Calimero: Next Generation

Boris Malinowsky Georg Neugschwandtner Wolfgang Kastner

Automation Systems Group
Institute of Automation
Vienna University of Technology

Treitlstraße 1-3, A-1040 Vienna, Austria
{malinowsky,gn,k} @ auto.tuwien.ac.at

Calimero is a Java library for access to KNX systems introduced in 2005. Its next generation provides numerous additional features. An improved architecture ensures continued ease of use. Increased modularity is provided by a feature bundle concept. Licensing options were extended. This paper presents Calimero Next Generation, discussing its design and benefits. Special emphasis is placed on key parts of the new API.

1 Introduction

Calimero is a Java library for KNX access. It was first presented to the public at the KNX Scientific Conference 2005 as a part of the KNXLive! project [1]. This project offers easy access to KNX systems, middleware, and development tools based on a Linux Live-System (Knoppix, [2]).

From the beginning, Calimero was also available on SourceForge (versions 1.x, [3]). The packages included in this distribution provided a KNXnet/IP (EIBnet/IP) [5] Discovery and Tunneling client, APDU (application layer protocol data unit) type translation services essential for runtime interworking, and a simple group address storage using XML together with a GUI based demo application. Ease of use was a key design goal. Client applications were enabled to communicate with KNX devices via a high level API hiding network protocol details.

Since then, development on Calimero continued. When considering new functionality to be included, the question arose how these features could be added while maintaining both ease of use and a compact footprint. Therefore, a considerable reorganization effort was undertaken, leading to a redesign of the user API as well as internal architectural aspects. The new Calimero NG API, in the following only referred to as “Calimero”, now supports additional connection protocols, management, property access, high level convenience methods for common tasks, and more.

- The Calimero library now only requires the Java 2 platform, Micro Edition (J2ME) APIs. Supporting J2ME enables deployment on resource constrained devices, such as embedded systems.
- Modularity was improved further by introducing feature bundles grouping related functionality. These bundles are loosely coupled, interdependencies were eliminated wherever possible. This concept helps the user in understanding the API. Moreover, it enables tailoring the library to a specific task since only bundles relevant to the application need to be included. Ease of use is maintained by a bundle offering high level convenience methods as a “single point of access” for common user tasks.
- Extensibility of functionality continues to be an important aspect. The architecture was designed to allow new features or alternative implementations of services to be added seamlessly.
- Besides KNXnet/IP Tunneling, Routing is now provided as an additional connectivity option. Network Address Translation (NAT) is supported as specified in the KNXnet/IP documentation.
- A generic KNX network access link abstraction was introduced to easily accommodate further types of network access units and the protocols they use. This abstraction also shields higher level services from connection and medium specific details.
- Support for network and device management was added. This includes support for KNXnet/IP Local Device Management as well as KNX application layer property services. The Calimero property client provides a unified interface for accessing Interface Object properties, both via local device management or remote property services.
- An API for buffering incoming KNX network messages was added, considering the difference between state semantics (caching, to improve performance) and event semantics (queueing, e.g., to support client polled applications).
- Logging is now supported throughout Calimero, using a generic logging API for various kinds of events such as errors, status or progress information.
- To introduce certain features in Calimero and give a survey of the library, example console based tools are published. Each of these covers a specific use case (such as reading a group value), highlighting the required APIs in tutorial fashion.
- Most Calimero functionality is now backed up by unit tests. Their aim is to provide an automated way to test if the corresponding specifications are implemented correctly, to check for regression issues when extensions or changes are made, and to trace problems in a controlled environment.

This paper presents the architecture behind these features and shows how to use the new API by way of short code examples.

2 Architecture

When writing software that accesses a KNX network, it is necessary to bridge the gap between the application and the KNX medium. This can be done with a variety of protocols, like working over an IP network interface or using a serial cable connection to a bus coupler. Additionally, the KNX network can use different communication media, requiring appropriate medium specific formats.

It would be beneficial for higher level services in Calimero to rely on a well defined interface abstraction that encapsulates and hides those kinds of differences. As a consequence, the entire design of Calimero follows an architecture also known as the *waist-line architecture.* Figure 1 shows how, in this architecture, functionality is divided into three layers. Implementations of basic services – in our case, the range of network access protocols – are located at the bottom. In the middle, the network link abstraction offers a homogeneous, standard interface for communication with KNX networks, hiding the chosen basic service. This link, with its slim, stable API, is considered the “waist” in the architecture. Building upon the link abstraction, high level services ease development, and users create their applications depending on a link, a high level service, or both.

