Service Interfaces for Field-Level Home and Building Automation

W. Kastner and G. Neugschwandtner
Institute of Computer Aided Automation
Technische Universität Wien
{k,gneugsch}@auto.tuwien.ac.at

Abstract

Gateway functionality is a key point in leveraging the potential of a home or building automation network. High-level interfaces providing abstractions of field area network (FAN) functionality significantly reduce the effort involved in creating application-level solutions. Such interfaces are presented on two different levels of abstraction. First, an interface corresponding to the Network layer of a FAN used in building automation (European Installation Bus, EIB) is shown, which unifies handling of the various physical media and bus access hardware available. Second, an approach with the goal of providing a generic representation of narrow-band building control functionality independent of a specific FAN technology is proposed. The interfaces are designed to be used by lightweight components of an OSGi (Open Services Gateway Initiative) platform, which is introduced as well.

1. Introduction

Field area networks (FANs) provide a way to significantly enhance the functionality of electrical installations in buildings. Here, most devices – like wall switches or light dimmers – relatively infrequently exchange small amounts of data only. Field bus technology is optimally suited to enable the flexible exchange of this *control* information.

Even if considerable improvements can be achieved through the use of a FAN alone, integrating it with other networks offers further evident benefits. Especially in the field of home automation, the additional possibilities arising with the use of gateways may even be attractive enough to tip the balance in favour of opting for “smart wiring” in the first place. Once control data are present in digital form, it seems natural to extend monitoring and control beyond the doorstep. Beyond FAN/WAN (wide area network) interaction, worthwhile integration perspectives exist within the boundaries of the living area as well.

While numerous gateway solutions exist, most of those suited for the residential application domain offer a fixed set of capabilities only. The integration of novel functionality frequently requires to start development from

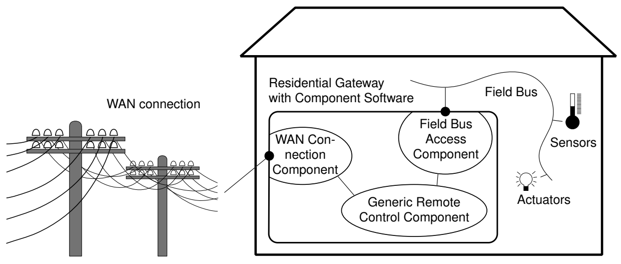

WAN connection
Residential Gateway with Component Software
Field Bus
WAN Connection Component
Field Bus Access Component
Generic Remote Control Component
Sensors
Actuators

Figure 1. Residential gateway using software components

scratch. A modular gateway design, proposed for example in [13], can alleviate this problem. It reduces the effort involved in creating new applications by allowing them to make use of high-level abstractions of field and wide area network functionality. The application of this concept to a residential gateway is illustrated in Figure 1.

Re-use of modules can be furthered by designing them as *software components.* This involves packaging them in a way suitable for independent deployment and third-party composition [2, 15]. Ideally, complete applications would be assembled by customising and connecting components without coding effort in the traditional sense. If supported by the execution platform, this binding may even take place at run-time. This opens the door on *service-based software models,* where services are flexibly configured to meet a specific set of requirements and are discarded when no longer needed [1].

Considering suitable component frameworks, the *Open Services Gateway Initiative (OSGi)* offers an interesting platform specification. It defines a dynamic environment for software components allowing the run-time configuration of services. By including a powerful concept for remote management, it goes a significant step towards supporting *remote services* [9]. Although designed with gateway applications in mind, the framework design is generic enough not to preclude other use. The OSGi framework is based on Java, allowing components to be used on various hardware platforms. Still, it has a small enough footprint to be useful in embedded environments.

To allow prefabricated components to interact properly, they have to provide and use common interfaces. The more components implement a specific interface, the

0-7803-8734-1/04/$20.00 ©2004 IEEE.

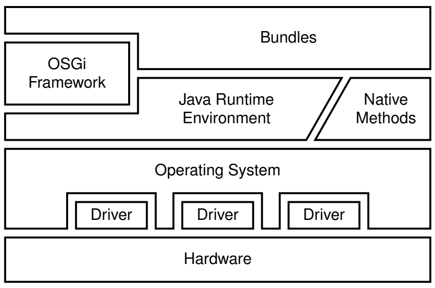


Figure 2. OSGi architectural overview

wider becomes the choice of modules available for mixing and matching (within the context of gateway applications, for example, network interfaces and application logic). Thus, such interfaces should cover a broad range of applications.

A popular approach towards such an interface suitable for the representation of arbitrary field bus functionality is to break the latter down to the data point level. One widespread component-based implementation utilising this concept is OPC (Open Process Control). Being aimed at large-scale automation systems, however, OPC is ill-suited for use in embedded environments like a residential gateway.

This paper proposes to use a hierarchy of interface abstractions for accessing FAN devices from within the context of an OSGi platform. *EIB (European Installation Bus)* is considered as a specific example. After an introduction to OSGi and EIB, an overview of the abstraction layers considered useful is given. Of these, the low-level interface chosen for EIB is covered in slightly more detail. Finally, the proposed high-level data representation, which aims to provide a technology independent view of control functionality in homes and small buildings, is treated in depth.

2. OSGi service platform

The Open Services Gateway Initiative is a non-profit technology consortium formed in February 1999. The open service platform specification maintained by OSGi is now available in its third edition [11]. The platform is designed to support a dynamic collection of services with all management aspects available through a communication channel free of choice. Although able to handle the complexity of an operator based network with multiple independent, external service providers, OSGi-based solutions are expressly not limited to this usage model. Arbitrary applications are possible, including entirely local ones where control rests with the end user and scenarios where no IP-based network is involved.[1]

The core of the OSGi specifications is formed by a service-oriented lightweight component framework, which allows to manage multiple applications in a single Java virtual machine. These execute concurrently, providing their functionality both to one another and to the outside environment. Service applications can be added, removed and updated at run-time, allowing continuous operation of the platform. The framework proper only provides the most basic management functionality, additional capabilities are added as standard service APIs.

