E2District

Deliverable 4.1
District Heating Operation System Platform Architecture Specification

Dissemination Level: PU

This project is partially funded by the European Commission under the H2020 Framework Programme - Grant Agreement no. 696009

Project Number	:	696009
Project Title	:	Energy Efficient Optimised District Heating and Cooling – E²District
Deliverable Dissemination Level	:	PU

Deliverable Number	:	D4.1
Title of Deliverable	:	District Heating Operation System Platform Architecture Specification
Nature of Deliverable	:	R
Internal Document Number	:	E2D_D4.1_WP4
Contractual Delivery Date	:	M12-31/01/2017
Actual Delivery Date	:	20/03/2017
Work Package	:	WP4
Author(s)	:	Christian Beder
Total number of pages	:	23

Abstract

This document will describe the District Heating Operation System (DOS) Platform Architecture. The role of the DOS is to abstract the heterogeneous building sub-systems from the actual implementation and deployments within the buildings, and to allow the development of energy services on top.

The document will specify the DOS architecture and describe its individual components. Furthermore it will go into detail on how to access the DOS through its various APIs. It can be used as reference for implementing services running within the DOS. These services can be internal E2District services, as well as external third-party applications.

Finally it will describe the end-user web application that can be used to easily access the data inside the DOS.

Keyword list

District Heating Operation System, Architecture Specification

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

Document History

Date	Revision	Comment	Author/Editor	Affiliation
31/01/2017	0.1	Initial version	Christian Beder	CIT
3/3/2017		Feedback from the GA	all	all
8/3/2017	0.2	Incorporated feedback	Christian Beder	CIT
9/3/2017		Internal review	Martin Klepal	CIT
13/3/2017	0.3	Update according to feedback	Christian Beder	CIT
16/3/2017		Feedback from STM	Kostas Kouramas, Marcin Cychowski	UTRC
16/3/2017	0.4	Update according to feedback	Christian Beder	CIT

Executive Summary

This document will describe the District Heating Operation System (DOS) Platform Architecture based on the functional requirement specifications of deliverable D1.1. In summary, the role of the DOS is to abstract the heterogeneous building sub-systems from the actual implementation and deployments within the buildings, and to allow the development of energy services on top.

The document will specify the DOS architecture and describe its individual components. Furthermore it will go into detail on how to access the DOS through its various APIs. It can be used as reference for implementing services running within the DOS. These services can be internal E2District services to be developed with the work packages of the E2District project, as well as external third-party applications.

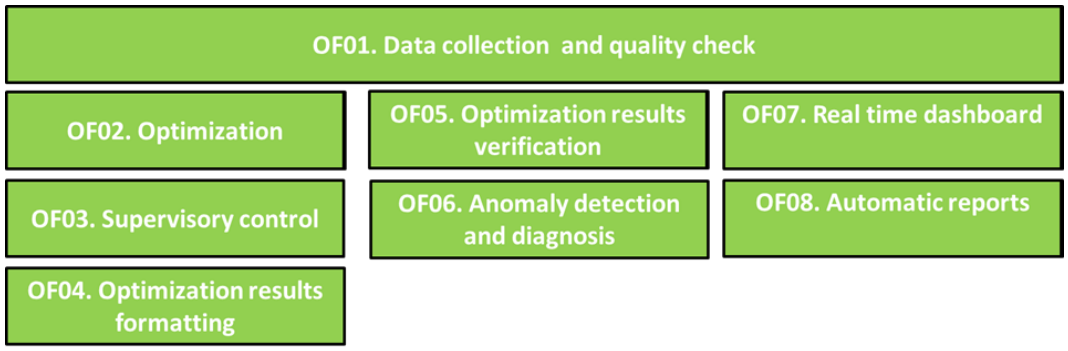
Finally it will describe the end-user web application that can be used to easily access the data inside the DOS.

Table of Contents

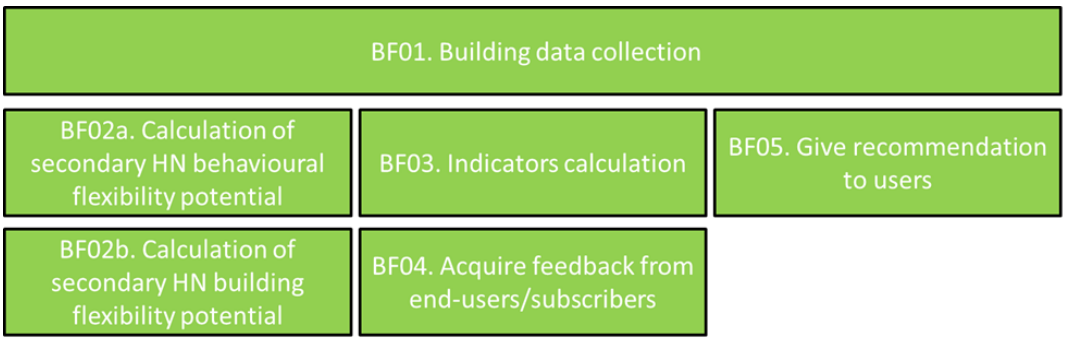
1 Introduction

A core vision of E2District is to create a district heating and cooling system operation system that abstracts the underlying embedded monitoring and control systems, facilitates the deployment of operational function blocks such as supervisory control, fault detection and diagnosis algorithms and the creation of energy management, prosumer engagement and visualisation services. Furthermore a public open API should enable system extensibility and integration with additional services and functions beyond the project.

The functional architecture derived in deliverable D1.1 for the operation of the primary heating network comprises the following functional blocks:



Further to that the functional blocks identified for the operation of a group of buildings heating and cooling network in deliverable D1.1 are the following:



The district operation system should therefore be designed such that it supports all these functions and enables the efficient implementation of the scenarios described in deliverable D1.1:

- Optimisation of the production scheduling
- Supervisory control of the network
- Anomaly diagnosis for the district heating and cooling network
- Secondary heating network behavioural flexibility
- Secondary heating network building flexibility

Grouping the functional blocks into logical modules, and connecting these modules according to the connections implied by these scenarios results in the overall E2District system architecture.