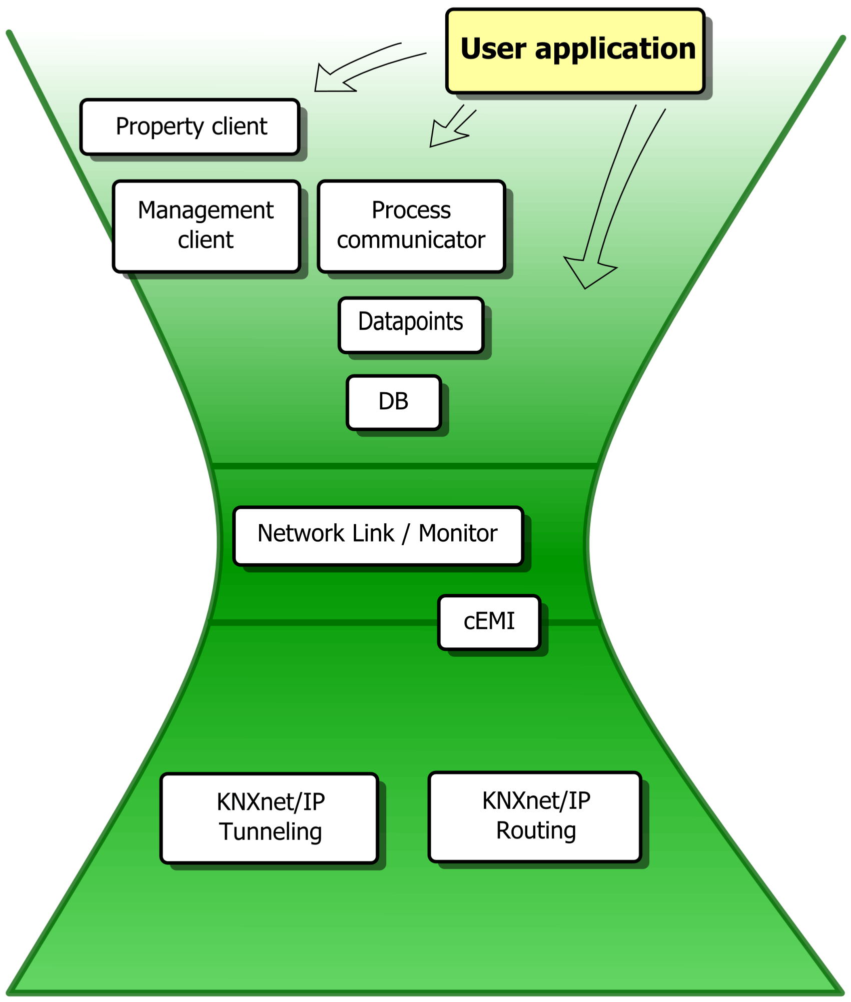


Figure 1: Waist-line architecture and non-strict layering in Calimero

Calimero follows a *non-strict layering* approach. Even if components build on one another, the user is not forced to interact with those at the top level only. Thus, a user is able to choose the desired level of API abstraction. Corresponding to particular application requirements, he may opt for convenience or detailed options. On the one hand, he could use a high level API such as the *Process Communicator* that provides common tasks out of the box. On the other hand, he could opt for the low level KNXnet/IP interface if a non-blocking communication mode is required.

High level services using the link abstraction can immediately and transparently make use of new network access protocol implementations as these are added. Also, most relevant interfaces have been designed to allow extensibility, for example by adding new cEMI types, additional application layer services to be used in the process communicator, or datapoint types.

2.1 The network link abstraction

For accessing a KNX network, it is necessary to pick the appropriate protocol for talking to the network interface (e.g., a KNXnet/IP router) and know its particular parameters (e.g., its IP address). An implementation of the `KNXNetworkLink` interface obtains this information when a new instance is created. In addition, it can store medium specific default values (e.g., the domain address) when overloading the defaults defined in the network interface is desired.

Subsequently, these details stay transparent to the user of this link object. All Calimero high level services use such a link for communication. On creation, these types are configured with a – previously created – network link. When, for example, taking a look at the management client constructor,

```
public ManagementClientImpl(KNXNetworkLink link)
```

no connection specific parameters are necessary at all, just an open link implementing the `KNXNetworkLink` interface. The same link object can also be supplied to other higher level services for shared use.