[1]The most striking example is the OSGi framework recently being chosen as the plug-in environment for the Eclipse tool platform.

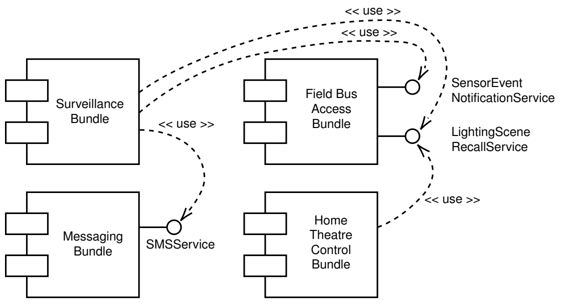


Figure 3. OSGi component interaction

2.1. Framework architecture

Figure 2 illustrates the relationship of the elements present within an OSGi service platform. Service applications, which are referred to as *bundles*, can access functionality provided by the framework, the underlying Java VM, and the operating system (by using native code libraries) as needed. Bundles represent the packaging and delivery unit within the OSGi framework. Only a single instance of a bundle exists at any given time. Bundles also provide hooks for switching between an active (usually including the execution of independent threads) and passive state. Consequently, they can be installed, started, stopped, updated and uninstalled.

Bundles collaborate by mutually providing and using *services*. A bundle can provide any number of services (including none), and services can in turn be used by any number of bundles. In Figure 3[2], the Surveillance bundle uses the Sensor Event Notification service of the Field Bus Access bundle to receive alerts from a motion sensor. In case the surveillance system is armed, it will use the Lighting Scene Recall service to turn on all the lights in case an intruder is detected and notify the owner using the SMS service provided by the Messaging bundle. Entirely independent of these actions, the Home Theatre Control bundle can recall the proper lighting scene for watching a film if needed.

Within the context of OSGi, the term “service” in the narrower sense refers to a Java interface definition with agreed-upon semantics.[3] A bundle using a service will only import the interface class. An appropriate implementation (called a *service object*) can be selected at run-

[2]It should be noted that although bundles are shown as UML *components* in Figure 3, OSGi services and bundles both contribute to the aspects usually associated with a component. Functionality is discovered and combined on the service level, while bundles represent the unit of composition for the gateway operator. This issue is detailed in [2].

[3]This should not be confused with the general, more high-level notion of a “service” as some added value presented to the end user.

time, potentially from multiple available choices. This approach de-couples specification and implementation of a service and allows programmers to bind to its specification only. It is important to note that the bindings between bundles providing and using services are established (and removed) entirely at run-time.

Service discovery is provided through a registry maintained by the framework. Here, bundles can register any number of services. Since in many cases, possibly different bundles will register multiple service objects implementing the same service, a set of descriptive properties (key/value pairs) may also be recorded along with the registration. Other bundles interested in a specific service can then look it up using the interface name and, if needed, specifying filter criteria over these properties. Once a reference to a suitable service is found, the client bundle uses it to obtain a Java reference to the service object. Bundles may register and get as well as withdraw and release services at any time. Since it is vital for bundles to react to these changes in their environment (like registration, withdrawal, or the change of properties of a service), the framework provides an appropriate *event mechanism.*

For remote administration, all administrative framework functions (like bundle life cycle management and configuration) are available through services. This "policy-free" approach leaves gateway operators free to choose a suitable communication channel for management by selecting an appropriate interface bundle.

Security in the framework rests upon the Java 2 security architecture. Though it is optional for a framework vendor to implement, significant consideration is devoted to it by the OSGi specifications. All parts of the API which are considered security relevant require appropriate permissions, which are granted on a per-bundle basis. This specifically includes service registration and access. Services can also define custom permissions.

2.2. Standard services

The OSGi specifications define a number of standard services, of which an OSGi platform can contain any subset (including none at all). Some serve to map existing Java standards to the dynamic nature of the platform, while others are uniquely OSGi-related. [7] contains a discussion of those included with the current specification release. For example, standard services add capabilities for role-based authorization of end users, provide a central logging facility and allow bundles to provide both static resources and dynamic content through a lightweight WWW server adopting the existing Servlet API.

The *Configuration Admin* service offers a uniform way to provide bundles with a persistent set of parameters. Adjustments made this way affect the global behaviour of a bundle, that is, all services it provides for client bundles.

The *Wire Admin* service supports the "wiring" together of services which register themselves as producers or consumers of data. Producers and consumers will usually not be aware of each other and do not need to hold each other's service object. They are, however, notified when a wire is attached to or removed from them. Both pull and push semantics are supported. To avoid flooding a consumer with unnecessarily frequent updates, a wire may apply filter criteria based on time intervals or value thresholds.

The OSGi *Device Access* concept assumes devices connected to the gateway are to be represented as device services, provided by appropriate driver bundles. Driver bundles are expected to build a hierarchy of device abstractions (for example, a Camera service on top of a Generic USB Device service). While these services and bundles are not fundamentally different from others, the Device Access service aims to automate this *refinement* process through automatic selection, download and installation of matching driver bundles. It should be noted that the OSGi Device Access specification defines no service interfaces for specific devices.

2.3. Specific challenges

The highly dynamic, multi-threaded nature of an OSGi environment presents a number of challenges to the service designer. First, and above all, one has to account for services the own code depends on being able to leave – and return – at any time. Therefore, their state has to be tracked permanently.

When a service indicates that it is about to leave, clients are required to release it promptly. Likewise, bundles are obliged to do so for all services they hold when they are being stopped. Releasing a service does not only mean ceasing calling any methods of the service object, but also releasing any further resources acquired by way of it, like event listeners. Actually, no references to instances of classes associated with the service should be kept unnecessarily. Failure to do so will result in effects highly undesirable for an "always-on" platform, like memory leaks or bundles not being able to get updated.

