Gorba

Common Software

Message Dispatcher

Architecture

Technical Description

Version:	0.2
State:	Draft
Classification:	Internal use only
Author:	WES
Creation date:	2015-05-11
Repository:	$/Gorba/Main/Common/Medi/Documents/

TD_MessageDispatcher.docx

Gorba AG
Sandackerstrasse
9245 Oberbüren
Switzerland

Table of contents

Modification management

Version	Date	Name	Dept.	Modifications	State
0.1	2015-06-05	WES	SW Dev	Initial Version	draft
0.2	2015-06-12	WES	SW Dev	Updated several chapters after feedback from EPT and LEF	draft

Review

Version	Date	Name	Dept.	Remarks

Release

Version	Date	Name	Dept.	Remarks

1 Introduction

1.1 Scope

This document is a technical description of the Message Dispatcher (Medi) software component. It gives all information to Gorba software developers to be able to develop with and for this component. This document is not intended to be a user manual.

The goal is to give a deep understanding of the architecture as well as the usage of Medi.
The configuration of Medi using medi.config is explained in more depth in TD_MediServerConfig.

1.2 Intended Audience

This document is written to be understood by Gorba software developers. Advanced technical skills are required.

1.3 Brief History

In 2011, Gorba was using the EventHandler protocol (later called Event Dispatcher – EDI) to exchange data between different applications. EventHandler was primarily used in the board computer, but also between Protran and Infomedia on the TFTs.

EventHandler had some major disadvantages:

- It was XML based and thus not very efficient; for this reason it was also not usable for the communication over mobile networks (e.g. GPRS).
- All message types had to be implemented in the EventHandler DLL, thus extending it was difficult and versions of different applications had to match.
- Configuration of EventHandler was done differently in every application.
- It was always necessary to have an EventHandler server through which the applications were communicating.
- EventHandler servers couldn't connect to each other, thus preventing more complex networks.

For these and other reasons, it was decided to create a new protocol that would be more flexible and future-proof.

Later the protocol should also replace the existing device-to-server communication protocols QNet (iqube) and ECI (VM.x).

2 Product Overview

2.1 Purpose

Medi is a message bus protocol optimized for the needs of Gorba applications. It is meant to be the only application-to-application protocol used when an embedded system (including TFTs with Windows Embedded) is involved.

2.2 Architectural Considerations

2.2.1 API

As Medi is being used in many Gorba software components, it is important to keep the API as backwards compatible as possible. Only source compatibility must be guaranteed; binary compatibility is not required since all applications are rebuilt completely in every build.

To make sure that internals of Medi can change while the interface stays the same, everything that is not intended for external use should be set "internal". This requirement does not apply to message types since they must be public to be accessible by BEC and XML serialization.

Only projects from namespaces inside `Gorba.Common.Medi` should be set "`InternalsVisibleTo`". This guarantees that changes to internal methods and classes only have an effect on the `Gorba.Common.Medi` solution and don't affect other components.

Whenever a part of Medi is completely independent of other components, it should be considered to create a separate DLL (and thus project) for this component.

2.2.2 Configuration

Medi configuration should, whenever possible, be done through the common medi.config file. This XML file is represented by the `MediConfig` class. If Medi needs to be extended, usually also additional parameters are required in the configuration. Those parameters should be added to the corresponding configuration class and they should always be kept backwards and forwards compatible – an old configuration file must always be readable by newer versions of Medi and the opposite should also be possible.

.NET XML serialization helps with the compatibility since it will ignore unknown elements and attributes, but one should never change element or attribute names or change the type of a certain value – if compatibility can't be guaranteed (e.g. converting an integer property to a string is ok, but the opposite is not).

