An innovative, open-standards solution for Konnex interoperability with other domotic middlewares

Vittorio Miori, Luca Tarrini, Maurizio Manca, Gabriele Tolomei

Italian National Research Council (C.N.R.),
Information Science and Technologies Institute "A. Faedo" (I.S.T.I.)
Research Area, Via G. Moruzzi 1, I-56124 Pisa, Italy.

vittorio.miori@isti.cnr.it

I. Introduction

The Information and Communication technologies spread across our life to make easier our everyday tasks and to increase the quality of our existence in every domain realizing the Ubiquitous Computing vision of M. Weiser[1].

However, in the home environment this phenomenon is not so remarkable.

Konnex is one of the most important home computing middleware and represents the European domotic standard.

Actually, there are many others home computing middlewares available and just wide diffused, but often they are scantily interoperable. In our opinion these facts represent the main obstacle to the domotic market growth.

So, from a *Konnex* perspective, the main goal is to reach a high interoperability degree with all the other home computing middlewares.

In this way, the *Konnex* subnetwork can be integrated with any other home subnetworks, and the *Konnex* devices can cooperate with all the other devices independently of the domotic subnetwork which they belong to.

This paper proposes an innovative framework in order to achieve the interoperability between *Konnex* and the others heterogeneous home computing middlewares.

This framework, called **DomoNet**, implements a *Service Oriented Architecture* (SOA)[2]; it is based on the Web Services communication paradigm and on a "universal" XML-based home automation language, called **Domotics Markup Language** (**DomoML**).

II. DomoNet Architecture

In the last years, some interesting solutions are emerged[3][4], and each one of them aims to provide the interoperability among heterogeneous home computing middlewares.

In order to overcome the one-to-one protocol conversion approaches[5] and to improve the scalability, **DomoNet** is constituted by a set of Web Services, called **DeviceWebServices** (*DeviceWS*). Each *DeviceWS* manages a particular category of devices, such as Lighting, Heating, Alarming, etc…

In order to achieve the interoperability between all the nodes through **DomoNet**, it is necessary to develop a particular gateway between each subnetwork and **DomoNet**, called **TechManager** (*TM*), and a standard XML grammar, called **DomoML**.

DomoML represents a universal language that allows a standard communication between *TMs* and **DomoNet**/*DeviceWSs*.

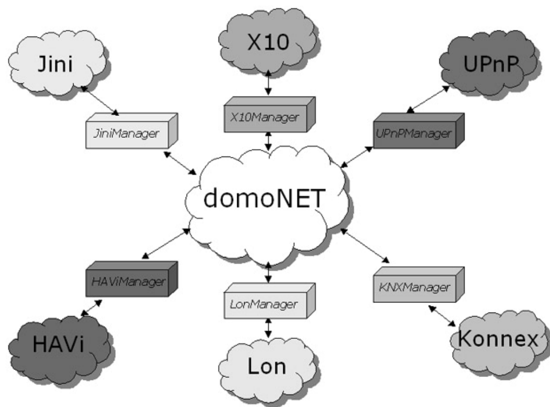

Figure 1.1 – DomoNet architecture.

DEVICEWS

Devices can be thought as services containers; each *DeviceWS* keeps track of all the devices belonging to a particular category and maintains the services that they offer in a useful data structure, an XML file.

The main task of a *DeviceWS* is to expose in a standard way all the devices it manages in order to give to all the subnetworks the possibility to have a uniform sight of the entire domotic network topology.

All the *DeviceWSs* share a standard service interface that is composed by a special set of web service methods.

DOMOML

DomoNet relies on two standardization processes: one of them works at the *DeviceWSs* level, while the other is necessary to develop a universal language to be applied in every domotic context. This universal language is an XML standard grammar called **DomoML**.

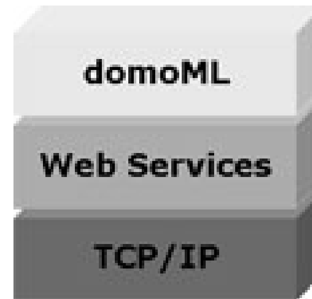

Figure 1.2 – DomoNet protocol stack.

DomoML could also be used in domotic contexts different from **DomoNet**. **DomoML** could be used by all applications based on automated treatment of the information in order to interchange data between domotic community entities.

TECHMANAGER

The *TM* is the application gateway between a particular home subnetwork and the **DomoNet** network; for this reason it must have two interfaces towards both networks.

There are three main activities the *TM* is dedicated to:

exporting of the home subnetwork configuration which is related to;

receiving all the other home subnetworks configurations through **DomoNet**;

translating the domotic protocol which is related to into the standard domotic language **DomoML**, and vice versa.