Yet, service providing bundles need not depend entirely on well-behaved clients. They can keep track of client bundles and their service requests. This is especially advisable when resources outside the Java runtime environment (like flash disk space) are held on their behalf. By monitoring a client's life cycle, these resources can be reclaimed once it stops even if it did not properly release them. Appropriate programming patterns can facilitate this task. One of them is the *Whiteboard approach* presented in [4]: By having event listeners register with the framework rather than the event source itself, the latter can take advantage of the automatic clean-up done by the framework to assist in releasing any allocated resources.

In a concurrent environment, possible race conditions and deadlocks also require careful attention. It is practically impossible to discover these by black-box testing. Yet – by the very nature of a component-based environment – most of the code a bundle interacts with will not be available for source-level analysis, and the provided documentation seldom reaches the necessary level of detail. Additionally, the bundle designer has little control over which implementation is provided for a required service, which introduces additional degrees of freedom. These issues should not be taken lightly as expectations regarding the stability of embedded platforms are usually high.

3. EIB

The *European Installation Bus* (EIB) is a field bus network designed to enhance electrical installations in homes and buildings. As such, it is clearly oriented towards narrow-band control applications. EIB is based on an open specification [5] maintained until recently by EIB Association (EIBA), which was founded in 1990. In 2002, EIB merged with other standards to form KNX [6] and is now maintained by Konnex Association. Nevertheless, "EIB" will definitely remain as a label for a specific KNX subset for some time.

The dominating physical medium is twisted-pair cabling. More recently, powerline communication and radio transmission were added (the latter already under the KNX umbrella). All of them operate at a data rate of less than 10 kBit/s.

The EIB network stack is conceptually aligned with the OSI reference model, with the Session and Presentation layers left empty. It adds a standardised application environment (referred to as User layer or Application Interface layer) to support nodes in using its application-level communication model.

The specification also encompasses standard hardware building blocks, the most important being the *Bus Coupling Units (BCUs).* BCUs provide an implementation of the complete network stack and application environment and can also host simple user applications. Since the processing power of a BCU is limited, more complex user applications will have to use a separate microprocessor. Although the application may still use a BCU for high-level access to the network stack in this case, light-weight twisted-pair (TP) solutions can opt for the so-called TP-UART IC. This IC also connects directly to the EIB but handles the network stack up to the Data Link layer only, with the further exception that the external microprocessor must determine whether its node is addressed or not.

3.1. Addressing modes

EIB is a peer-to-peer network system where the mapping between sensor inputs and desired actuator actions is also maintained in a decentralised way. Maintenance and regular operation often have different communication patterns. Especially in home and building automation, regular operation often involves addressing groups of communications partners simultaneously. Modifying the behaviour of a node however requires the ability to specifically address a single device. The design of EIB consequently addresses these diverse needs by providing two fundamentally different modes of addressing.

Group addressing is the preferred mode for regular operation. It allows to pass a piece of information to an arbitrary number of receivers by way of a single message with a fixed-length address field. This is achieved by making use of a *publisher-subscriber model.* Here, the sender uses a logical group address as destination address. Receiving stations know which group (or groups) they belong to, and accordingly either ignore or process incoming messages.

Thus, a sender does not require information which nodes will actually be receivers of its message. Any node can elect to "subscribe" to a group without the "publisher" knowing. Actually, it is neither necessary nor possible for a node to determine which other nodes will act as publishers or subscribers to a specific group address. To determine which nodes participate in a certain group communication relationship, one would have to examine every node in the system.

The *physical address* of an EIB node is closely related to its position within the topological structure of the network. The basic building block of an EIB network is the (sub-)line, which holds up to 254 devices in free topology. Following a tree-like structure, sub-lines are connected by main lines to form a zone, which can in turn be coupled by a backbone line. Overall, the network can contain roughly 60.000 devices at maximum.

Physical addressing is used whenever it is necessary to communicate with a single node specifically. This mainly involves device configuration and management tasks like downloading application software, setting parameters concerning device behaviour or configuring communication relationships.

3.2. Application level communication model

EIB uses a shared variable model to express the functionality of individual nodes and combine them into a working system. Although this model uses state-based semantics, communication remains event-based.

Every device publishes several application related variables which expose specific aspects of its functionality. They will usually either be data sources providing information to other devices, or data sinks which carry out certain actions according to the information received. These application related variables are referred to as *communication objects.*[4]

Communication objects of various devices are grouped at set-up time to form network-wide shared variables. The values of all communication objects of a group will be held consistent by the nodes' system software. Group membership is defined individually for each communication object of a node, and communication objects can belong to multiple groups. No limitations exist concerning the semantics associated with the individual communication objects of a group. This binding is entirely within the local responsibility of a node application.

Every group of communication objects is assigned a unique group address. This address is used to handle all network traffic pertaining to the shared value. Usually,

[4]It should be noted that they are loosely related at best to the concepts associated with an "object" in object-oriented programming.

data sources will actively publish new values, although a query mechanism is provided as well. Since group addressing is used for these notifications, the publisher-subscriber model applies: The group address is all a node needs to know about its communication partners. Its multicast nature also means, however, that no authentication or authorization can take place this way.

User applications are not concerned by which groups the node they reside on actually belongs to. The Application Interface layer actually provides them with an implementation of communication objects which they can use in a way similar to local variables.

For the EIB network stack, the contents of shared variables are opaque octet strings only. To assure that their values will be interpreted in a consistent way, the EIB Interworking Standard (EIS) defines a standardised bit-level syntax for various variable types. Unique identifiers are defined for these types as well as for various physical units which allow the configuration tool to ensure at set-up time that only compatible types of communication objects are combined. None of this typing information is ever transferred during EIB regular operation, however.