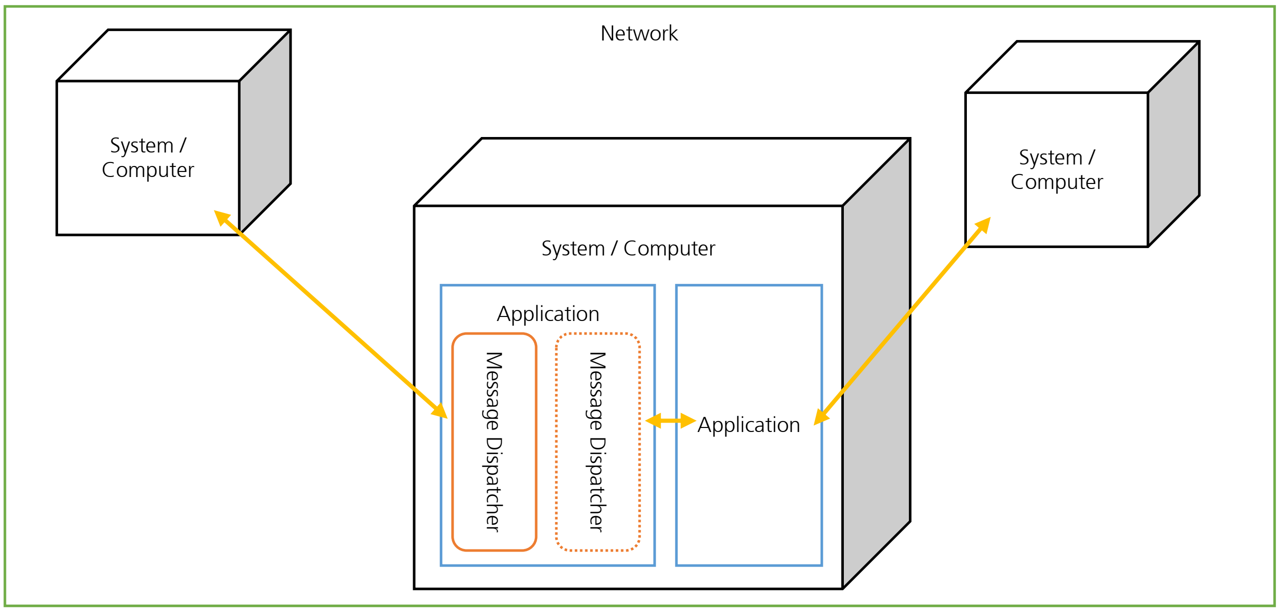
2.2.3 Compatibility

Besides the configuration file compatibility (see chapter 2.2.2), also any communication should be kept backwards and forwards compatible. This must be guaranteed so that one application can be updated without breaking the connectivity with other applications. Since the Gorba Update application also relies on Medi, it would be impossible to update any application without having to update every application in the entire network at the same time.

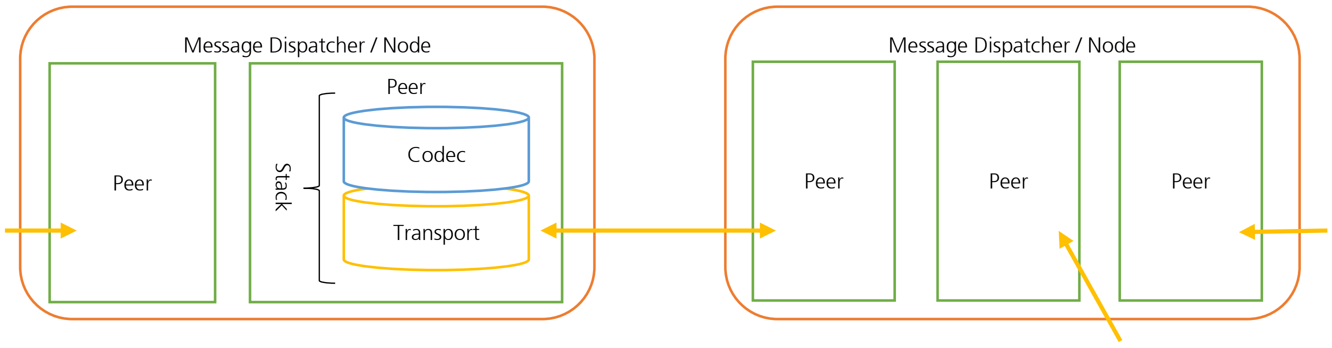
For example all codecs are identified not only with their type but also with their version and TCP transport provides a way to negotiate features that can then be used if both parties know about them (see chapter 6.1.1.1.1).

