwards them to the ENS (see Section 3.3.2).
Figure 3.10 EMD Bootstrapping Summary
4 Application Management Services

4.1 Event-Based System Architecture

A complex, dynamic, IoT-based system cannot be fully specified statically. Components join and leave the system dynamically either due to faults or to other reasons (e.g., maintenance, addition of new devices and applications, decommission of devices and applications, etc.). System development becomes much harder if one attempts to use an architectural style like client-server. This is because clients need to know the identities of the servers they are supposed to use and unavailability of these servers causes the clients to fail.

An event-based architectural style avoids this problem by organising a system around events of interest that are published by some components and subscribed to by others. In this style the event publishers are not affected (nor are they informed) when subscribers to some events leave or join the system. Event subscribers are similarly not affected (nor informed) when event publishers leave (or join) the system – if all of them leave, they simply do not receive any more events. As a consequence of this, the system becomes more robust. It also becomes easier to extend with new types of events and new event publishers and subscribers. The new event subscribers may in fact be used to extend the system by using existing events to implement some new process, without the existing event publishers noticing so.

In IoT@Work, event communication is achieved through the Event Notification Service (ENS). This is used along with a set of event namespaces to allow events published by some component on a particular namespace to be communicated to the components that are subscribing to events of that namespace.

The event processing performed by components can be so as to implement part of the application logic itself or so as to establish another set of run-time checks of certain aspects of a system in parallel to whatever safety and security mechanisms exist already (or indeed as a way to implement such safety and security requirements). As argued in [1], an event-based approach can increase the robustness of software (a counter position is presented in [16], mainly focusing on which style can achieve a higher degree of concurrency). It should be noted that if event processing is used to implement part of the application logic that has real-time requirements then both the event processing itself and the event communication mechanisms should have real-time characteristics.

4.2 Event-Driven Applications

An example of an event-driven application is the energy monitoring part of the robot maintenance scenario described in Section 3. There the maintainer needs to receive energy consumption measurements from the robot and process them. This processing can entail something simple like a graphical representation of these events or it can involve more complex filtering. For example, only events with measurements above a certain threshold should be visualised – the rest should be ignored.

This kind of application is an example of a situation where new applications are added to the system without disrupting existing ones – the reception of energy measurements by the maintenance application does not disrupt the reception of these events by other applications that are already receiving these events nor does it require their modification.

The event-processing itself can be realised either by bespoke code or by using a complex event-processing (CEP) engine, as for example Microsoft StreamInsight.
[11], [9], Drools Fusion [8], Jess [4] (which, like Drools, implements the Java Rule Engine API [10]), Esper [2], etc. An important advantage of using a CEP engine is that the event processing logic is cleanly separated from the remaining code and is thus easier to inspect (especially so by domain experts) and modify. This is extremely important in an agile IoT-based manufacturing environment, where new (potentially unknown) devices can join and leave the system dynamically. Validating each device against a set of safety and security policies is quasi-impossible. Therefore, it does not really matter whether these already check and comply with the aforementioned policies – one must be able to convincingly argue that this is the case. A CEP makes this possible, as it is easy to inspect its higher-level monitoring rules to argue that these follow the required policies. Such an ability is particularly important for legal and regulatory requirements, where lack of conformance (in fact lack of evidence for conformance) can cause a (part of a) factory to be shut down for an extended period of time. Another gain from employing a CEP for monitoring conditions that are already checked by various components is that it adds defence in depth. The checks of individual components may fail to be performed due to some unrelated bugs of their functionality. By adding another monitoring element one decreases the probability that a check will not be performed, as the components and the CEP are unrelated entities and therefore are expected to fail independently. Finally, a CEP allows for monitoring non-local conditions that involve more than one component, as, unlike individual components, it has access to the states of multiple entities (through their events that it observes) and can correlate them. This is also extremely useful when attempting to identify the real source of a fault or unexpected behaviour – the history of monitoring rules that have been executed is essentially a centralised event log that shows how some events caused other events, etc., while filtering out events that did not participate in the execution of these rules.