Although the key intention of the network link interface is to provide a medium independent abstraction, passing medium related details specific to single KNX messages should not be prevented. Therefore, a link supports two ways of sending a frame. First, it will accept a *network layer service data unit* (NSDU) plus a KNX destination address. In this case, the link creates the remaining KNX message frame structure, using its knowledge about the medium and specified defaults. All Calimero higher level components use this NSDU API.

As an alternative, the link also accepts link layer cEMI messages. For this purpose, Calimero provides types which are aligned with cEMI message structures (described below). The cEMI message format (common external message interface, [4]) is leveraged as a standard way of accommodating all possible medium specific aspects. This way, a user can control most details of an outgoing message. He could even add additional information records to a message without making changes to Calimero.

Messages incoming from the KNX network are always provided in cEMI format through the link abstraction. Since cEMI encapsulates medium-independent information in a generic way

– independent of any medium-specific details which may be present – message receivers are free to choose the abstraction level appropriate for them.[1]

Since KNX devices in bus monitor mode do not allow outgoing transmissions, an additional link interface is available for this purpose. In contrast to `KNXNetworkLink`, `KNXNetworkMonitor` only supports receiving bus monitor indications. Such a Calimero monitor optionally decodes incoming raw frames from the medium. Medium dependent message types parse control information up to (and including) the network layer. Since TPDUs (Transport layer PDUs) are medium independent, the same parts of Calimero that are used for standard communication can be used to decode them.

2.2 Messaging with cEMI

Since cEMI [4] plays a key role in both the Calimero link abstraction and KNXnet/IP, appropriate support is essential. The Calimero cEMI API allows construction and parsing of cEMI messages in a comfortable way. It provides medium-independent access to KNX messages.

Specialized cEMI types represent the different kinds of cEMI message formats. Currently, the link layer data (L-Data), bus monitor, and local device management formats are implemented. All of them support creating new messages – using supplied cEMI field parameters and/or default values – and creating messages by decoding byte arrays containing the appropriate frame format.

Two variants of the L-data type are available. The base type supports standard frames on the TP medium, which is sufficient for the majority of KNX installations today. Its objects are immutable and thus have the advantage of avoiding synchronization and copy on return issues. In addition, a subtype supports extended frames and additional information. Since medium specific information is usually added by the link after the message was created by a medium independent high level service, this subtype is mutable. PL and RF specific information is implemented. Both L-Data types encode and decode control information up to the network layer. The TPDU is treated as an opaque byte array. The bus monitor type decodes (and allows to set) all status information and the timestamp, with the raw frame expected and returned as a byte array.

2.3 Footprint considerations and modularity

Mobile phones running Java have become a commodity item. As the trend towards ubiquitous computing continues, embedded Java can be expected to gain importance continuously. Therefore, a key design goal for Calimero NG was to require only a Java runtime environment suitable for resource constrained devices.

The minimum runtime environment for Calimero now is J2ME, CDC (Connected Device Configuration) 1.1 with the Foundation profile. Since Java 2 Micro Edition does not provide XML handling and logging, Calimero comes with its own implementations of these tasks.

XML support was included to support data persistence using an open, flexible and editable format. The implementation contains a basic XML reader and XML writer, both non-validating.

[1]While medium dependent aspects can be ignored when receiving a message, this is not the case for sending it. Therefore, it would not make sense to let high level services create medium independent cEMI messages for transmission, since they would have to be re-packed into a medium dependent type anyhow.

To reduce memory consumption, the reader uses a pull parser that reads the input piece by piece, with the application processing data as it is parsed.

The logging framework allows filtering according to multiple severity levels. Logging to a stream, a file, or a network socket is possible. It is designed to be generic enough to be useful as the single logging facility for smaller user applications. On the other hand, log output can easily be integrated into any preferred application global logging facility.

Besides keeping the required runtime environment compact, another design goal was to allow the user to reduce the size of the Calimero library itself. He should have the ability to actually remove unused library parts by merely deleting class files, without touching the source code.

The first step towards achieving this goal was to arrange functionality in a highly modular way. Services related to each other were grouped to *feature bundles*. Each feature bundle addresses a specific task; dependencies between bundles were kept to a minimum. Bundles form a dependency hierarchy, with high level services relying on basic services. Cyclic dependencies are avoided. This particularly clean design makes it easy to grasp which bundle dependencies are present.