2.3 Important Terms

This document uses some general terms to explain parts of the Medi ecosystem. The following drawings show the relationship of those terms.



A Medi network consist of one or more systems (computers) that each have one or more applications running. Inside an application there is usually one Message Dispatcher, but it is possible to configure multiple (see chapter 3.1).



In terms of the Medi network, each dispatcher is also called a "node". Each node contains one or more peers that talk to another peer in the network. Usually peers inside the same node don't talk to each other using Medi but are directly connected through the message dispatcher.

The regular Medi peers (see chapter 4.1) consist of a stack with a codec and a transport on top of each other. It is actually the transports of different peers (on different nodes) that talk to each other.

3 Core Functionality

3.1 Creation of Message Dispatchers

There are two ways of "creating" an `IMessageDispatcher`:
- In regular applications, one just accesses the singleton `MessageDispatcher.Instance`
- If a message dispatcher must be created (and disposed) dynamically in code `MessageDispatcher.Create()` can be used

3.1.1 Singleton

Before ever accessing any functionality of the singleton message dispatcher, it needs to be configured using `MessageDispatcher.Instance.Configure()`. Afterwards it can be used for the entire lifetime of the application.

3.1.1.1 Configuration

Every time `MessageDispatcher.Instance.Configure()` is called, the entire Medi node is reconfigured, so it is strongly suggested to call it exactly once at start-up. Most applications will use the `AutoConfigurator` which will search for the `medi.config` file in the usual places.

It is possible to set the unit and application names when configuring Medi, but it is strongly suggested to use the predefined names, which are the machine name (unit) and the executable name without file extension (application).

3.1.2 Manually Created Message Dispatchers

When an application requires a Medi node for a certain time only and it needs to be configured from the software, one can use the method `MessageDispatcher.Create()`. This will create and configure a new instance of `MessageDispatcher` that can then be used to send and receive messages as well as accessing all other functionality.

3.2 Message Dispatching

The most important functionality of Medi is of course the dispatching of messages. From an API point of view, this is completely done through the `IMessageDispatcher` interface (implemented for example in `MessageDispatcher`).

The two methods `Send(MediAddress, object)` and `Broadcast(object)` can be used to send messages. `Broadcast(object)` is just a shortcut for using `Send()` with the broadcast address.

3.2.1 Transmittable Objects

A message can be any kind of object. It is strongly suggested to use simple POCOs (Plain Old C# Objects) without additional functionality. Also, messages should only use public properties (read-write). This guarantees that the messages can be encoded using both BEC (see 7.1) and XML (see 7.2) codecs.

3.2.2 Addressing

A message is always sent to certain address. Addresses are used for routing and can be either specific, multicast or broadcast (see 3.4).

3.2.3 Local Messages

If a message is sent to a local subscription (not through a connection to another node), the message is passed as-is. For this reason, messages should never be changed after sending them. Messages should also not be reused since this could lead to issues in the dispatching. POCOs are cheap to create, so just create a new message object every time. If this is not easily achievable, use an object and create a deep clone before sending it, then send the clone instead of the original object.

3.2.4 MediMessage

Every object that is being sent over Medi is encapsulated inside a `MediMessage`. This is simply a wrapper around the source and destination addresses as well as the message object itself (called "payload"). `MediMessage` objects are only meant to be used within Medi.

3.3 Subscriptions

To receive messages sent over Medi, a component needs to register itself with an `IMessageDispatcher`. A subscription is always done exactly for one given type at the time.