Using a CEP the business logic of the robot maintenance scenario would be expressed through a high-level rule such as:

Table 4.2. Event Rule C - Exerting control

| WHEN ev1:EnergyMonitoring AND ev1.value > 35 |
| THEN DERIVE(vev1:VisualisedEnergyMonitoring(ev1)) |

Such a rule filters through all the Energy Monitoring (EM) events (ev1) and for those that have a value higher than 25 (the value of the threshold) derives a new complex event (vev1) of type Visualised Energy Monitoring (VEM), using the value of the original event ev1. The visualisation component then visualises events of type VEM instead of the basic EM ones. The logic in the rule can change easily, either by modifying the rule or adding further rules that derive VEM events under other conditions.

It is very easy to add new applications to the system – VEM-type events can be subscribed to by other components as well, for example one that produces filtered logs for the system administrators of a factory.

So far, the examples presented do not attempt to close the automation control loop, i.e., there are no control events directed to the automation layer. As far as the CEP engine is concerned, this is not something difficult – in fact, it has no knowledge of the automation layer or of any other system component (just like any other publisher and subscriber in an Event-Based Architecture). So in principle it can process rules that instruct the creation of “control command” events whenever some condition becomes true, as in the following rule:

Table 4.2. Event Rule C - Exerting control

| WHEN ev1:EnergyMonitoring AND ev1.value > 35 |

| THEN DERIVE(vev1:VisualisedEnergyMonitoring(ev1)) |
THEN DERIVE(ev2:EnterEnergySavingMode(ev1.device.cell))

Rule C produces a command event (ev2) that requests the entire manufacturing cell of the device that emitted the energy monitoring event to enter into an energy saving mode. Of course, in order for such a command event to be actioned upon, the cell’s PLC needs to be subscribing to events of type EnterEnergySavingMode, and to have the logic that allows it to treat such events.

4.2.1 Publishing Events from Rules – Choosing Namespaces

The careful reader may wonder whether deriving a command event such as the one shown in Table 4.2 makes sense or whether it can cause the fans of all cells to stop. In fact, the event rule itself is ambiguous on this matter. A simple solution to rendering such an event addressable to a specific cell is to (a) include information on the cell inside the Energy Monitoring event, in which case it will get included as well into the derived event. Then the derived event will be received by all cell PLCs but will only be treated by the cell’s PLC that corresponds to the original energy monitoring event. Obviously, this kind of solution is far from optimal, as it increases the network traffic (the derived event needs to be sent to all cells) and the amount of work each cell PLC performs (it needs to filter away events meant for other cells).

A different general solution to (a) is to (b) publish the derived event on a namespace that is cell-specific. That way, only the respective cell (along with any other application that might be interested in these events) will receive the event messages and it will know that it needs to act on them. However, rule engines have no notion of event namespaces – this is a concept introduced in the IoT@Work project to help structure event streams, be able to apply security checks in a more refined manner when accessing them, and to reduce the number of event messages that need to be broadcasted. So unless the specific rule engine can (b.1) call code to publish an event at a particular event stream (e.g., using the ENS Java API), or it can (b.2) identify output streams in some other manner, then the only solution is to (b.3) run multiple instances of the rule engine – one per namespace where derived events should be published. In that situation, each rule engine instance is configured so that its “output” stream is directed to a particular namespace.

This last solution of instantiating a different rule engine per output namespace (b.3) is much more efficient than the first one (a), both with respect to the number of event messages exchanged and the amount of message processing a PLC will need to perform. It has one drawback though – it forces all output events of a set of rules to be sent to a single namespace. If output events need to be sent to different namespaces then we either need to (b.3.1) employ an intermediate forwarding component that directs output events to their respective namespaces, or (b.3.2) employ separate rule engine instances per output namespace. In the latter case (b.3.2), the rules themselves will need to be modified so that the only events that are derived are the ones for the current namespace. This introduces two problems. First, the number of rule engine instances can become too large on occasions, and, second, it requires that the rules are modified – an exercise that (being manual) can introduce errors and will make the resulting rules both difficult to maintain and to understand.

As such, plant designers should try to use solution (b.1) or (b.2), i.e., a rule engine that can either call code directly or has a way to identify output event streams. It might still prove beneficial to create separate rule engine instances per controlled device as in (b.3), so as not to overload a single rule engine with all the different rule instances that refer to different devices – on some rule engines [14] the reduction of the response time observed in such cases is substantial.
4.2.2 Subscribing to Events – Choosing Namespaces

The previous section considered the problem of directing derived events to specific event namespaces and identified a number of possible solutions for it. However, there is also the question of how to direct the input events to a rule engine instance. If the rule engine instance needs to read events from namespaces A and B, should it be configured so that it reads from two separate event streams or from one? This apparently depends on whether the rule engine can receive events from multiple streams. If it cannot, then one needs to ensure that all the events it needs are received as a single stream.