4. Abstraction hierarchy

When designing the interface of a field bus gateway towards the outside world, one encounters two conflicting demands. While most applications would profit from a technology-independent presentation, the inevitable (and actually desired) loss of detail incurred with such an approach will prohibit the use of specific features of the underlying field bus design. Acknowledging the fact that a single interface cannot serve every purpose, the obvious approach is to provide multiple levels of abstraction.

This way, a low-level interface can give access to the communication medium, offering only the most basic protocol support, but maximum flexibility. An intermediate, still bus-technology specific abstraction then can add the parts of the protocol stack needed for a particular application. On top, a technology-independent interface tailored to home and building control applications can reside. Figure 4 illustrates this hierarchy and its realisation for EIB. All components shown represent OSGi services. No assumption is made concerning their combination into bundles for deployment.

While this concept builds upon the well-known idea of layered protocols, it goes further by only loosely coupling the individual parts. This way, elements can be combined as needed at run-time by installing and linking suitable components. It also allows for the vertical division of concerns, as multiple components may contribute to one logical abstraction layer. One interface may service multiple higher-level components in parallel, as well as an abstraction layer may combine information from multiple components. This allows the resulting solution to be as lightweight as possible, while leaving the option to add functionality later on. Since interactions be-

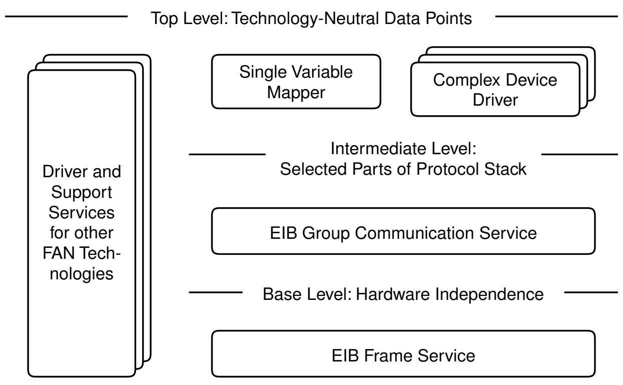

Top Level: Technology-Neutral Data Points
Driver and Support Services for other FAN Technologies
Single Variable Mapper
Complex Device Driver
Intermediate Level: Selected Parts of Protocol Stack
EIB Group Communication Service
Base Level: Hardware Independence
EIB Frame Service

Figure 4. Abstraction levels, components

tween OSGi services are effectively implemented as direct method calls, a fine-grained component structure like this should not incur inadequate overhead.

4.1. Base level

The interface at the low end of the abstraction hierarchy is to provide an abstraction for tasks which cannot be addressed adequately from within Java, specifically tight real-time requirements and hardware-dependent issues. In all that, it should not restrict the applications which can be built on top of it by offering maximum protocol detail.

For EIB, such a low-level interface is relatively straightforward. The services of the EIB Data Link layer do not place real-time requirements on clients[5] and allow uniform access to the physical media available. This interface can for example immediately be used to implement EIBnet/IP tunnelling.

4.2. Intermediate level

To gain high-level, yet technology specific access to EIB devices, one might expect that services representing individual EIB devices (possibly leveraging the OSGi Device Access mechanism) would provide a suitable additional abstraction level. This is inappropriate for a number of reasons, however.

The OSGi Device Access mechanism is designed to allow the platform to automatically adapt to changes in its environment without operator intervention. To this end, it is necessary that the networks it is connected to support some kind of automatic device discovery mechanism and provide a standard way for devices to provide enough information to allow the selection of a matching driver. Traditional EIB, however, being designed for a field of application where change is expected to be infrequent, provides neither. Without this basis, the automatic device refinement process is pointless.

While these issues could be worked around (by manually registering the needed information, if necessary), one encounters a more fundamental problem when attempting to expose functionality of a node as an OSGi service, as suggested in most OSGi-related literature [4, 11]. This

[5]Higher layers may very well make such requirements, however.

approach contains the tacit assumption that a device driver can “talk” to its target device without further ado to trigger some action. But due to the publisher-subscriber concept, access to application functionality of EIB devices in regular operation is associated with shared variables rather than devices. This means one cannot specifically change the state of a certain actuator, one can only do so through changing the state of the group (or one of the groups) it belongs to.

Thus, exposing EIB nodes as OSGi device services seems the wrong approach as far as regular operation is concerned. Since the service access points of an EIB network are actually values shared via group addressing, these are the entities to be exposed by an intermediate-level EIB access service.

Also, group communication relationships within the EIB network have to be designed taking into consideration the functionality to be provided by the gateway (given the state of the art, this will require a skilled individual). It has to be ensured that group addresses exist which allow it to address devices with precisely the required granularity. Also, no nodes must be left with a stale assumption regarding the state of another due to such an intervention.

Therefore, it does not make sense to provide special support for EIB set-up tasks. An intermediate abstraction can confine itself to handling regular operation (i. e. shared variables). The relevant EIB Application layer services do not prescribe upper bounds on execution times and are handed transparently through the Network and Transport layer. They are mapped to Data Link layer group communication services, but are restricted to messages which fit a single Data Link layer frame. Since group communication is unconfirmed, no timing restrictions therefore exist regarding the EIB network protocol.

Although outside the EIB network stack, translating transmitted values between the EIS bit-level syntax and appropriate Java data types will be another important task at this level. Besides transmitting updates of a shared value to other group members, the *Group Communication* service will also monitor any given group address, maintaining a copy of the last value distributed using it on the network for the use of client services. It can also automatically respond to read requests from the network if required.