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

The architecture depicted in the following picture reflects this and shows the individual modules together with their mutual interconnections. The orange box delineates the scope of the district operation system, and the purpose of the district operation platform should be to support the implementation and interconnection of the modules contained therein.

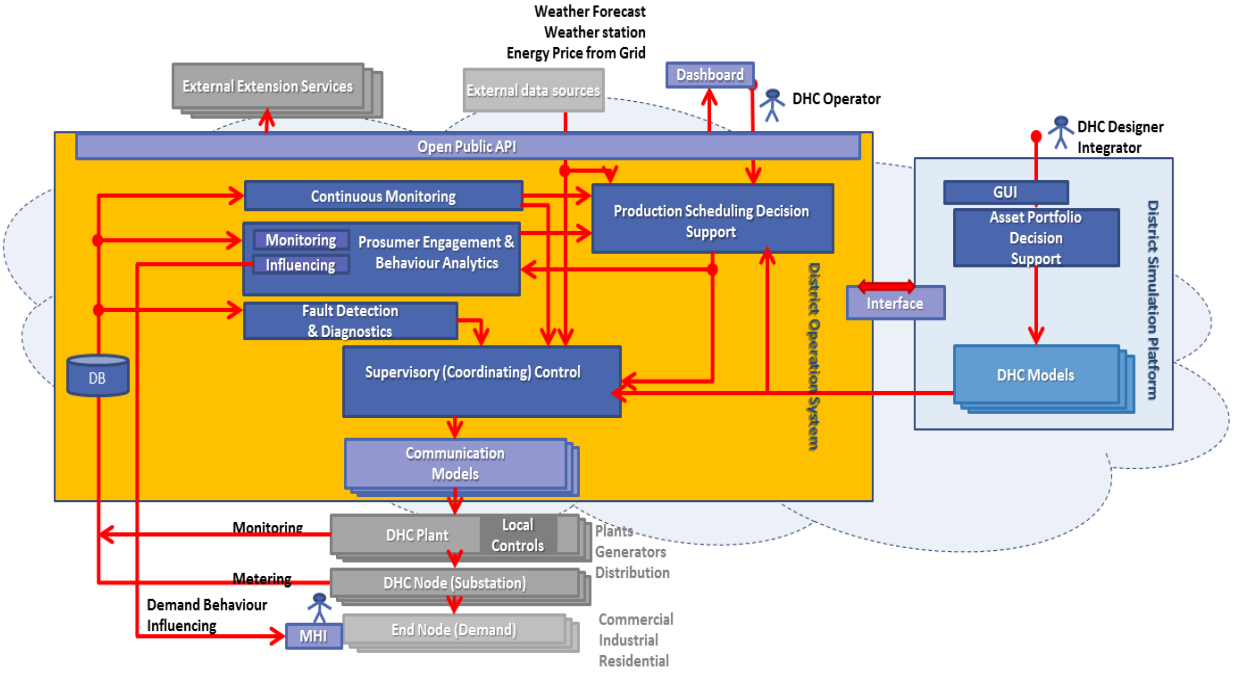


Figure 1: The District Operation System in the context of the E2District architecture (in orange)

Having an operation system cloud based platform will significantly ease the technical challenges associated with system integration of individual modules. The district operation system (DOS) serves three major purposes within the architecture above:

1. It abstracts the various heterogeneous building sub-systems from the building energy services, allowing them to be implemented and deployed without the need to interact with the different building subsystems directly
2. It provides an API for external services to connect to the building energy services and the building subsystems, allowing third-party energy services as well as end-user applications to be built on top of the E2District platform.
3. It provides internal communication and data storage facilities to allow E2District services to communicate with each other and access the building data in a unified way

In the following we will describe, how the cloud-based NiCore platform is adapted to enable this and how it can be accessed in order to implement services in the E2District platform.

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

2 Platform Architecture

In order to support the system architecture requirements outlined in the previous section we believe that an event-driven platform is best suited to this task. The architecture presented in the previous section comprises of the following functional modules relating to the district operation:

- Continuous monitoring
- Production Scheduling Decision Support
- Prosumer Engagement and Behaviour Analytics
- Fault Detection and diagnostics
- Supervisory Control

All of these are triggered by pre-cursory events and generate new events for subsequent processes as depicted in Figure 1, which allow for an effective implementation within an event-driven platform architecture (EDA). The benefit of implementing these services in an event-driven platform architecture is two-fold:

- It decouples the individual modules and allows them to be developed and deployed independent of each other and distributed across multiple machines
- It enables the development of modules that react to environment triggers and constraints in real time

This allows the E2District Operation Platform to be deployed at scale and to adapt to input triggers accordingly, scaling out into the cloud if necessary to facilitate seamless operation of large and growing districts.

Further to that the E2District system architecture presented above also has to enable the following auxiliary services during the district operation

- Public APIs
- Communication with the Buildings / Plants
- Interfaces to the District Simulation Platform

In order to achieve all these requirements, the NiCore framework, which is an event-driven application enablement platform developed by CIT, will be extended towards these goals. The NiCore platform architecture will look like this:

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

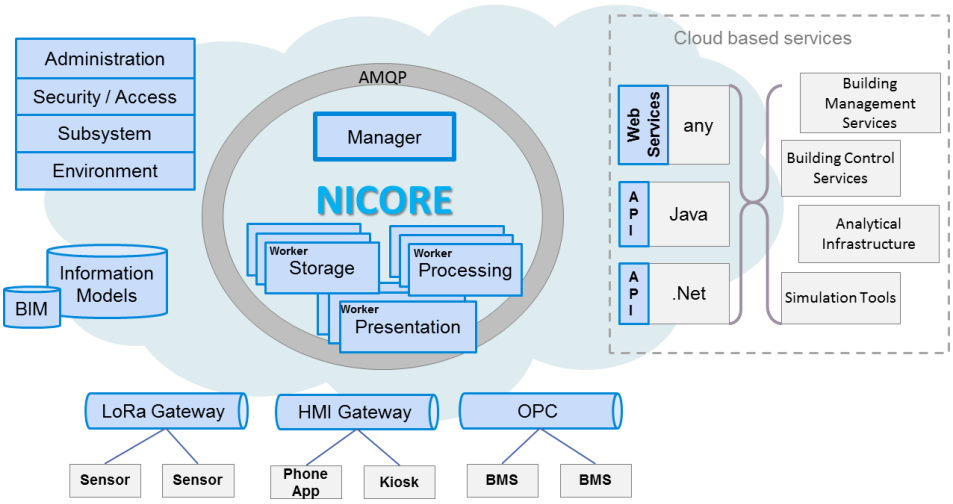


Figure 2 The E2District DOS architecture based on the event-driven NiCore framework.