3.3.1 Subscribe

The method `Subscribe<T>(EventHandler)` is used for subscribing. The given delegate will then be called whenever a message of the given type arrives at the message dispatcher. Of course only messages that are sent explicitly (with a specific address) or implicitly (with a multicast or broadcast message) to the dispatcher are forwarded to the subscription.

3.3.2 Unsubscribe

Once a component doesn't need notifications anymore, it should always call `Unsubscribe<T>(EventHandler)`, giving the same type and delegate as parameters.

3.4 Routing

All messages sent through a message dispatcher are subject to routing. During routing, a message is sent to zero or more receivers which can either be local message dispatchers and/or remote Medi nodes that handle and/or forward the message again.

3.4.1 Addressing

A Medi address consists of two fields:
1. Unit: the name of the Unit (usually device or system name)
2. Application: the name of the application (usually the name of the executable)

Every address in a network of Medi nodes should be unique; if not, the results of routing might be unexpected.

Addresses are always written as `<Unit>:<Application>`, for example: `TFT-00-11-22:SystemManager`.

3.4.1.1 Local Address

The local address of every message dispatcher is available with `IMessageDispatcher.LocalAddress`. Usually one doesn't have to worry about this address since it is automatically used as the source address by the dispatcher when sending a message.

3.4.1.2 Address Types

Three different types of addresses can be used as the destination address when sending messages:
- **Specific:** contains a Unit and an application name and thus addresses only a single message dispatcher. Example: `TFT-00-11-22:SystemManager`

- **Multicast:** there are two different kinds of multicast addresses:
  - Unit-multicast: send the message to all applications on a given Unit. For this, the `Unit` property is set to the name of the Unit and the `Application` property only contains the wildcard character "*". Example: `TFT-00-11-22:*`
  - Application-multicast: send the message to a given application on all Units connected in the network. For this, the `Application` property is set to the name of the application and the `Unit` property only contains the wildcard character "*". Example: `*:SystemManager`
- **Broadcast:** uses the wildcard character for both Unit and application name and thus addresses all message dispatchers in the entire network. Example: `*:*`

3.4.2 Routing Table

To know how to reach which addresses, every node in the Medi network has a routing table. The table is updated when a new connection is created or closed, but also when routing updates are received from other nodes.

3.4.2.1 Routing Table Structure

The routing table contains for every entry the following information:

- **Address:** the address for which the entry contains the information
- **Session Identifier:** the way to know through which peer and which session a message should be forwarded. If this is the special `SessionIds.Local` then the local message dispatcher is responsible for this address. See chapter 5 for more information about sessions.
- **Hops:** the number of nodes that have to be passed to get to the address. This allows the routing algorithm to decide if a newly announced routing entry is better than the existing entry in the routing table. This is zero when the address is handled by the local message dispatcher and it is incremented by one for each hop.

Every address exists exactly once in the routing table, so there is no priority handling required for different entries.

3.4.2.2 Routing Updates

Routing updates are sent as list of `RouteUpdate` only from one node to an adjacent node. These messages are never sent any further since, if a routing table entry was updated, it would automatically create a routing update for all adjacent nodes anyways.

Routing updates are never sent back to the peer sending an update since this could cause infinite loops and doesn't make sense – all entries have lower hop counts on the sending node than on the receiving node).

3.4.2.3 Address Conflicts

If for some reason the same address exists twice in a network, the one with the lower hop count is used. This guarantees that a given address is always reachable with the shortest path, but it also means that if two nodes have the same address, any addressed message will only reach one of them.