4.3. Top level

The top-level interface is to be used by OSGi services presenting added value to the end user. Its task is to supply them with status information of and accept control messages for FAN devices. It is explicitly not intended to provide a generic way of addressing set-up issues, as their intimate relation to the specifics of a field bus design would make this a highly involved effort. The FAN (or possibly multiple FANs or other automation systems) whose functionality is exposed through this interface are assumed to be properly pre-configured.

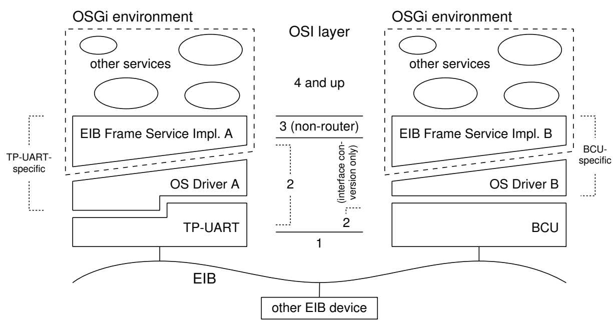


Figure 5. Hardware-software collaboration

Therefore, a *data point-oriented approach* is adopted. Its aim is to present the available control points in a uniform, technology-agnostic way centred around real-world entities, so that client services need not deal with specifics of a particular FAN. The binding of client services to these data points is expected to be accomplished manually by an operator or the end user him/herself. For this purpose, a scheme for enriching data points with *semantic information* is proposed, which allows to present them clearly arranged for easy identification.

Since the entities manipulated by this interface are objects relevant to the end user, his/her perception defines the applicable timing restrictions. A considerable fraction of a second seems allowable, although it would be desirable to stay below an end-to-end delay of 0.1 s. For EIB, [12] suggests that this is possible in a Java environment.

Since the application level communication paradigm of EIB is state-based as well, it will in many cases be sufficient to maintain a 1-to-1 relationship between a data point service and an EIB shared variable. A single component will be sufficient to perform this translation for any number of such values, like temperature sensor readings or wall switches. Nevertheless, a *driver* component[6] can also perform arbitrary additional processing, like a moving average. It can also handle the additional effort necessary for integrating more complex devices like dimmer or drive actuators, which are controlled through multiple group addresses and do not use entirely state-based semantics.

5. EIB base level abstraction

The *EIB Frame service* (EFS) provides the capability to exchange EIB Data Link layer protocol frames with other EIB devices. It also decodes the Network layer protocol control information, which consists of a routing counter only. Since end devices need not further process this value, the EFS also entirely covers layer 3 functionality when the node it resides on is not to act as a router. Still, arbitrary routing functionality can be built on it.

As illustrated in Figure 5, an implementation of the EIB Frame service will have to be specific to the EIB hardware interface (which in turn depends on the physi-

[6]Note that the standard Device Access service is not involved.

cal medium) and the low-level device driver of the host operating system. Yet, only a single common service interface exists, which is designed to accommodate all these variations. Implementations will provide an appropriate subset of the features accessible through it. For example, a TP-UART based solution can easily allow the dynamic configuration of a node's physical and group addresses, while a BCU is not designed for this information to be changed frequently.

To allow a client service to check on the actual capabilities of the service object implementing this interface, these are advertised as service properties together with its service registration. Should the client service request unimplemented functionality nevertheless, an exception will be raised. Global settings related to the underlying hardware, like the physical address of a TP-UART based node, are managed via the Configuration Admin service. This also includes whether the EIB hardware interface is operating in standard or bus monitor mode.

In order to allow different parts of the network stack to be handled by separate components, the EIB Frame service has to provide support for multiple client services. This way, for example, one component can handle group communication, while another is concerned with tunnelling. Obviously, the gateway operator has to ensure that competences are clearly divided between such components, either based upon the addresses they are prepared to process or by means of handling non-intersecting protocol aspects only. This is a significant extension with respect to the standard EIB network stack.

Figure 6 shows how the EFS interacts with clients and the environment. It is withdrawn when communication with the EIB is not possible for any reason and re-registered as soon as connectivity returns. Such exceptional conditions, which may be of interest to an operator, are passed to the standard Log service.

To transmit a frame, a client service simply calls the appropriate method on the EFS service object. This method will not return until an acknowledgement from the remote Data Link layer was received or the local Data Link layer gave up retrying. The result is passed as the method return value. To satisfy special requirements like router implementation, one method allows to specify a maximum of control information: priority class, routing counter value, source address, destination address and user data. For normal use, an overloaded method is provided where the EIB Frame service will supply default values as configured. Source and destination addresses are encapsulated by an immutable utility class, which also provides high-level methods for constructing and parsing them.

To receive incoming frames, clients register a specific listener service, whose service properties contain the addresses they are prepared to process. The EFS uses this information for two reasons: First, to relieve clients from having to perform this filtering on their own; second, to be able to fulfil the EIB layer 2 protocol, which requires an immediate acknowledgment if a node has been addressed.

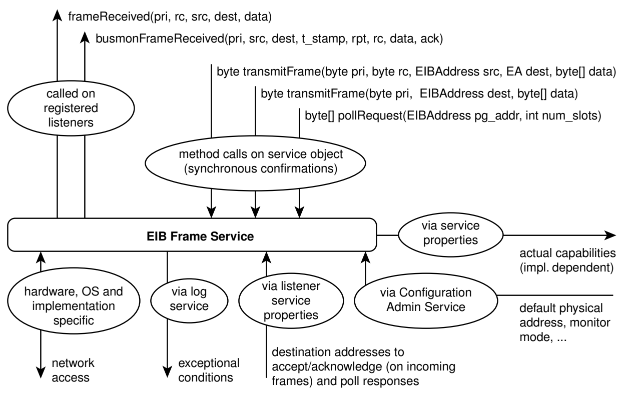

frameReceived(pri, rc, src, dest, data)
busmonFrameReceived(pri, src, dest, t_stamp, rpt, rc, data, ack)
byte transmitFrame(byte pri, byte rc, EIBAddress src, EA dest, byte[] data)
byte transmitFrame(byte pri, EIBAddress dest, byte[] data)
byte[] pollRequest(EIBAddress pg_addr, int num_slots)
called on registered listeners
method calls on service object (synchronous confirmations)
EIB Frame Service
via service properties
actual capabilities (impl. dependent)
hardware, OS and implementation specific
via log service
via listener service properties
via Configuration Admin Service
default physical address, monitor mode, ...
network access
exceptional conditions
destination addresses to accept/acknowledge (on incoming frames) and poll responses