At its core is an AMQL message broker middleware, facilitating all communication with in the platform and enabling the platform scalability. The platform will allow for the execution of worker services, allowing the functional modules of the E2District architecture to be run inside and being managed by a platform manager process. It also will provide access to the underlying building and sensor subsystems, and provide APIs to be used by the district simulation platform as well as opened for external service developments.

In summary, the framework supports a set of functionalities that constitute the reference architecture for the E2District DOS. Of particular interest are the following features, which will be described in the remainder of this document:

- The abstraction of various subsystems
- A message broker based middleware
- Event-driven service provisioning
- Data storage
- A cloud based services API

2.1 Message broker based middleware

At the core of the E2District DOS is a message broker based middleware that allows the individual modules to communicate and exchange data. In this architecture a centralised message broker facilitates the communication between decentralised modules using a publish/subscribe pattern.

From a module point of view there are a couple of aspects to consider depending if it wants to transmit data to or receive data from other modules:

- The transmitting module needs to define
    - the data structure it wants to transmit and its serialisation
    - the exchange name it wants to transmit the data to and its type

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

- The receiving module needs to know
    - the exchanges it wants to receive data from
    - the data structure and its serialisation as defined by the transmitter in order to be able to de-serialise and work with the data

It is important to note that the structure of the data and its serialisation is up to the modules to mutually agree on and remains transparent to the communication infrastructure. This enables the platform to be augmented and extended with new modules if necessary without the need to alter the platform and without imposing restrictions at the design stage.

The process of communication is then as follows:

- every participating module connects to the message broker and subscribes to the message exchanges it wants to receive data from; the message broker creates a message queue for each receiver
- the transmitter creates and serialises the data it wants to transmit
- it the transmitter then connects to the message broker and publishes the serialised message to the exchange
- the message broker then routes the message to all subscribing receivers via the queues created on subscription
- on receiving a message from the message broker the receiving module de-serialises its content and further processes the received data

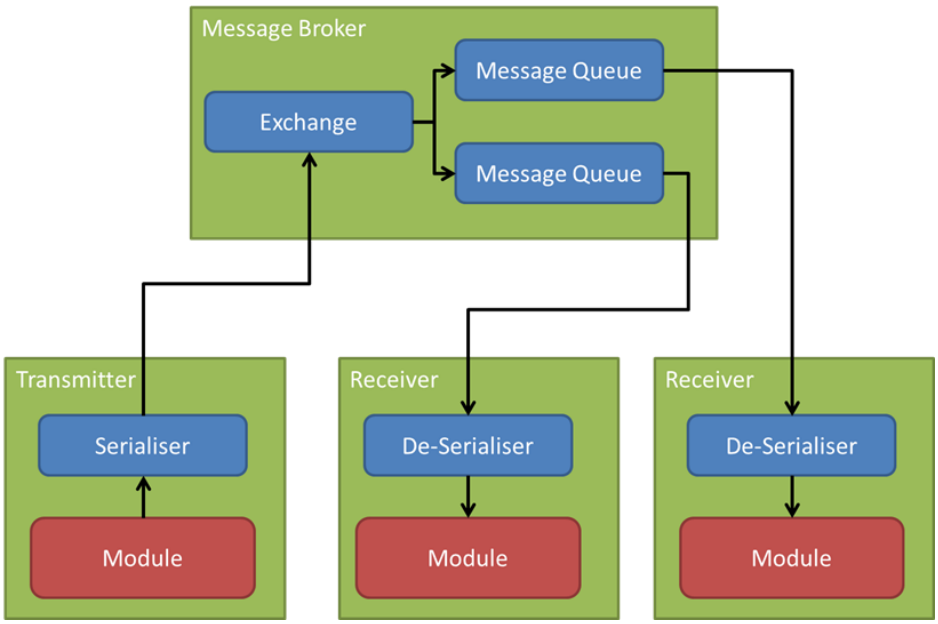


Figure 3 Data flow between the transmitting module, data broker, and receiving modules

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

The message broker decouples the transmitter from the receivers, so that no direct physical connection between them is necessary for the communication. Every module only needs to make sure that it is connected to the message broker, which on receiving a message from a transmitter stores it internally and routes it to the subscriber's queues for processing when the receiver is ready.

In a complex scenario every module can be both transmitter and receiver and multiple receivers can receive messages from a single transmitter. Furthermore exchanges can be pre-defined or created ad-hoc by the individual modules enabling a number of communication patterns which we will describe later in this section.

2.1.1 Serialisation

First we will look at the issue of serialisation for which there are multiple options. As the payload of the messages is transparent to the message broker it is up to the communicating modules to mutually decide and it is well possible to have a heterogeneous system where different objects are transferred between different modules in different formats. The key point is that whenever two modules communicate with each other through the E2District DOS they need to mutually agree on the method used. We will propose a couple of options and discuss their pros and cons:

- Whenever two modules are developed using the same platform (e.g. .NET or Java) they can use the build in platform specific class serialisers. This option is extremely convenient and easy and enables the transmission of very complex data structures but it requires the two modules to be very closely linked together during the development process.

- Crossing platform boundaries requires more serialisation effort on both ends as data types don't automatically match between transmitter and receiver. Many options are available depending on the platforms that need to be bridged. Very efficient and popular are Google's Protocol Buffers that support a wide variety of different platforms and produce very efficient and small binary data packets.

- If the previous two options are not feasible text-based human readable formats like JSON or XML can be used. The advantage is that those are highly flexible and supported on almost any platform, however because they are non-binary formats they are extremely verbose and inefficient to process and transmit.