In order to allow interoperability between devices that belong to distinct home subnetworks, such as *Konnex*, *LonWorks*, *UPnP*, etc… **DomoNet** framework is based on the concept of *virtual device*. Each real device belonging to a specific home subnetwork, is mapped on a virtual device in any other subnetwork available in the home environment. This process involves both the *DeviceWSs* and the *TMs*.

III. KNXMANAGER

`KNXManager` is the interface module software that enables *Konnex* subnetwork to interact with **DomoNet** framework.

It was entirely developed in *Java* on *Windows* platform and it is composed by various software blocks, each one of them carries out specific tasks.

There are two main modules of which `KNXManager` is composed: one module is necessary to interact with **DomoNet** and Web Services through standard protocols such as *TCP/IP*, *HTTP*, *SOAP*, *WSDL* and *UDDI*, while the other module allows to interact with *Konnex* fieldbus through the *EIBA/Falcon* libraries.

The picture below shows all the modules by which it is composed the `KNXManager`'s software architecture.

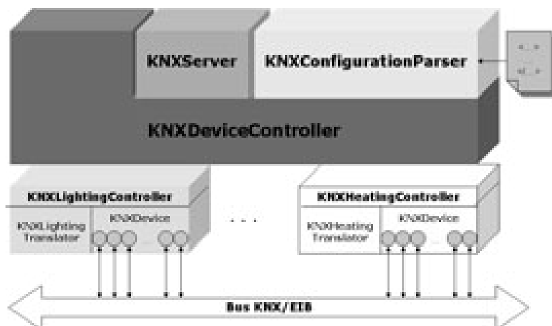

Figure 1.3 – KNXManager software architecture.

Following a top-down approach, let us analyze every single software module in an exhaustive way.

KNXCONFIGURATIONPARSER

In order to configure and setup a working *Konnex* network in which are present *S-Mode* devices, it is necessary to use the *ETS* software toolkit.

All the data and the informations made available by *ETS* must be collected in a structured standard way, a XML file.

This XML file contains all the *Konnex* configuration parameters (e.g. *group addresses*), and adds semantic informations about all the devices available on the network (e.g. device typology).

The `KNXConfigurationParser` module implements a DOM parser that takes in input the *Konnex* configuration XML file.

After parsing this XML file, it creates as many `KNXDevice` Java objects as the number of the devices that are present in the configuration file. Each Java object is a software entity that respects all the characteristics and plays the role of a real *Konnex* device.

Moreover, this module must also interact with the *DeviceWSs* in order to export towards **DomoNet** the *Konnex* configuration network.

KNXSERVER

This software module implements a multi-thread server to manage connection requests coming from *DeviceWSs* via socket TCP.

It carries out two main tasks: it accepts the connection requests coming from all the *DeviceWSs* to which **KNXManager** is registered to and it manages the data streams that flow on the sockets. In particular it receives **DomoML** messages via HTTP and it invokes the opportune translator module.

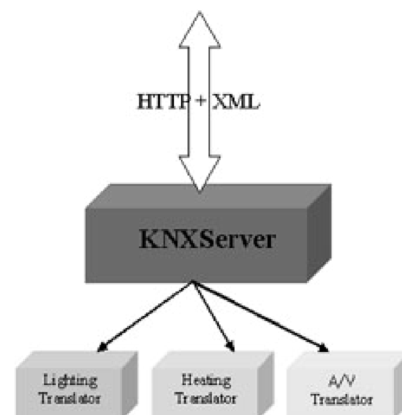

Figure 1.4 – KNXServer.

KNXDEVICECONTROLLER

It represents the software interface between **DomoNet**/*DeviceWSs* and the *EIB/Konnex* bus.

It is composed by a set of *controllers* modules, each one of them manages a specific device typology.

KNXDEVICE

At the heart of the interaction with both real and virtual *Konnex* devices, it is a software model that is based on the Java classes inheritance.

From an abstract point of view, the *Konnex* network appears like a set of devices that are able to read and write on the bus. For this reason, every *Konnex* device can be represented by an instance of a generic Java class that we have called `KNXDevice`. The instances of this class are proxy entities in the cares of both the real and the virtual *Konnex* devices.

`KNXDevice` class constitutes the root of the entire devices hierarchy: all the devices must extend this class adding their specific *services*.

We can imagine how to map this devices hierarchy in a real scenario; for example we can have two devices: a simple light and a dimmable light.

Both of these two devices must be represented by a Java class that extend `KNXDevice` class, then we can build a class in order to represent a simple light (e.g. `KNXLight`) and another class, that extends `KNXLight`, in order to represent a dimmable light (e.g. `KNXDimmableLight`).