Within the hierarchy, bundles with no incoming dependencies can be safely removed. Such a removal just reduces the library by the features of the respective bundle. On the other hand, when a bundle (let us call it A) depends on another (let us call this one B), it usually makes no sense to remove B since A would not be able to do anything useful without the functionality provided by B anyway. However, there are cases where A would just lose a nice-to-have feature, but could still provide its core function. In Calimero, this is the case with XML handling.

XML handling is used widely throughout the library for data persistence. Usually, this would create static dependencies between the XML bundle and many others. To remove this dependency, the bundle's top package (in the Java package hierarchy) only contains `interface` declarations, with the actual service implementation moved to a sub package. Other bundles only reference these interfaces and obtain the actual implementation classes from a factory, which is also included in the top package. Thus, the bundle can be removed by eliminating the sub package, with only the small bundle stub containing the interfaces remaining.

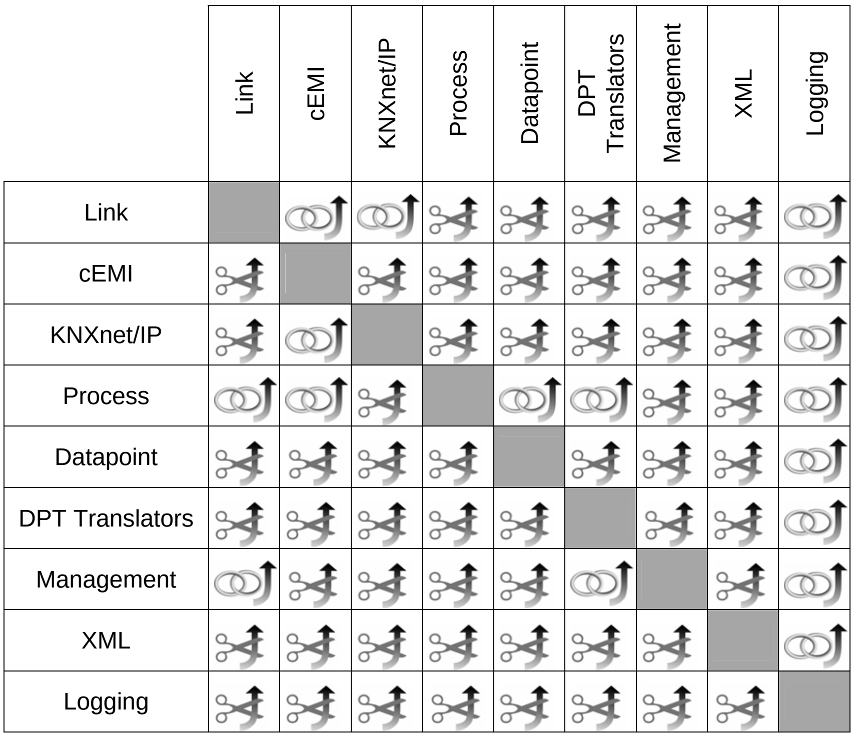
```
package tuwien.auto.calimero.xml;
        // stub, contains XML interfaces and factory

package tuwien.auto.calimero.xml.impl;
        // the actual implementations are contained in subpackage(s)
```

This approach allows the user to remove the XML bundle when no XML data needs to be handled (e.g., in an embedded system) or persistence is managed in the user application. Additionally, it enables Calimero to switch to a different (e.g., validating) parser in the future.

Logging is another classic example of a cross-cutting concern. However, the Calimero logging bundle is lean enough that the complexity added by separating implementations and interfaces would outweigh the benefits. Therefore, it is considered an integral part of Calimero and cannot be left out without leading to compile time errors.

Figure 2 shows the static dependencies present between Calimero feature bundles, with arrows representing the direction of the dependency. A chain symbol indicates a dependency, while a scissors symbol indicates that no dependency is present. For example, the sixth row



	Link	cEMI	KNXnet/IP	Process	Datapoint	DPT Translators	Management	XML	Logging
Link	▨	🔗	🔗	✂	✂	✂	✂	✂	🔗
cEMI	✂	▨	✂	✂	✂	✂	✂	✂	🔗
KNXnet/IP	✂	🔗	▨	✂	✂	✂	✂	✂	🔗
Process	🔗	🔗	✂	▨	🔗	🔗	✂	✂	🔗
Datapoint	✂	✂	✂	✂	▨	✂	✂	✂	🔗
DPT Translators	✂	✂	✂	✂	✂	▨	✂	✂	🔗
Management	🔗	✂	✂	✂	✂	🔗	▨	✂	🔗
XML	✂	✂	✂	✂	✂	✂	✂	▨	🔗
Logging	✂	✂	✂	✂	✂	✂	✂	✂	▨

Figure 2: Static bundle dependencies in Calimero

shows that the DPT translator bundle only depends on the logging bundle. As another example, the rightmost column shows that every bundle in the library depends on the logging bundle.