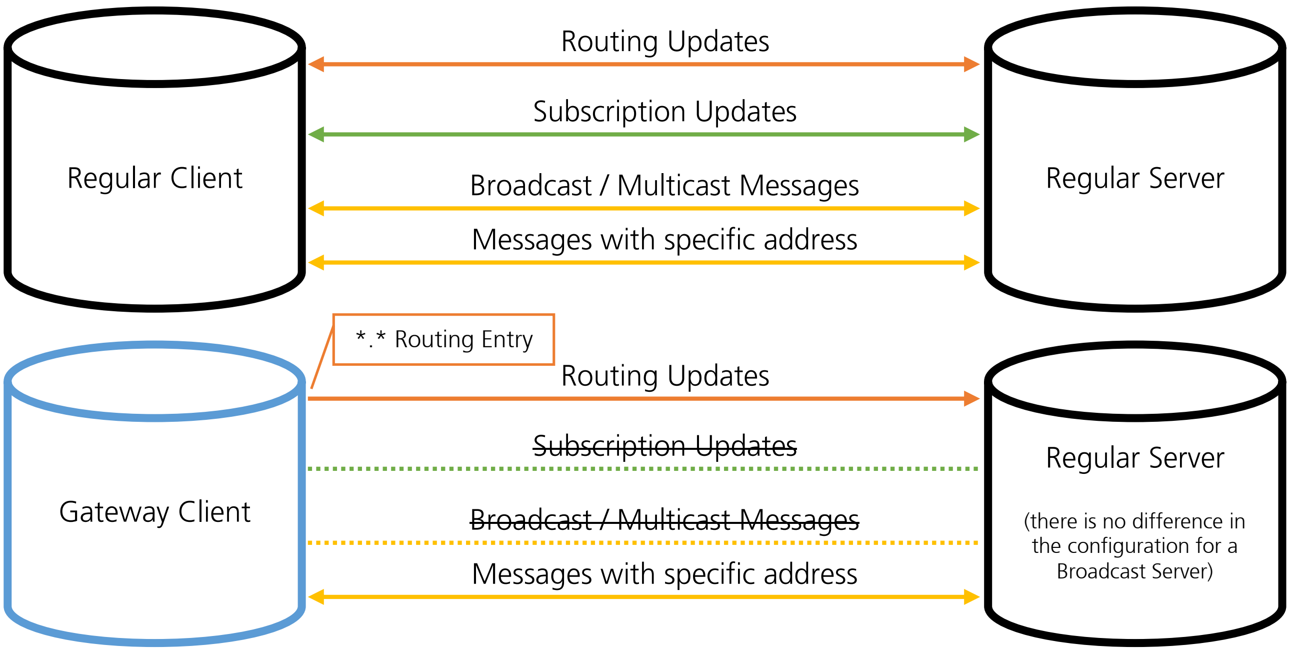
3.4.3 Gateway Connections

Since Medi supports both broadcast and multicast messages, there must be a way to limit the propagation of such messages in large networks. This is done by setting some connections to be "gateway connections." Such a connection always contains two ends: the gateway client and the gateway server.

Contrary to other kinds of networks (e.g. IP), a node can't be configured to be a gateway, but only a connection can be in gateway mode. To enable this, the client of a connection has to initialize the connection as being a gateway connection. This is done by setting `ClientPeerConfig.IsGateway` to `true`. This allows the same server to be a gateway for some connections and a regular peer for others.

Gateway connections have the following special behaviors:

- The gateway client:
  - Adds a routing entry for `*:*`, which means that any address not found in the routing table can be reached through this connection.
  - Continues to send routing updates to the gateway server.
  - Doesn't send subscription information to the gateway server.
  - Doesn't forward multicast and broadcast messages to the gateway server.
  - Forwards messages of all types for unknown addresses to the gateway server.
- The gateway server:
  - Doesn't send routing updates to the gateway client.
  - Doesn't send subscription information to the gateway client.
  - Doesn't forward multicast and broadcast messages to the gateway client.
  - Forwards messages of all types for specific addresses known to be reachable through the gateway client to it.