As discussed before it is up to the individual modules to mutually agree on a common format and serialiser taking the above considerations into account. For the purpose of communicating the subsystem data the following JSON format will be used for example:

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

```
{
        "$type": "NimbusCR.Core.Interfaces.ServiceRequestMessage, NimbusCR.Core.Interfaces",
        "MessageType": "OPCRawMeasurement",
        "Parameter": {
                "$type": "System.Object[], mscorlib",
                "$values": ["2015-08-24T11:07:09.5680361Z",
                "EnergyINTIME",
                "FaroBMS",
                "TempRetPrimAQ",
                22.3]
        }
}
```


2.1.2 Messaging Patterns

Every module can be both transmitter and receiver and exchanges can be either pre-defined or defined ad-hoc by every module. Furthermore different routing schemes can be applied allowing for a variety of messaging patterns of which we will discuss some.

The first pattern is a simple broadcast. In this scenario a transmitter keeps on broadcasting messages to a predefined fanout exchange. Every subscriber will receive a copy of the message and can work on it. An example is the constant stream of sensor data from the data acquisition module. Whenever a new sensor reading becomes available it will be broadcast to a predefined exchange and modules can subscribe to this stream to trigger processing events

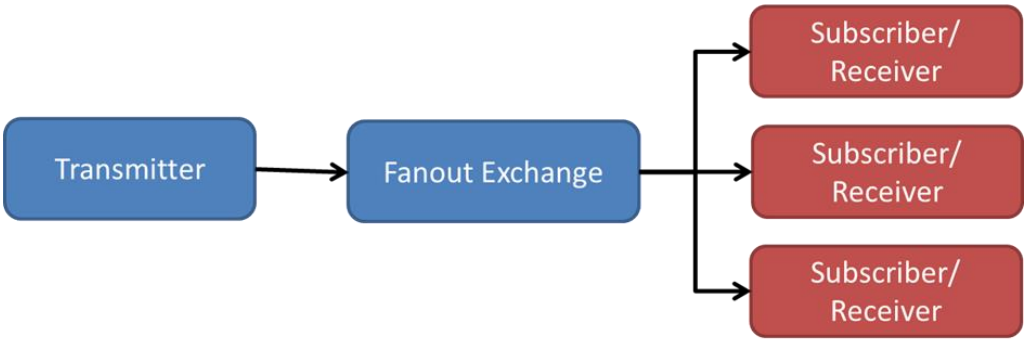


Figure 4 Broadcast messaging pattern. The message is transmitted to a fanout exchange which delivers it it all subscribers.

A second useful messaging pattern is not to deliver a copy of the message to every subscriber but only to a single receiver amongst the potentially multiple subscribers. This is useful for load balancing and ensuring that data is processed by some entity within the platform.

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

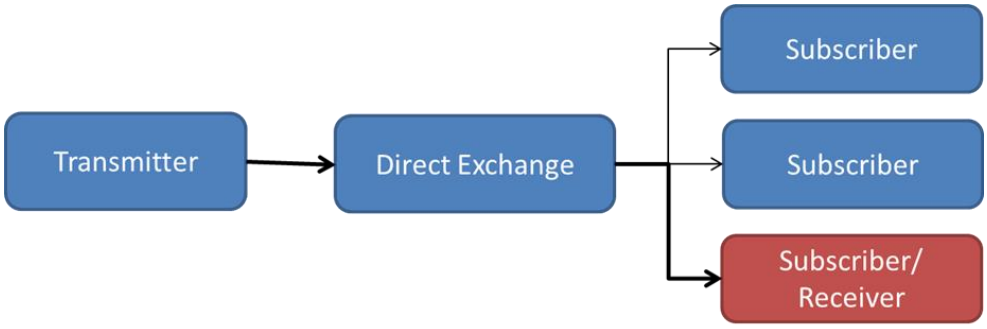


Figure 5 Direct exchange pattern. The message is transmitted to a direct exchange which selects exactly one of the subscribers to receive it.

The previous pattern can also be extended to facilitate communication back to the original transmitter thereby implementing remote procedure calling between the modules. To achieve this the transmitter declares and subscribes to a temporary exchange and communicates the name of this to the receiver via the message sent. The receiver then processes any data and communicates the result back over the answer exchange. As before this is useful for offloading some of the load from the transmitter module to other entities within the platform thereby enabling scalability. It is also a very useful pattern for the implementation of frontend GUIs that need to interrogate the platform for data to present to the users.

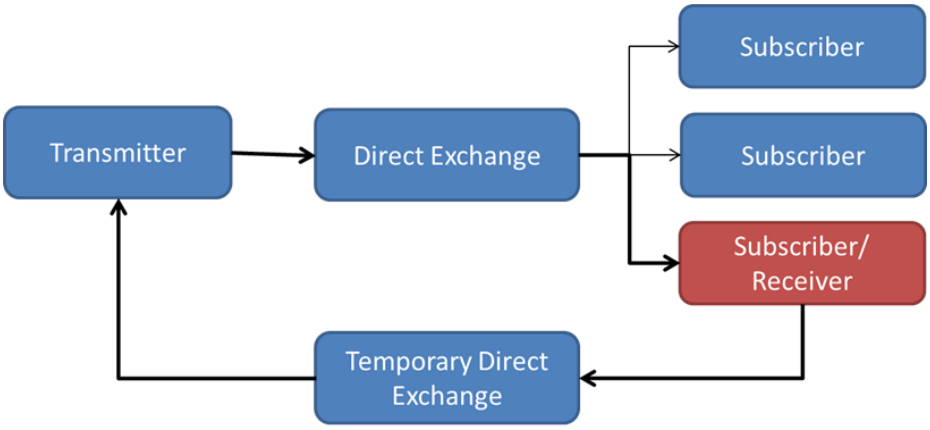


Figure 6 Remote procedure call pattern. The message contains information for the receiver to send back an answer to the original transmitter.

The implementation of this communication infrastructure in E2District we will be using RabbitMQ as message broker which implements the AMQP standard.

2.2 Event-driven service provisioning

Event driven data processing services reside inside the E2District Operation System. These services are triggered whenever a certain type of processing is necessary and can be requested by the connector on receiving new data from the building management

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

system if required. Examples of such services in E2District include logging incoming data to the database or triggering a new building simulation iteration.