Besides footprint issues, the feature bundle concept has the benefit of giving the library an easily comprehensible structure. It allows a developer to focus on features that are of particular interest to him. The new console "tool" examples take the same line, focusing on a single aspect of the API at a time (besides, they also work in a J2ME environment).

3 KNX network access via KNXnet/IP

KNXnet/IP (formerly known as EIBnet/IP) is a protocol specification for working with KNX networks over an IP connection. From the beginning, KNXnet/IP support has been a core feature of Calimero. With Calimero NG, client implementations of all KNXnet/IP service protocols whose specifications are available in the KNX Handbook [5] are available in the KNXnet/IP bundle.

These are Core (discovery & description), Tunneling, Routing, and Device Management. `KNXNetworkLinkIP` implements the `KNXNetworkLink` interface to provide Tunneling or Routing. `KNXNetworkMonitorIP` provides Tunneling on the bus-monitor layer via a `KNXNetworkMonitor`. KNXnet/IP Device Management is used by the Calimero Property Client (described in Section 6.2). Currently, communication is done using UDP only. Wherever applicable, specification compliant [6, p. 37] Network Address Translation (NAT) aware communication is possible to address situations where IP datagrams have to traverse network routers using NAT.

The `Discoverer` offers KNXnet/IP server discovery and self description. The discovery API can be used in nonblocking or blocking mode, optionally with a timeout limit for remote discovery endpoints to respond. It is possible to use a specific outgoing local network interface for search requests. A search can also automatically be performed on all available network interfaces.

KNX messages are handled using appropriate cEMI message types as described in Section 2.2. For outgoing messages over KNXnet/IP connections, three different kinds of blocking modes are available: *non-blocking*, *wait-for-acknowledge* and *wait-for-cEMI-confirmation.* In non-blocking mode, no response is waited for; the user has to mind any protocol tasks. With wait-for-acknowledge, the send method blocks until the KNXnet/IP service acknowledge response, handling all necessary KNXnet/IP protocol tasks such as retransmission. With wait-for-cEMI-confirmation, the send method blocks until the confirmation is received, implementing cEMI flow control [4, p. 10]. Since Routing is an unreliable protocol by design, only non-blocking mode is supported here.

4 DPT translators

DPT translators bridge the gap between low-level data representations on the KNX network (KNX datapoint types, DPTs [7]) and Java data types. A KNX datapoint type consists of a data type and a dimension. The data type specifies the format and encoding, while the dimension describes a range and unit associated with a particular data type. Datapoint types are identified using a main number referring to the data type, and a sub number referring to the dimension.

One Calimero DPT translator handles a single KNX main number. It also provides a list of the available KNX datapoint types (i.e., dimensions) for that data type. The translator encodes and decodes data items according to the additional semantic information provided by one of these DPTs. New or modified DPTs can be added by the user. Available DPTs could also be removed.

A datapoint type is represented by the `DPT` type. A `DPT` stores semantic information about one KNX data point type behind a unified interface, consisting of the unique DPT identifier, a general description, unit of measure and a lower and upper value representation in `String` format, specifying the value range. The general description largely follows the DPT names in [7], but is edited for readability. For example, assume that `dpt` refers to the KNX DPT 5.003 from the 8-bit unsigned translator:

```
dpt.getID();          // returns "5.003"
dpt.getDescription(); // returns "Angle"
dpt.getLowerValue();  // returns "0"
dpt.getUpperValue();  // returns "360"
dpt.getUnit();        // returns "° "
```

`DPT`s may subtype `DPT` for offering additional type specific information. This applies to composite DPTs in particular. For example, B_1U_3 (3.007 “Control Dimming”) provides the B_1 control Bit type information. To configure a translator to use a particular DPT, the `DPT` is supplied on translator creation:

```
DPTXlator t = new DPTXlator8BitUnsigned(dpt);
```

The API of each DPT translator supplies Java data types suitable for the data type (main number) it covers. For example, the Java type `boolean` is provided for the KNX data type B_1. All translators accept and return value representations in `String` format. Both for string output as well as for dimensionless input, the dimension information of the DPT is applied automatically. Thus, a user can rely on "real-world" value representations exclusively, without even being aware of the data type encoding. The string representation (e.g., "250.4 lx") is interpreted by the translator using `DPT` information and offered as a KNX data byte array (and vice versa).