Regular Client
Routing Updates
Subscription Updates
Broadcast / Multicast Messages
Messages with specific address
Regular Server
. Routing Entry
Gateway Client
Routing Updates
~~Subscription Updates~~
~~Broadcast / Multicast Messages~~
Messages with specific address
Regular Server
(there is no difference in the configuration for a Broadcast Server)

3.4.4 Broadcast Subscriptions

If the connection between two nodes passes through a gateway connection, no broadcast or multicast messages can be received on either end. Usually this is the expected behavior, but there are circumstances where one node would like to receive such messages anyways for a certain amount of time. This can be achieved by using "broadcast subscriptions" which basically tells a node in the network to forward all messages of a given type to a given address.

Further details about how to use broadcast subscriptions can be found in chapter 8.3.

3.5 Management

Every node in a Medi network can provide management information. This information is organized in a tree and can be accessed locally and remotely. Medi Management is similar to SNMP (Simple Network Management Protocol).

Currently the management tree is read-only and has to be queried for updates – there is no automatic value update.

Medi itself already provides some information about its internal state. Other components can easily add more information into their own subtree.

The entire API can be found in the `Gorba.Common.Medi.Core.Management` namespace and the factory for accessing management trees is accessible through `IRootMessageDispatcher.ManagementProviderFactory`.

3.6 Log Observation

Medi provides a way out of the box to access the live log messages created on any node. This allows applications to display the log messages currently being created at a remote node.

Log observation doesn't allow to read past log messages, but only sends messages when they are created.

To prevent log message flooding there are two limitations:

- Log messages coming from the `Gorba.Common.Medi` namespace with a level below Info (i.e. Trace and Debug) are not sent to observers
- Log messages coming from any `ProducerConsumerQueue` with a level below Info (i.e. Trace and Debug) are not sent to observers

Internally log observation uses timers so that log messages are only sent to a requesting node for a limited time, unless a new request is made.

Logs can be observed locally by calling `MessageDispatcher.Instance.LogObserverFactory.LocalObserver` and remote logs are available in the same factory by using the `CreateRemoteObserver()` method. The log observer then has the possibility to define the expected level as well as registering to an event when a new message is created.

3.7 Named Dispatchers

Sometimes it is necessary that a component is reachable by a commonly known name or that a Medi node must impersonate something.

Therefore it is possible to create so-called named dispatchers that can have a different unit and/or application name. Those named dispatchers provide methods to receive and send messages; so they can be used like a regular dispatcher. A named dispatcher is created by calling one of the `IRootMessageDispatcher.GetNamedDispatcher()` methods.

Named dispatchers should be disposed of once they are not needed anymore. This will update the routing for the previously registered address.

An example for a commonly known name is the "fake" application "SystemManagementController" which is used on Units to send messages to the System Manager without knowing the application's name (it could be SystemManager, ServerManager or SystemManagerShell). Impersonation could be used in Comm.S to address legacy iqubes (they don't have a Medi address, but a QNet address).

4 Peers

Peers are one of the configurable parts of Medi. They allow the connection to and from other applications.

If a new messaging protocol needs to be implemented in Medi, this is the extension point where to start. If for example the QNet and ECX protocols were to be implemented in Medi (which is one way of fixing the communication gap between old and new protocols – the other being to add Medi to Comm.S), one should create a new peer for each of those protocols.

4.1 Medi Peers

The default peer to be used in Medi is simply called the "Medi Peer". There are actually two different classes implementing this, the `ServerPeerStack` and the `ClientPeerStack`.

Both support a stack consisting of:

- Codec (see chapter 7) on top of a
- Transport (see chapter 6)

The Medi peer implements all the stuff common to all kinds of codecs and transports. It is responsible for the routing to and from remote nodes (see chapter 3.4) as well as subscriptions (see chapter 3.3) from those nodes. It also handles the reception of Ping messages and the sending of the corresponding Pong response.

The `MediPeerBase` class exists so that it would be possible to later implement a peer that is based on the known Medi messages, but might not consist of Codec and Transport.

4.2 Event Handler Peers

The Event Handler peer exists for two reasons:

- Legacy compatibility
- Connections to third party applications

The Event Handler (or Event Dispatcher, EDI) protocol is the predecessor to the Medi protocol and is (as of this writing) still used in board computers and in older TFT installations (e.g. BusTerminal).