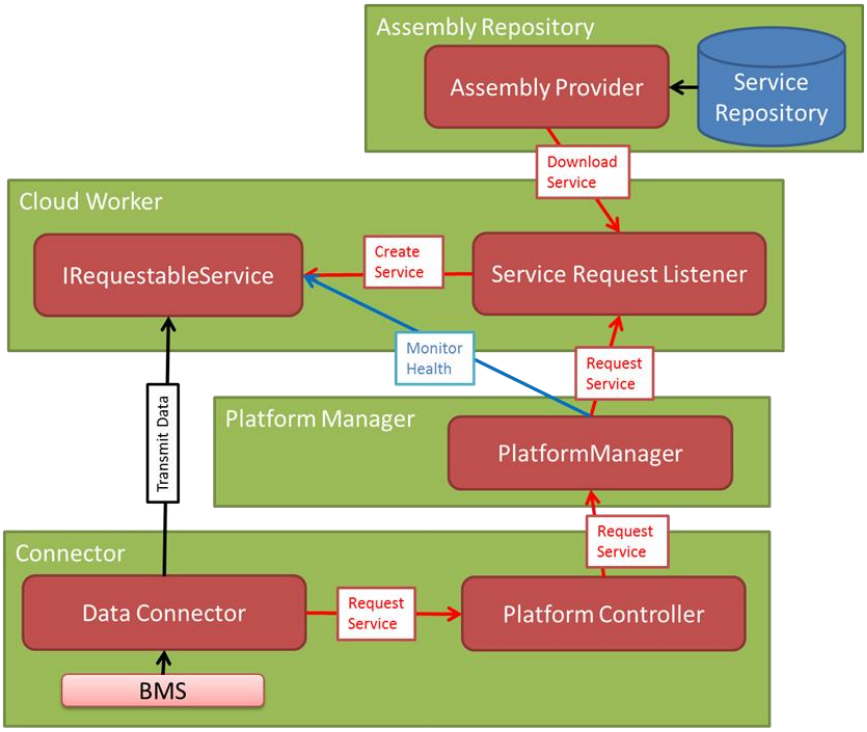


Figure 7 Data driven service provisioning. Incoming data triggers the creation of a worker service if necessary.

Whenever the data connector receives new data from the building management system it checks with the platform controller if there is already a processing entity available for the type of data received. If this is the case the data is transmitted directly to this entity through an exchange provided by the platform controller.

In case this data processor does not already exist, the platform controller asks the platform manager to provide such. The platform manager then identifies a suitable worker with sufficient available resources and requests the instantiation of the required data processing service. A service request listener resides within each worker and answers this requests by downloading the required service from a service repository if necessary and instantiating it locally stating to listen for incoming data. The exchanges the newly instantiated worker service is listening to are reported back to the platform manager and platform controller, so that the connector can now start transmitting its data directly to the worker service.

The platform manager continuously monitors the health of each service so that services can be re-provisioned in case of a fault.

Every entity within the platform, i.e. not only the connectors, can request services so that processing can be cascaded if necessary. Furthermore new services can be uploaded to the service repository allowing dynamic updates of the processing pipeline without having to shut down the entire system.

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

2.3 Subsystem abstraction

The goal of this function of the E2District DOS platform is the ability to connect a multitude of heterogeneous sensing and actuation sub systems and making their data available to the platform and its services in a unified way.

Connectors to vendor specific APIs need to be developed, which can be plugged into the existing infrastructure and communicate the data to the cloud based platform for further processing. In there it can be stored, processed, or forwarded to the modules and components that subscribe for the data without the need for the E2District modules to be aware of the specific implementations of the underlying infrastructure sub systems.

2.3.1 OPC

The de facto industry standard for interfacing SCADA systems is OPC and most system vendors offer OPC servers to interface their systems. While OPC offers a broad range of interfacing options most common is still the OPC DA 2.0 standard which is a DCOM based client-server architecture that allows third party client software to read and modify certain values (called tags) inside the building management systems.

OPC is based on Microsoft's Component Object Model (COM) and allows the client to interface with the OPC libraries on the server machine and exchange data in a standardized way. In case of OPC the client opens a connection to the server and receives a list of tags to subscribe for. A call-back function can then be registered with the server for these tags and the client is notified in case any value is changed in the OPC server by the BMS. In the other direction the client can also change the values of tags, which are then communicated back to the BMS by the OPC server for instance in order to change a set point.

Accessing the data through OPC DA 2.0 via COM is still widely used by building management systems and seems to be still the de-facto industry standard despite this mechanism now being almost 20 years old. As a consequence there are a lot of security concerns and modern internet based communication seems to be more appropriate where DCOM solutions for instance struggle to penetrate modern firewalls. Some building management system vendors are therefore now also offering web service based access to the data, however standardisation is lacking behind and despite some efforts by the OPC foundation this is not widely supported, yet, and currently most available web APIs are proprietary. Therefore, unlike in the case of OPC, dedicated connectors will need to be implemented for individual system vendors where required.