The `XlatorTypes` interface maintains a mapping between all available KNX data types (identified by their main number) and their corresponding translator implementation classes, together with an optional description. It can be used for lookup or to request a specific translator given a main number and sub number or a DPT ID. These map entries can be added, modified, or removed.

When working with KNX Interface Objects, property values need to be translated. The DPT translator bundle also addresses this requirement. A Property Data Type (PDT) describes the type of such a property. `PropertyTypes` can be used to create and obtain an appropriate translator for a given PDT. It maintains PDT to DPT map entries, initially associating `PDT`s with a DPT suitable for canonical representation (and sufficient for most translation purposes). Again, the map entries can be modified.

A property value might not only consist of a single element, but also of an array of elements of the same encoding type. This is taken into account in DPT translators by supporting containment and translation of multiple items of the same type, for example by supplying a Java byte array with KNX data containing multiple elements.

5 Process communication

Process communication refers to the exchange of process-related data (e.g., sensor values or actuator commands). In KNX, process communication de facto exclusively uses group communication. A number of steps are necessary for sending a message to the KNX network to engage in process communication:

- The data value (or command) has to be transferred to its KNX representation. This is done by the DPT translator bundle already presented.
- The appropriate application and transport layer control information has to be added to create a TPDU.
- Control information for the lower layers has to be obtained. At the least, this information consists of a destination group address.
- All this information has to be supplied to a `KNXNetworkLink`, which assembles the frame and transmits it to the network.

The same procedure has to be performed the other way round for receiving. Each of these tasks can be performed on different levels of abstraction via specific APIs available in Calimero. However, Calimero also offers a simplified one-stop API. In addition, a sophisticated framework for buffering data incoming from the network is provided. Both make use of a network configuration database.

5.1 Datapoints

The Datapoint bundle enables Calimero to store information about the KNX network configuration for process communication purposes. In KNX, process communication endpoints are identified by group addresses. Thus, the `Datapoint` type holds semantic information associated with group addresses – for use by the user application or high level services within Calimero. The set of `Datapoint`s is provided through a simple interface offering datapoint lookup, enumeration and persistence in XML format.

A `Datapoint` is uniquely identified by its associated main group address. It also holds a user-supplied friendly name as well as the KNX DPT main number and sub number. Thus, a `Datapoint` possesses the knowledge to identify KNX messages belonging to it and to interpret them correctly.

With Calimero NG, `Datapoint`s store further useful information. This includes transmission priority and the possibility to distinguish between state based and event based datapoints. State based datapoints describe a process value ("switch is off"). Such states persist over an interval of real time. Event-based datapoints transport commands ("turn 20 degrees"). These events are associated with a single point in time.

Often, a state based datapoint can be assumed to hold its value until an update request for this datapoint appears on the network. However, it may also change its value in response to time passing (e.g., the status output of an actuator implementing a timer switch) or communication activity on other datapoints (e.g., the output level of a dimmer when a "dim up" command is received). Depending on the network configuration, these changes may or may not be announced over the network.

For this purpose, a state based datapoint may optionally contain a time-out interval and a list of invalidating group addresses. Incoming group value write indications with their address in the invalidating list will invalidate the current data point value, i.e., the value is marked to not be up-to-date anymore. In special cases, the state based datapoint may be known to take over a value from another group address (e.g., the status feedback output of a switching actuator being deactivated by an "all-off" command). To cover these situations, a state based datapoint may contain a list of updating group addresses.

5.2 Using the Process Communicator

The `ProcessCommunicator` brings together the parts of the Calimero API required for solving common process communication tasks. Like every other high level service, it expects a `KNXNetworkLink` object at creation time which it uses for all subsequent communication.

```
// create some network link
KNXNetworkLink lnk = new ... ;

// create communicator and supply link
ProcessCommunicator pc = new ProcessCommunicator(lnk);
```

The `ProcessCommunicator` offers a blocking API with a collection of convenience methods for reading and writing group values. It is also possible to listen for incoming group value write indications. If required, a user defined response timeout for `read`-methods to complete can be specified. All APDU and type translation issues are hidden. For example, reading a `boolean` from group address 1/2/4 is as easy as

```
boolean b = pc.readBool( new GroupAddress("1/2/4") );
```

Writing a `float` is no more difficult:

```
// write Pi (rounded) to group address 3/2/2
pc.write( new GroupAddress("3/2/2"), 3.14159f );
```

The API can be used in an especially convenient way by leveraging `Datapoint`s:

```
System.out.println( pc.read(aPoint) );
```

will provide a human-readable `String` describing the value held by the datapoint referenced by `aPoint`.