Figure 6. EFS information flow

When a client is stopped, these registrations are automatically removed by the framework. This is an application of the Whiteboard approach, allowing the EIB Frame service to keep its list of addresses to acknowledge free from stale entries without having to depend on well-behaved clients. In bus monitor mode, a different method which allows to pass the extra detail available in this mode is called. Listeners need not implement this if not needed.

Additionally, the EIB Frame service supports the polling mode specific to the TP medium which allows a master to frequently monitor the state of a limited number of slaves with minimum protocol overhead.

6. Technology-neutral data abstraction

The top-level abstraction is specifically targeted at the residential application domain and smaller-sized functional buildings. It confines itself to handling control information only. An important goal is to support the construction of a clearly structured and immediately comprehensible user interface for monitoring and control purposes as well as binding data points to client services.

The abstraction provided can be used by any OSGi service. This may be an HTTP server which presents a control panel for local or remote use, a service which will send an SMS when the washing machine is detected to be spilling water or one implementing any of the technologies discussed in [14]. It should be noted that these services need not be IP-based. As an example, consider an SMS service which operates using a GSM modem connected directly to the OSGi platform host.

A key benefit of a technology independent abstraction is to ease the work of service providers and gateway operators, who are spared of having to provide and administrate multiple versions of a service with otherwise identical value to the end user. Moreover, it also plays an important role for local integration. Although one will initially select a single FAN technology to cover all demands, one may simply not have the choice for later additions. This may be due to the fact that the chosen technology does not support some new requirement, like device discovery, which is indispensable for the integration of loose goods. Such an abstraction also allows to cleanly integrate OSGi services as data point providers. For instance, a single "house mode" control may, for example, arm the alarm system and activate presence simulation.

The approach chosen is a *state-based* one, breaking down system functionality into values of primitive type. All these data points are represented in a uniform way. Technology independence is achieved by using real-world objects and properties as entities in the representation of the process image. This high-level semantic information is presented in textual form to the user only, however. Behavioural aspects are explicitly not addressed.

As a consequence, complex devices providing multiple points of control are exposed as a set of unrelated data points. This is not a problem regarding interaction with other software components, but certainly one for human users. To enable them to pick the right data point from this pool, another main constituent of the approach proposed is a concept for associating data points with additional semantic information pertaining to their effect location and purpose. This concept is not limited to data points belonging to the same node, but has universal applicability.

6.1. Functional aspects

Every data point is registered as a separate service object. In addition to a common service interface, a set of required property keys is specified, which enables client services to query the Service Registry for data points meeting specific criteria.

Although the state-based approach used may be reminiscent of the EIB application-level communication model, there are some major differences. First, the semantic information associated with data points is available for perusal by examining their service properties during regular operation. Secondly, this information is not related to field bus nodes, but associated with real-world entities. Third, the state-based paradigm is applied with full consequence. In an EIB system, data sinks are explicitly free to silently change application related status associated with a group variable. For example, an actuator controlling a stairway light usually switches off after a pre-set timeout without announcing the state change on the network. In contrast, data points are required to accurately reflect the status of their associated real-world entity at any time. This has to be ensured by the respective driver component, by dead reckoning if necessary. That way, the entire state of the system is always available.

Despite this obvious benefit, a purely state-based model cannot accommodate actions like the sending of an alarm message, which do not possess state by their very nature. To be able to include such functionality in the data point model as well – for example, to have a panic button trigger it –, a special type of data point representing an event is provided.

The properties of a data point are enumerated in Table 1. The *Property* property holds the aspect sensed or controlled of the real-world entity described by the *Object* property (e. g. "speed"). This *Object* is defined in a way independent of the specific use this information is put to (i. e. "wind", not "thunderstorm warning"). Both the *Object* and the *Property* properties can hold free-form text.

- Persistent unique identifier (PID)
- Object affected (air, door lock, glass, . . .)
- Property of object (temperature, state, integrity, . . .)
- Read/only (sensor) or read/write (actuator)
- Data type descriptor (constant with specified mapping)
    - Boolean, long int, float, string, . . .
    - Time stamp, day of week, time of day
    - State Set
    - Event (memory store/recall)
- Physical unit or state labels
- Minimum/maximum value

Table 1. Data point properties

In addition to the standard set of native Java data types, a means of specifying a certain day of week or time of day is included for the use with timer programmes. Also, data points which can enter a number of discrete, mutually exclusive states can be described properly.

Each of these types is associated with a specific constant value, which allows a client component to automatically generate appropriate user interface elements. For the same purpose, Boolean and multi-state types are accompanied by a set of state labels. The physical unit of numerical values can be provided as free text.

Concerning interaction with other services, it is obvious that multiple clients have to be supported. To ensure consistency, no read accesses must occur during the value being written. Implementing this requirement using the Java programming language is straightforward by declaring read and write access methods as synchronized.

For notification on update events, the Whiteboard approach again offers an elegant solution. By registering a single listener service, an interested party can receive update events from all data points. To identify the event source, its persistent unique identifier (PID) will accompany every notification.

The concept of a *pool of data points* integrates perfectly with the OSGi Wiring scheme. Given a suitable user interface, being able to "wire" data points can offer the end user a powerful, yet reasonably easy-to-handle tool for customizing platform functionality. Also, logging functionality based on time intervals as well as value thresholds can be implemented in a straightforward fashion. Yet, such criteria can only be associated with a single data point.