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

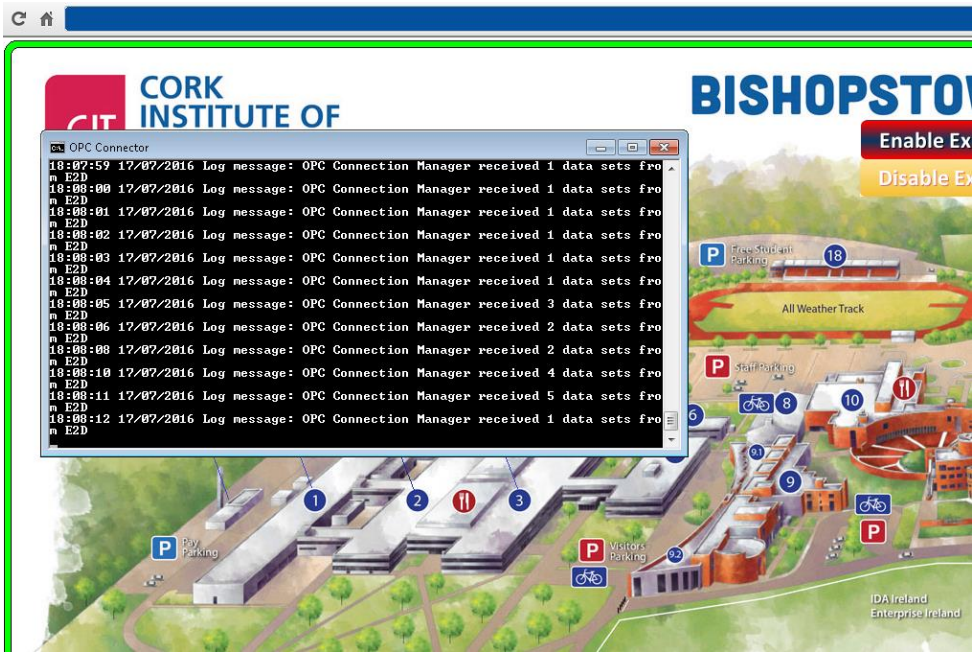


Figure 8 The OPC connector installed on the E2District pilot district SCADA in Cork.

The OPC standard specifies individual items as follows (in C# notation):

```
public class ItemValue : ItemIdentifier
{
    public Quality Quality { get; set; }
    public bool QualitySpecified { get; set; }
    public DateTime Timestamp { get; set; }
    public bool TimestampSpecified { get; set; }
    public object Value { get; set; }
}

public class ItemIdentifier : ICloneable
{
    public object ClientHandle { get; set; }
    public string ItemName { get; set; }
    public string ItemPath { get; set; }
    public string Key { get; }
    public object ServerHandle { get; set; }
}
```

We will follow this standard in the E2District DOS, aggregating or omitting unnecessary fields, and add a unique GUID specifying the E2District connector from

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

which the data item originated, which can be used to identify the building and building management system. Every data item therefore comprises of the following information:

- A unique server identifier filled in by the connector
- A timestamp filled in by the data aggregator in order to avoid problems with synchronization
- A server unique item name taken from the OPC item name and path
- An item value taken from the OPC item value

Items are assumed to keep their value until changed resulting in a new item to be transmitted and stored in the database. E2District modules can access the data through the communication middleware either by subscribing to a message stream or by querying a web server, both of which will be described in more detail below.

2.3.2 HMI

The E2District solution tries to achieve additional energy savings through communicating with energy prosumers. To that end, prosumer flexibility will be made available to the energy optimisation process by the Prosumer Engagement modules as presented in the system architecture in Figure 1. In order to achieve this flexibility, appropriate messages have to be communicated to the energy prosumers through appropriate human machine interfaces (HMI). While the optimisation process is to be implemented inside a worker process, the District Operation system platform has to support the communication to the system end-users similar to other control subsystems available inside the district.

In order to facilitate this the District Operation System will enable the connection of mobile and stationary human machine interface devices and will allow the Prosumer Engagement module to communicate with the system end-users through these.

Figure 9 Human Machine Interface (HMI) devices to be used in E2District

The NiCore framework enables the development of applications for Android/iOS based smartphones, as well as large touch screen devices in order to communicate with the participants. Messages from the server can be transmitted to the DOS middleware,

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

enabling the direct engagement for E2District modules with the building occupants. This influencing of building occupants is expected to provide extra flexibility to the energy optimisation process that can be utilised as any other building asset.

2.3.3 LoRa

In order to supplement the sensing capabilities of the fixed installations deployed to the buildings, like the building and asset management systems interfaced through the OPC interfaces as described above, additional IoT sensors may need to be deployed.

Figure 10 A LoRa gateway installed on the roof of the Cork pilot site.

On the Cork pilot site we will be using LoRa based sensors for measuring the state of the building and assets, in particular to support and calibrate the prosumer engagement optimisation.

The advantage of these battery powered devices is the ease of deployment and the low maintenance costs of this infrastructure. As with all other sub systems connected to the E2District DOS, the IoT sensor readings will be available to all subscribing modules of the E2District architecture.

2.4 Data Storage

The NiCore platform also provides database storage capabilities. In order to facilitate a wide range of applications, both a rational database (Postres) as well as a NoSQL database (MongoDB) are supported.

In particular for the storage of the sensory data collected from the various subsystems deployed across the district MongoDB is the method of choice, due to the amount of data collected as well as due to the constant flow of new incoming data sets that have to be written to the database.

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

The E2District DOS stores all incoming data from the connected subsystems in a MongoDB key value store, together with the timestamps taken when the new value is written to the database. In order to facilitate efficient retrieval, the database of raw incoming data is indexed over these timestamps as well as over the sensor names and subsystems. This enables the API calls to access historical data efficiently, while at the same time allowing the database to be constantly appended by new data records coming in from the connected sub-systems.