5.3 Network buffer

Sometimes, it is useful to temporarily store information incoming from the network. One reason may be to be able to answer frequent application queries for process values without causing network load; another, the requirement to support a client polled mode of communication. Calimero includes a flexible, customizable network buffer API that addresses this requirement. It is backed by a generic cache infrastructure.

To enable buffering of messages incoming via a `KNXNetworkLink`, its owner creates a *buffer configuration* for it. This creates a tee in the message flow, allowing filters to be inserted. These filters apply individual rules to incoming messages to decide whether they should be stored in the cache. Likewise, they may answer user requests with cached information. A *buffered link* is created which can be used transparently in place of the original `KNXNetworkLink`, if filter semantics allow.

Besides these filter settings, the buffer configuration also holds related settings such as the maximum total buffer size associated with its buffered link or which KNX installation a link belongs to. Multiple configurations (and thus filters) are possible for a single link. A repository of all buffer configurations currently in place is available.

Two predefined filters are implemented. The first checks the APCI of incoming and outgoing messages for group value services. Received datapoint values are stored in the cache and used to answer group value read requests coming from the application. This filter can also be used with the Process Communicator. The other filter queues incoming events. Thus, instead of having to react immediately to indications from the network, user applications can query this filter for new data at arbitrary intervals. In addition, developers may implement their own, tailor made filters.

6 Network and device management

The Calimero library also contains client-side implementations of network and device management functionality. They offer an easy-to-use blocking user API. Special support is provided for handling KNX interface objects.

6.1 Management client

The management client offers common application layer services for network and device management. It builds upon the `TransportLayer` interface for connection oriented communication. A user can read and modify the individual or domain address in devices in programming mode (or by serial number), read and write properties (on a basic level) or memory as byte arrays, perform authorization, modify or delete access keys, read the ADC, read a device descriptor, or request a restart of the remote communication partner.

For point-to-point communication, an object of type `Destination` has to be created by the user first. It describes a remote device destination, specifies the desired communication mode – connection oriented or connection less –, and allows to control if verify mode should be used for the particular destination. For connection oriented communication , the `Destination` also maintains the protocol state machine together with its associated time-outs. The following example shows how to obtain the device type number from a BCU 1:

```
// create the client and supply a KNXNetworkLink
ManagementClient mc = new ManagementClient(lnk);

// create connection-oriented destination
Destination dst = mc.createDestination(new IndividualAddress("2.4.1"),
  CONNECTION_ORIENTED);

// read 2 bytes of memory from the destination into mem
byte[] mem = mc.readMemory(dst, 0x105, 0x02);

// close connection & destroy destination object
dst.destroy();
```

6.2 Property client

The property client is a high level service for working with interface object properties. It offers read and write access to properties by way of Java types, readout of property descriptions, property scans, PID lookup and support for loading, saving and working with user defined property definitions. When a property value is read, its property data type can automatically be evaluated to return an appropriate DPT translator or even the ready formatted string representation of that value.

The `PropertyClient` interface offers unified access to interface object properties. The KNXnet/IP Device Management protocol can be used in the same way as application layer remote property access services. For this purpose, the property client depends on a *property adapter*. This adapter manages the connection specifics and compensates the differences present between “local” and “remote” property access (e.g., authentication). This concept is shown in Figure 3.