Concerning security and privacy, the OSGi framework already provides the necessary mechanisms to ensure that only trusted bundles are able to use data point or even low-level services unless they are given the proper permissions. Additionally, if a client bundle provides an entry point for user authentication (like the HTTP service), this information can be passed to the data point service

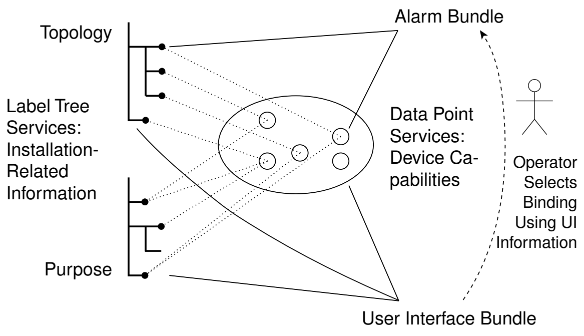

Topology
Alarm Bundle
Label Tree Services: Installation-Related Information
Data Point Services: Device Capabilities
Operator Selects Binding Using UI Information
Purpose
User Interface Bundle

Figure 7. Data points: Semantics distribution and binding

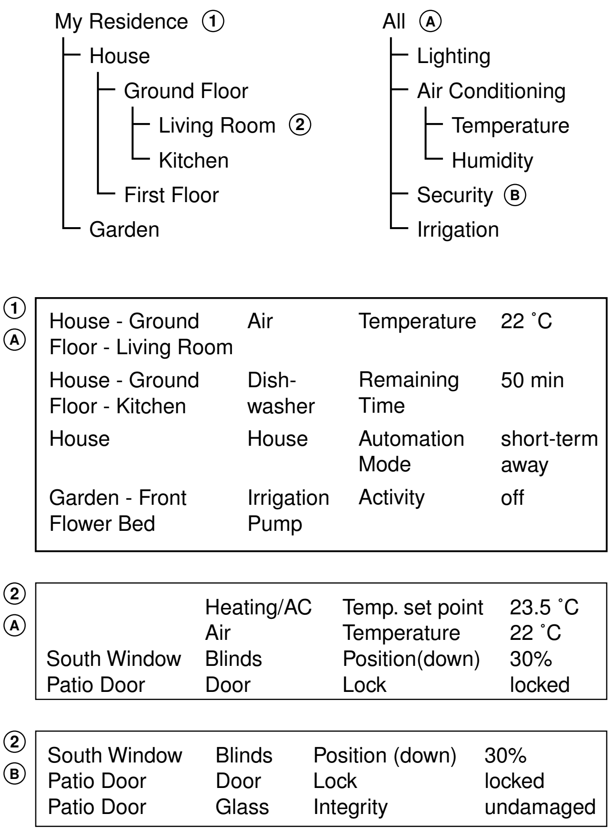
addressed. Another extension would be to limit the frequency of access for certain bundles or users as a privacy measure by implementing a custom permission class. For example, remote meter reading could be allowed once a month only. All these approaches benefit from having data points available as separate services for more fine-grained control. Although all these efforts are void as soon as an attacker gains access to the EIB physical medium, they raise the barrier for attacks via a remote connection.

6.2. Presentation

Individual data points only hold information about the kind of real-world entity they represent. Yet, this alone is insufficient for the end user, who will probably have multiple (for example) air temperature sensors within his/her living environment. Therefore, the concept proposed includes two persistent *tree data structures* holding information pertaining to the location of effect and purpose of data points. Every tree node is labelled with a free-text description and references a list of data points via their PIDs. Client components still access functionality associated with data points using the respective service interfaces of the latter. They will only be concerned with the tree data when the need arises to identify a data point they are dealing with towards human users. This concept is illustrated in Figure 7.

Regarding the topological structure, storing the *location of effect* was chosen over referring to the physical location of the associated node for the reason of three specific characteristics of the envisioned field of application. First, nodes are frequently installed in distribution panels, with passive cabling leading to the – often drastically different – actual location of effect, which is the one relevant to the end user. Secondly, an approach strictly aligned with physical topology cannot easily accommodate higher-level functions, such as one that allows to switch off all power outlets on the ground floor reachable by small children. Last, but not least, EIB as a widespread representative uses a communication model which renders such an approach plainly impossible.

Therefore, an approach is adopted which from a technical point of view can be considered *function-oriented* (as opposed to structure-oriented) [8, 14] in that it en-



```
My Residence ①                All Ⓐ
├ House                       ├ Lighting
│ ├ Ground Floor              ├ Air Conditioning
│ │ ├ Living Room ②           │ ├ Temperature
│ │ └ Kitchen                 │ └ Humidity
│ └ First Floor               ├ Security Ⓑ
└ Garden                      └ Irrigation
```

① Ⓐ

House - Ground Floor - Living Room	Air	Temperature	22 ˚C
House - Ground Floor - Kitchen	Dish-washer	Remaining Time	50 min
House	House	Automation Mode	short-term away
Garden - Front Flower Bed	Irrigation Pump	Activity	off

② Ⓐ

	Heating/AC	Temp. set point	23.5 ˚C
	Air	Temperature	22 ˚C
South Window	Blinds	Position(down)	30%
Patio Door	Door	Lock	locked

② Ⓑ

South Window	Blinds	Position (down)	30%
Patio Door	Door	Lock	locked
Patio Door	Glass	Integrity	undamaged

Figure 8. Data point presentation: Possible user interface

tirely disregards the underlying FAN topology. Yet, far from ignoring structure, it imposes it on the functional attributes of data points. Maintaining them in tree structure helps to ensure consistency better than having every data point hold this information individually (consider converting the children's room into a study).