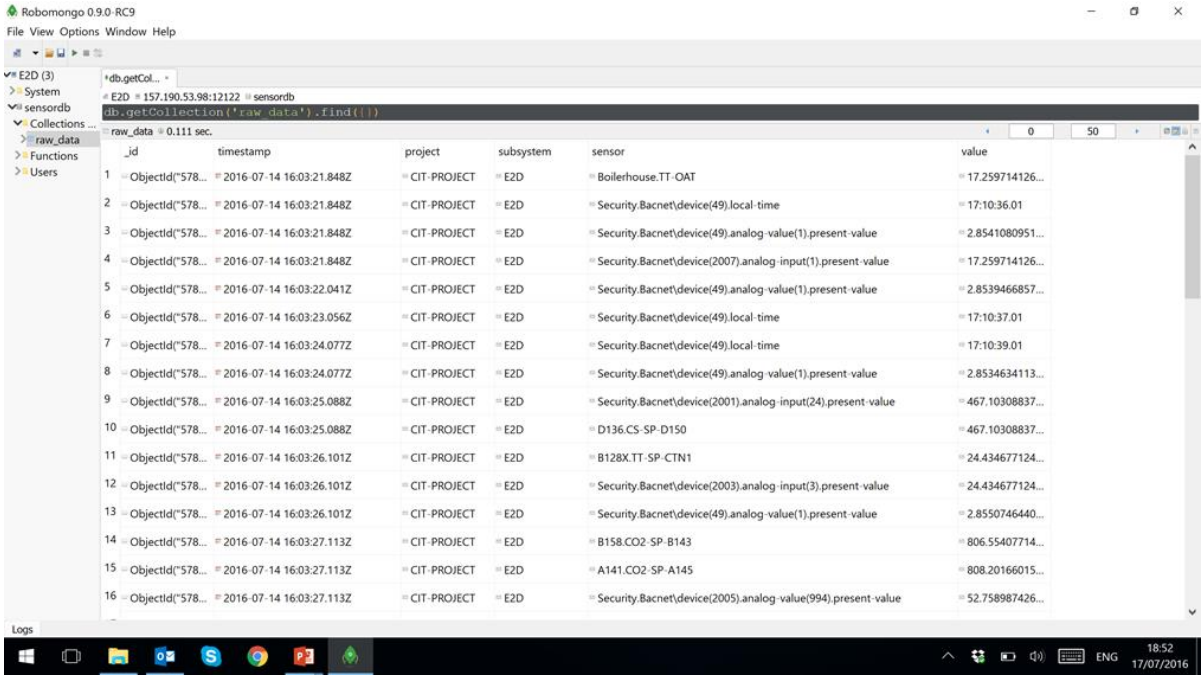


Figure 11 A screenshot of the MongoDB database for the E2District pilot site in Cork

2.5 Cloud based services API

In order to access the data and services of the District Operation System, the NiCore platform provides a range of application programming interfaces (APIs). These interfaces allow the implementation of services that interact with other services of the E2District platform and provide access to the data stored within the platform.

All these APIs are cloud-based, and can therefore be executed on any host provided a connection to the NiCore service hosts is presents. This architecture allows for distributed development and distributed deployment of the E2District solution.

2.5.1 .NET / C#

The E2District reference platform itself has been developed in C#, therefore the most powerful API to access the functions of the District Operation System are available in .NET. At the core of this API are the access functions to the RabbitMQ message broker:

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

```
namespace NimbusCR.Core.Interfaces
{
    public interface IMessageReceiver
    {
        void SubscribeToBroadcasts(string exchangeName, ICastableService service, string exchangeType);

        void SubscribeToQueries(string exchangeName, IQueryableService service);

        void UnSubscribeFromBroadcasts(string exchangeName, ICastableService service);

        void UnSubscribeFromQueries(string exchangeName, IQueryableService service);

        void UnsubscribeFromEverything();
    }
}
```

These functions allow to subscribe to messages passing through the E2District platform and enable the implementation of services that react to such messages. The corresponding API functions for transmitting data are as follows:

```
public interface IMessageTransmitter
   {
       object Query(string exchangeName, string messageType, object parameter, MessageDataType datatype, int timeout);

       void Broadcast(string exchangeName, string messageType, object parameter, MessageDataType datatype, string exchangeType);
   }
```

These two interfaces allow access to all services running within the District Operation System. In order to access certain function in a more convenient way, specific gateways are available. For instance the gateway for accessing functions relating to the sensor data collected within the district is the following:

```
public interface ISensorsGateway
{
    bool StoreInputDatas(string _project_id, List<RawSensorData> _datas);

    bool StoreInputData(string _project_id, ulong _timestamp, string _subsystemName, string _subsystemType, string _sensorName,
string _dataValue, string _datatype);

    Dictionary<Sensor, List<RawSensorData>> GetRawData(string project_id, List<int> _subsystems_dbids, List<int>
_sensorDevice_dbids, List<int> _sensors_db_ids, bool _onlylatest, DateTime _from, DateTime _to, int _limit);

    Dictionary<long, List<int>> GetDataHistogram(string project_id, List<int> _sensors_db_ids, DateTime _from, DateTime _to, string
_step);

    List<RawSensorData> GetSensorRawData(string project_id, string _subsystem_name, string _sensor_name, bool _onlylatest,
DateTime _from, DateTime _to, int _limit);

    DateTime GetLastDataTimeStamp(string _project_id, string _subsystem_name);

    Dictionary<string, SensorSubsystem> GetAllSensorSubsystems(string project_id);

    List<SensorSubsystem> GetSubsystems(string project_id);

    int UpdateSensorSubsystem(string project_id, SensorSubsystem sensorSubsystem);

    List<SensorDevice> GetSensorDevices(string project_id, List<int> _subsystem_dbids, List<int> _environment_sensors_dbids);

    int UpdateSensorDevice(string project_id, SensorDevice sensorDevice);

    List<Sensor> GetSensors(string project_id, List<int> _subsystem_dbids, List<int> _sensorDevice_dbids, List<int>
_sensorType_dbids, List<int> _db_ids);

    List<Sensor> GetSensorsbyName(string project_id, List<string> names);

    List<Sensor> GetSensorsbySubsystem(string project_id, string _subsystem_name);

    int UpdateSensor(string project_id, Sensor sensor);

    List<SensorDataType> GetSensorDataTypes(string project_id, List<int> _db_ids);

    int UpdateSensorDataType(string project_id, SensorDataType _sensorDataType);

    List<SensorType> GetSensorTypes(string project_id, List<string> _names, List<int> _db_ids);
```


This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

```
    int UpdateSensorType(string project_id, SensorType _sensorType);
    List<SensorDisplay> GetSensorDisplay(string project_id, List<int> _sensor_type_db_ids);
    int UpdateSensorDisplay(string project_id, SensorDisplay _sensorDisplay);
    List<SensorAnimation> GetSensorAnimation(string project_id, List<int> _sensor_type_db_ids);
    int UpdateSensorAnimation(string project_id, SensorAnimation _sensorAnimation);
    bool SetSetPoints(SortedDictionary<DateTime, List<OPCData>> _opcdata, bool _setOldtoo);
    bool SetSetPointDirectly(string project_id, string _subSystemName, string _sensorName, string _dataValue, string _dataType);
    List<SensorDataFile> GetSensorDataFiles(string project_id);
    List<SensorDataFile> QuerySensorDataFile(string project_id, string subsystem, string sensor_name, long from, long to);
    bool InsertSensorDataFile(string project_id, string file_name, string subsystem, string sensor_name, long from, long to);
    bool CleanSensorDataFile(string project_id);
}
```