A simple solution (a) would be to subscribe to the first namespace that contains both A and B (these are organised as a tree). However, this would require the rule engine to have capabilities for subscribing to that namespace (and thus reading more information than actually required to perform its task – a potential security risk), as well as increase substantially the number of events received by the rule engine – part of which will be subsequently ignored immediately as non-relevant. The response time degradation in this case can be extreme – in some implementations, the response time for processing an event is exponential with respect to the number of events received [14].

A better solution (b) is to introduce a wrapper component that subscribes to the required namespaces A and B, merges their event streams and forwards them to the rule engine. This component does not have to be an external one – the CEP can simply subscribe to namespaces A and B using the same event listener callback.

4.3 Events for Run-time Monitoring

As aforementioned, event-processing can be used for either implementing safety and security requirements or for replicating the mechanisms that are meant to check these (especially when a new version of a component is introduced into the system).

For example, according to the specification of the ENS component, in order for an event to be published the publishing component must demonstrate that it holds a valid publishing capability. This is something that should be checked by the ENS Authorisation Service (EAS), as discussed in Section 3.3.2.2. However, one may well wish to introduce an additional check to ensure that the implementation of the EAS actually does so, as is done in Table 4.3 below. This check utilizes a function VALID\_CAPABILITY to check whether the capability contained within a publishing event is indeed a valid one for publishing that event at that time point.

<table>
<thead>
<tr>
<th>Table 4.3. Event Rule P - Valid publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHEN pev1: Publish</td>
</tr>
<tr>
<td>AND NOT VALID_CAPABILITY(pev1.capability, publish, NOW)</td>
</tr>
<tr>
<td>THEN DERIVE(al1: ALERT(pev1, &quot;Invalid Publishing Capability&quot;))</td>
</tr>
</tbody>
</table>

A similar check can be introduced for event subscriptions (discussed in Section 3.3.2.3), so as to ensure that whenever an event is received by a subscriber, the subscriber’s capability is still valid:

<table>
<thead>
<tr>
<th>Table 4.4. Event Rule S - Valid subscriber</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHEN rev1: Receive</td>
</tr>
<tr>
<td>AND NOT EXISTS sev1: Subscribe</td>
</tr>
<tr>
<td>WHERE rev1.recipient == sev1.requestor</td>
</tr>
</tbody>
</table>

|
This last rule is interesting because the capability token presented to the EAS by the subscriber at the time of subscription may have been invalidated by the time the ENS forwards an event to the subscriber. Ensuring that this property holds without access to the code of the ENS component is extremely difficult (and even with access to the code it can be a difficult task). So when one needs to evaluate ENS implementations by different vendors, it makes sense to add basic rules like these to increase their confidence (but not prove\textsuperscript{5}) that the implementations implement the requirements correctly.

Another situation where such rules may be used, even though they are already implemented inside the respective components, is the case where the requirement is of a legal/regulatory nature – then it needs to be easy to demonstrate that the system does indeed implement it. Being able to point to a high-level monitoring rule or set of rules as a proof of compliance is far less costly than validating the configuration of a particular system (which may need to be repeated each time something changes in the system configuration).

4.4 Minimal Event Information

The AMQP protocol \cite{AMQP} used internally by the ENS is able to manage any kind of data, therefore ensuring that the ENS can manage any kind of content for events.

The actual event data (i.e., the payload of the message) can be a numeric value (e.g., the observed power consumption for namespaces devoted to energy monitoring), a string (e.g., a string describing a warning condition in a network device for a namespace devoted to collect software modules anomalous conditions), or a chunk of binary data (e.g., a snapshot of a cell periodically taken). The meaning of the information carried by the event has to be properly described in the associated semantics as referenced by the event’s Semantics URI.

Indeed the AMQP specification makes it possible to associate attributes to each event, therefore giving the possibility to the IoT@Work ENS to associate a set of standard attributes to the events processed via the ENS (standard means that these attributes are available in all event messages, independently of their type/payload and the namespace they are published in).