While a data point will only appear once in the tree structure describing the location of effect (so it is fully qualified by its path plus its object and aspect), it may appear multiple times in the one describing its purposes. Window blinds, for example, keep out the sun as well as potential intruders.

Although the actual design of the user interface is not prescribed, Figure 8 shows a possible example. Part of the available locations of effect and purpose are displayed on the left-hand and right-hand sides respectively. The list of data points is filtered according to the tree nodes selected. No restrictions are made at first. Next, the selection is narrowed to data points related to the living room. Finally, the scope of view is further limited to security-related data points within this room.

7. Conclusion and outlook

Undoubtedly, gateways have a central role in "smart" homes and buildings by connecting control networks with one another and the outside world. OSGi offers a powerful framework to dynamically configure their functionality. It allows the run-time combination of software components, while still remaining suitable for resource-limited devices. Its advantages, however, come at the price of subtle dependencies providing pitfalls to the unwary software engineer. Taking into account that a gateway platform should operate continuously, a highly defensive style of programming seems definitely recommended.

For the technology-neutral representation of FAN functionality, a consequently state-based, data point driven solution was adopted. Data points are associated with extensive meta information, allowing the automatic construction of user interfaces. The concept also includes update notifications for clients and is designed for immediate integration with OSGi Wiring. In addition, basic directions for ensuring security and privacy were given.

Regarding presentation, the drawbacks of an approach oriented on the FAN physical topology were discussed. Consequently, a function-oriented approach was taken, which relates data point functionality to real-world entities meaningful to the end user. Nevertheless, it imposes a clear structure on the functional properties of a data point, replacing the physical topology by a tree of locations of effect. To provide further orientation, data points can be grouped according to their purpose.

Concerning the integration of EIB, it has become apparent that EIB cannot fully exploit the dynamic capabilities of an OSGi environment due to their static nature. Moreover, their integration is not straightforward owing to the fact that, concerning regular operation, devices can only be accessed by addressing groups of nodes. Specifically, it is necessary to consider the functionality to be exposed by the gateway when configuring an EIB network. As an additional obstacle, the state-based nature of the EIB application-level communication model was found not to be consequently implemented.

In spite of these difficulties, a solution was proposed which provides the optimum support possible for device driver components. In that, it surpasses solutions already on the market, like the EIB driver bundle available for the ProSyst "mBedded Server" [16]. This product only offers very basic assistance for group communication (for example, no translation of EIS types) and will not serve multiple clients.

The implementation of the proposed architecture is currently underway. For future directions, [10] outlines an extension of the EIB access components supporting the automatic reconfiguration of communication relationships and device parameters. This task will be greatly simplified by new "plug-and-play"-related features introduced by the KNX standard, which allow to exploit the dynamic nature of an OSGi platform more fully when implemented as well. In addition, the Internet remote service architecture announced by EIBA and Konnex Association should be examined for possible integration as soon as it is standardised.

Also, support for further FANs technologies needs to be implemented. Here, LonWorks comes to mind specifically as it is widely used in building automation and a comprehensive solution for device access is readily available [3]. Finally, considering spontaneous networking protocols, [11] proposes mechanisms for the integration of OSGi platforms into Jini and UPnP-enabled networks as well. A convenient mapping between these technologies and the approach presented here will further increase its usefulness.

References

[1] K. Bennett, P. Layzell, D. Budgen, P. Brereton, L. Macaulay, and M. Munro. Service-based software: The future for flexible software. In *Proc. 7th Asia-Pacific Software Engineering Conference*, pages 214–221, 2000.

[2] H. Cervantes and J.-M. Favre. Comparing JavaBeans and OSGi towards an integration of two complementary component models. In *Proc. 28th Euromicro Conf. on Component Based Software Engineering*, pages 17–23, 2002.

[3] S. Chemishkian. Building smart services for smart home. In *Proc. IEEE 4th International Workshop on Networked Appliances*, pages 215–224, 2002.

[4] K. Chen and L. Gong. *Programming Open Service Gateways with Java Embedded Server Technology*. Addison-Wesley, 2001.

[5] EIB Association. *EIB Handbook Series 3.0*, 1999.

[6] Konnex Association. *KNX Specifications, V. 1.1*, 2004.

[7] C. Lee, D. Nordstedt, and S. Helal. Enabling smart spaces with OSGi. *IEEE Pervasive Computing*, 2(3):89–94, 2003.

[8] M. Lobashov, G. Pratl, and T. Sauter. Applicability of Internet protocols for fieldbus access. In *Proc. 4th IEEE International Workshop on Factory Communication Systems*, pages 205–213, 2002.

[9] D. Marples and P. Kriens. The Open Services Gateway Initiative: An introductory overview. *IEEE Communications Magazine*, 39(12):110–114, 2001.

[10] G. Neugschwandtner and W. Kastner. EIB network access and configuration services for OSGi environments. Proc. KNX Sci. Conf., `http://www.konnex.org`, 2003.

[11] OSGi Alliance. *OSGi Service Platform Specification, Release 3*. IOS Press, 2003.

[12] R. Ott and H. Reiter. Connecting EIB components to distributed Java applications. In *Proc. 7th IEEE International Conference on Emerging Technologies and Factory Automation*, volume 1, pages 23–26, 1999.

[13] G. Pratl, M. Lobachov, and T. Sauter. Highly modular gateway architecture for fieldbus/Internet connections. In *Proc. 4th IFAC Conference on Fieldbus Systems and Their Applications 2001*, pages 293–299, 2002.

[14] T. Sauter, M. Lobashov, and G. Pratl. Lessons learnt from Internet access to fieldbus gateways. In *Proc. 28th Annual Conf. of the IEEE*, volume 4, pages 2909–2914, 2002.

[15] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[16] D. Valtchev and I. Frankov. Service gateway architecture for a smart home. *IEEE Communications Magazine*, 40(4):126–132, 2002.