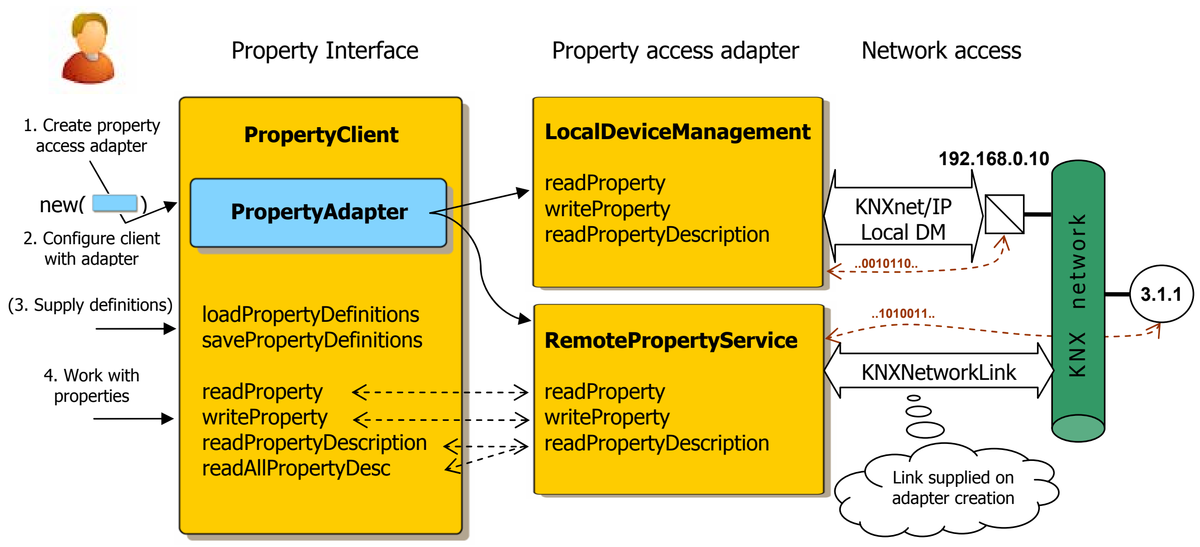


Figure 3: Using the property client interface

Optionally, it is possible to load and save property definitions to be used by the property client for lookups and automatic property type translation. Such a definition contains useful information like the PID, PID name, a friendly name of the property, object type, PDT and DPT (not all of it required). The definitions can also be used for providing a more descriptive user interface.

These definitions may be loaded from (and saved to) any resource, such as a file, network stream, temporary storage or similar. By default, a file handler for loading and saving property definitions using XML is provided. Calimero supplies all property definitions in XML format listed in [8]. Nevertheless, a user may choose to specify a different resource handler for loading and saving definitions.

7 Summary and outlook

Calimero NG relies on a flexible, extensible and modular architecture. Introducing a network link abstraction according to the waist-line concept decouples network access protocol implementations from the rest of the library. This allows high level KNX related services to be written at an appropriate level of abstraction and rely on the link to handle medium specific issues.

Non-strict layering of components enables users to choose the level of abstraction that is appropriate for them. Components are grouped into feature bundles. Dependencies between these bundles are minimized to allow the pruning of library parts even after compile time. Together with the move towards J2ME as the required runtime environment, this enables Calimero to enter the world of resource constrained devices.

A network buffer reduces network load and allows client polled application designs. KNX network and device management functionality is now part of Calimero, and the client side of all published KNXnet/IP protocols is fully supported.

Moreover, licensing options have been extended. While the GPL remains in full effect and Calimero stays published open source, interested parties that have made a significant contribution can now obtain an additional license. This license allows including Calimero in a project without having to release its entire source code, but still requires modifications to Calimero to be made available. These licenses are individually negotiated.

Regarding future work, efforts are already being made to at least partially support the draft KNX XML schema [9]. This data encoding will be highly valuable for the exchange of configuration data with ETS. Also, additional console tool implementations will be provided.

Although the existing codebase would be very well suited to implement USB connectivity, this is currently not an option since communication via USB is not supported in current Java runtime environments. However, a BCU 2/FT 1.2 connection would be an interesting extension (although it would probably require the Java Standard Edition). In any case, we are looking forward to seeing Calimero develop – in close collaboration with its user community.

References

[1] KNXLive! project Website, https://www.auto.tuwien.ac.at/a-lab/knxlive.html

[2] KNOPPIX – Live Linux Filesystem On CD, http://www.knopper.net/knoppix/

[3] Calimero – EIBnet/IP Tunnelling (and more) for Java, http://calimero.sourceforge.net

[4] KNX AN033, Common EMI Specification, v03 AS and
KNX AN066, cEMI Adaptations, v03 AS

[5] KNX Handbook, Vol. 3, Part 8, Ch. 1–5, System Specifications: EIBnet/IP, v1.2 DV

[6] KNX Handbook, Vol. 3, Part 8, Ch. 2, System Specifications: EIBnet/IP: Core, v1.2 DV

[7] KNX Handbook, Vol. 3, Part 7, Ch. 2, System Specifications: Interworking:
Datapoint Types, v1.2 DV

[8] KNX KSG259: Identifiers, 20061003a

[9] KNX Handbook, Vol. 3, Part 8, Ch. 10, System Specifications: KNXnet/IP:
XML Data Encoding, v0.7 DP