The current ENS architecture supports the following standard attributes associated to each event:

- **Device ID**: it is the unique identifier of the device to which the event is related to (e.g., the URL of the device in the Directory Service);
- **Media type and encoding of the payload**: this data is required to properly read the event content;
- **Semantics URI**: this URI points to a semantic description of the information carried by the event (e.g., the kind of measured parameters, the kind of measuring device, units of measures, tolerance, etc.) so that consumers can understand the informative content of the event;

\textsuperscript{5} Run-time monitoring (just like testing) can never prove that a system implements a specification (the set of monitoring rules) correctly. It can only identify mismatches between the two when these occur. As such, an error can go undetected for a long time before it is actually identified.
• **Event timestamp**: it provides information about the point in time when the event has been submitted\(^6\) to the ENS. This piece of information is automatically added by the ECL (see Section 3.3.2.1).

In addition to the standard attributes above, events can have additional specific ones. Furthermore, events arranged in the same namespace may have the same attributes. For example, events published on an *energy monitoring namespace* may envisage an attribute that identifies the measuring device to give the opportunity to monitoring applications to be aware of measures’ characteristics (e.g., measure’s precision).

Events have no local or global IDs, unless these are embedded within the event payload or as an additional attribute. Of course the event semantics URI plays a central role in supporting the flexibility provided by the ENS in managing a variable set of event attributes. This semantics URI avoids having to hardwire within the subscriber applications specific knowledge, in order to adapt their behaviour to changing conditions (e.g., replacement of a measuring sensor with a different one able to measure the same physical quantity but having different measuring characteristics).

---

\(^6\) There can be situations in which the submission time is different from the actual event’s happening time. In these situations the publisher can add further attributes to the events to report this additional information.
5 Conclusions

This deliverable describes the application programming interfaces (APIs) that are available to developers and users of the technologies developed within the IoT@Work project. The APIs defined in the project are made available to address both (a) the design process of a manufacturing system (i.e., extending some already existing planning concepts); and (b) to allow interfacing to the physical system during the configuration transfer and (c) to allow some autonomy between key ICT functions while operating the manufacturing system.

The design and programming phases of applications include some planning of the devices and the purpose of their hosted embedded services. During configuration and operation of the application, the IoT@Work architecture focuses on the dependencies between the different management elements centred around providing (1) communication services, (2) security and integrity services, as well as (3) mapping application context and data purpose provided by embedded services to real IoT devices.

The IoT@Work-enabled device is defined in this document as container of key services that could be embedded in a real sensor, actuator or controller but also deployed in a proxy or gateway approach to interface to legacy or devices and things with limited resources. The architecture defines the functions needed to program and configure several IoT embedded services accessing or offering IoT resources to applications. This translates into the ability to access the device embedded intelligence through a given API to allow transfer the application context to the specific device.

Given the requirements of automated systems (i.e. several devices interacting with each other in an automatic manner as part of a control network), there are different ways to link a device to an application context, one where the device is carefully identified at the planning phase according to its function, capabilities and context, and therefore providing a specific role within the automated system. There exists also a range of applications that do not require the device as such but the embedded service or function only independently of the identity of a specific device. The difference between the two applications sums up the main differences between automation system needs, and IoT application needs. The latter are less tied to the physical system identities, although they could be defined around a flexible context. The former applications are programmable control systems, which we address as legacy applications. The control programming and engineering systems are out of the scope of this project, however, the assumption that some planning and design of device interactions and automated control systems require a much tighter context matching between the offered services and the planned identities of devices. Therefore, both application types need to be catered for in this project.

Section 2 of the document presented the overall architecture of the technologies from the perspective of an application and the ways that an application can interact with them through their publicly available APIs. It discussed tasks such as system pre-configuration, security pre-configuration and security mechanisms, application engineering and planning, and system bootstrapping and component dependencies. In each case along with the APIs themselves it discussed the underlying principles and assumptions so that it is easier for application developers to understand how to employ IoT@Work technologies productively.

Section 3 uses the architecture components presented in Section 2 to address a more concrete real-world scenario involving multiple stakeholders, devices, and applications.
Finally, Section 4 concentrated on ways to structure and employ higher level application management services, in order to increase the system robustness, safety and security without unnecessarily hampering its agility.
6 References