It is simple to note that there is a tree-hierarchy in which the root is the `KNXDevice` class and the nodes represent all the real devices available in a particular *Konnex* configuration.

n+1-level nodes must offer at least one more service respect of those offered by n-level nodes.

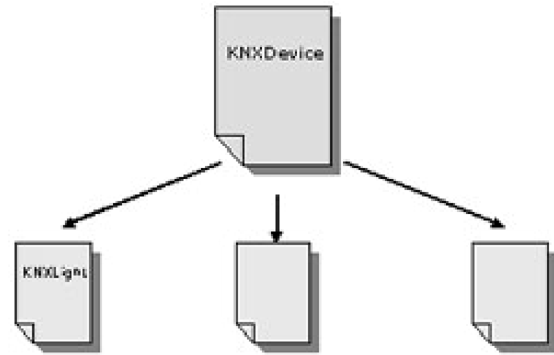

Figure 1.5 – KNXDevice hierarchy.

KNXMANAGERUI

This software module implements a simple user interface based on Java Swing classes. The interface carries out three main tasks: it shows the entire application status through a log window, it recreates the *Konnex* network topology through a hierarchical view of all the available devices and finally it allows to interact with the devices.

Figure 1.6 – KNXManager user interface.

IV. EIB/*KONNEX* BUS INTERACTION

In order to read from and to write to *EIB/Konnex* bus, it is necessary to use *Falcon* suite libraries, distributed by *EIBA* (*EIB Association*).

Falcon are *COM*-based libraries and so they are available for Windows platform only; for this reason it would seem to be natural to write an application that uses *Falcon* under the most

important Microsoft development environment, such as *Visual Studio 6* or *Visual Studio .NET*.

These IDE offer a native support for the *COM* model and allow to write the necessary source code in one of the most popular programming language, such as *C*, *C++* or *Visual Basic*.

However, the entire **KNXManager** architecture has been thought to be developed using Java programming language. So, in order to invoke the *Falcon/COM* libraries from a Java application, it has been necessary a wrapping operation through *JNI* (*Java Native Interface*) and `javah` tool.

JNI is the interface between a Java application and native code developed with other programming languages, such as C or C++. In other words, it allows to make native function calls from a Java application, and vice versa.

In order to interact with *JNI* there is a standard process composed by 4 phases:

- i) create a Java stub class that contains the signatures of all the native methods and a static method to load the `dll` wrapper that will be created at iv);
- ii) compile with the `javah` tool, the Java class just created and obtain a relative `.h` file;
- iii) implement this .h file in the native language (C or C++);
- iv) compile the source code developed at iii) in order to generate a `dll`.

An example of this `dll` wrapper, called `JFalcon.dll`, has been developed in *C++* under *Microsoft Visual Studio .NET 2003*; this `dll` is loaded by a Java stub class called `JFalconUtility`.

In this way, the real interaction between **KNXManager** software and the *EIB/Konnex* bus, has been managed by `JFalcon.dll` that is able to interact directly with the *Falcon/COM* libraries.

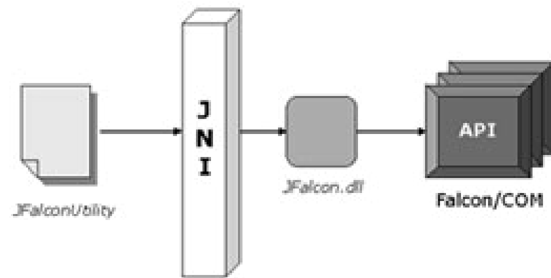

Figure 1.7 – *Falcon/COM* invocation through *JNI*.

V. THE PROTOTYPE

According to the specifications that were earlier defined, it has been developed a **DomoNet** prototype in order to demonstrate the interoperability between *Konnex*, *X10* and *UPnP*.

Obviously, this prototype presents some restrictions: in particular it manages only lighting devices.

However, it has been completely tested; its scalability and robustness allows any kind of future improvements.

VI. REFERENCES

[1] Weiser, M. "The Computer for the 21st Century". In *ScientificAmerican*. (1991).

[2] Papazoglou, M.P. and Georgakopolous, D. "Service Oriented Computing". In *Communications of the ACM 46, 10*. (2003), 42-47.

[3] Tokunaga, E., Ishikawa, H., Morimoto, Y. and Nakajima, T. "A Framework for Connecting Home Computing Middleware". In *ICDCSW Conference*. (2000).

[4] Lee, C.E. and Moon, K.D. "Design of a Universal Middleware Bridge for Device Interoperability in Heterogeneous Home Network Middleware". In *CCNC Conference*. (2005).

[5] Chemishkian, S. and Lund, J. "Experimental Bridge LonWorks/UPnP". In *CCNC Conference*. (2004).