2.5.2 Java

Not all services and applications running within the District Operation System need to be developed in .NET. A subset of the API functions are available on other platforms as well. The following API calls are available for Java based services:

```
public void SubscribeToRawData(ICastableService service)

public void UnSubscribeFromRawData(ICastableService service)

public Object GetSubsystems(String project_id)

public Object GetSensors(String project_id, String subsystem_name)

public Object GetSensorHistoricData(String project_id, String subsystem_name,
String sensor_name, long from, long to, int limit)

public Object GetSensorCurrentState (String project_id, String subsystem_name,
String sensor_name)

protected Object GetSensorRawData_(String project_id, String subsystem_id, String
sensor_id, boolean onlylatest, long from, long to, int limit)

public Object SetSetPointDirectly(String project_id, String subsystem_name,
String sensor_name, String data_value, String data_type)
```

2.5.3 MATLAB

The Java functions described above can also be called in MATLAB. The following list of MATLAB API functions is available for accessing the sensor data and actuators in the district:

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

function client = init_nicore(server, port, username, password)

function subsystems = get_nicore_subsystems(client, project)

function sensors = get_nicore_sensors(client, project, subsystem)

function [value,timestamp] = get_nicore_sensor_state(client, project, subsystem, sensor)

function [values,timestamps] = get_nicore_sensor_history(client, project, subsystem, sensor, from, to, limit)

function success = set_nicore_setpoint_directly(client, project, subsystem_name, sensor_name, data_value, data_type)

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

3 Web Application

To demonstrate the capabilities of the open APIs for accessing the district operation system a web application has been developed. It is also useful to easily access the available data within the District Operation System. While the API calls described in the previous section are intended to be used by automated services operating on the DOS, the web interface is intended to give human operators easy access to the available data and to provide them with simple download and visualisation facilities.

The web application supports user authentication in order to restrict access to the district data to authorised persons only. Data can be visualised for subsets of individual sensors, or combined monthly datasets may be downloaded from the servers.

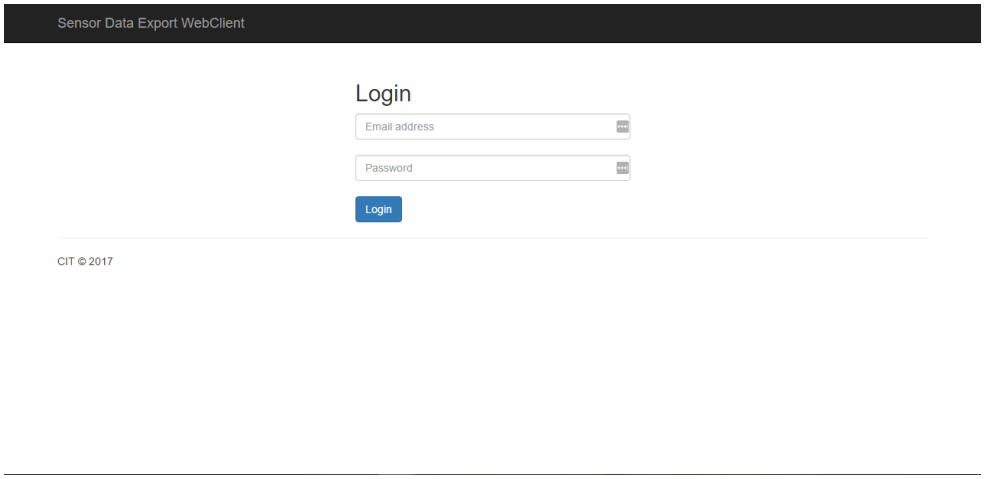


Figure 12 Login screen of the web application

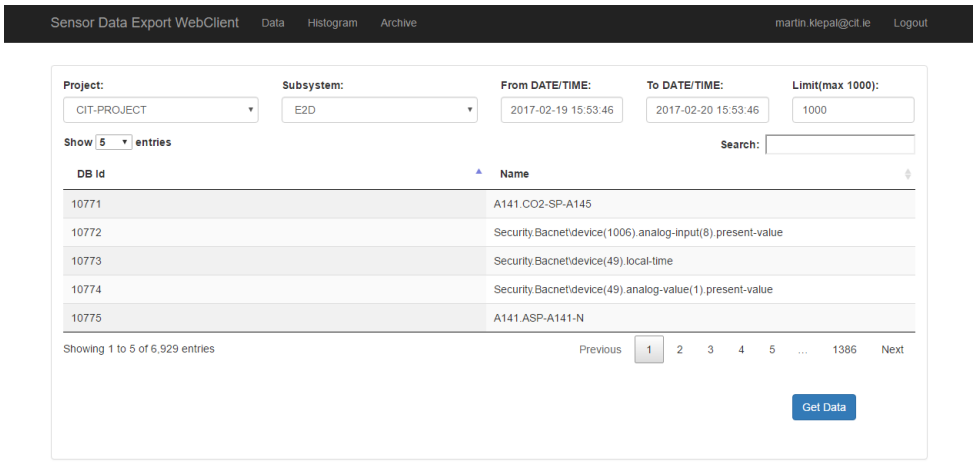


Figure 13 Data selection screen of the web application.

The web application is based on the platform APIs and will be extended towards the evolving requirements during the project.

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*

4 Conclusion

This deliverable defined the District Operation System for supporting the functional requirements of the functions defined in deliverable D1.1 with regards to the operation of the primary heating and cooling network as well as with regards to the operation of individual groups of buildings. It was shown how the operational scenarios described in D1.1 are translated into a system architecture comprising logical modules and their interactions. It was argued how an event-driven platform architecture would best suit the requirements of the E2District Operation System.

The document describes how the overall E2District architecture and its modules can be implemented using an extension of the existing NiCore platform framework. This framework will be used to facilitate a reference implementation of this architecture in a cloud based district operation system platform.

Furthermore the document provides a reference for the various APIs available to module developers, both external as well as internal, and shows how building and district heating and cooling sub-systems are abstracted through the platform to allow a unified access to data and controllable assets.

This project has received funding from the *European Union's Horizon 2020 research and innovation programme under grant agreement No 696009*