Since the Event Handler protocol is very simple to implement (just send and receive XML structures over a TCP socket), it is also used in several projects to connect third party applications with Gorba applications. Usually it is used to exchange Ximple messages.

Both client and server implementations are available since in some scenarios our application might be serving Event Handler connections or might be connecting to an EventHandler server or EdiServer.

On the board computer a newer version of EDI was created to simplify the detection of messages. It contains start and end tags (`<1>` and `<2>`) to delimit messages. This behavior (called "tagged mode") is implemented as a configurable option in the Event Handler peers as well.

The Event Handler peers use the standard .NET XML serializer and are therefore not directly compatible with BEC (see chapter 7.1). To prevent issues with unknown types, the Event Handler peer should always be configured in the application that creates or consumes the given messages.

5 Sessions

In Medi peers all connections are assigned a unique session identifier. This identifier is used to recognize existing sessions to reuse resources and to ensure proper reception of all messages (in the right sequence).

Every transport implementation has its own way to generate and exchange session identifiers.

5.1 Reuse of Resources

Since both ends to a connection know the identifier for the session, it is possible to reuse resources by reusing the same session identifier.

This can be used in state-less transports where there is no direct logical connection between two peers, or in stateful transports where the logical connection (e.g. a TCP/IP socket connection) might be interrupted. The TCP transport (see chapter 6.1.2) uses session identifiers to recognize when a client reconnects a server. Like this a session can be reused while the underlying TCP/IP connection might be interrupted for example when the device is in an area without reception (Wi-Fi or cellular network).

Currently the following functionality relies on sessions:

- Routing: routing tables are not changed when a TCP/IP connection is lost and later reconnected (before the Session Timeout); this reduces the number of routing messages transferred through Medi
- BEC mappings: String and schema mappings of BEC (see chapter 7.1.2) are stored for each session and thus strings and schemas don't have to be retransmitted when a connection is lost and later reconnected

5.2 Frame Handling

Frame handling provides a way to retransmit possibly lost frames and identify duplicate frames. When a connection between two peers is lost, both ends don't know which messages might have been lost at that time. Therefore similar to TCP (but in the application layer) Medi supports frame handling.

Each message is called a frame and gets additional information:

- The identifier (consecutive number) of the current frame
- Last received frame identifier

Both peers keep a list of all sent messages with their identifier and only deletes them once the opposite peer acknowledges them.

When a session is "restarted", each peer starts immediately sending previously not acknowledged messages to the other end. That peer will then ignore the frames it has already received and only process the previously unknown ones.

It is important to note that (contrary to TCP) there are no pure "acknowledge" messages. They are always sent "piggyback" on a message that is to be sent anyways. Since in most transports there is a way of checking if the connection is still alive ("keep-alive"), those messages play the role of acknowledgements if the general message flow is unidirectional.

6 Transports

To send messages from one node to another, two Medi peers are connected with a common transport. Transports allow to easily change the underlying connection without having to touch the upper layers (codecs and message dispatching).

All transports implement `IMessageTransport` which provides an interface to send and receive regular messages (which have been serialized to buffers using a codec) as well as stream messages. All transports must support regular messages, but supporting stream messages is optional.

Most transports have a client that connects to a server, so one peer must be configured using a `ClientPeerConfig` and the other using a `ServerPeerConfig`. If in a type of transport both ends are equal (so there is no "client" and "server"), then all peers should be configured using `ServerPeerConfigs`; an example for this is the (experimental) File Transport (cf. chapter 6.2).

6.1 Stream Transports

Most transports use .NET streams to exchange information. Therefore one common transport base class framework was created to handle such streams.

Stream transports support both regular and resource messages (also called stream messages) and use two separate connections to separate these channels. This allows for small messages to be transferred quickly on the first channel while a larger resource message might occupy the second channel for some time. To identify a channel after connecting, the two transport layers use a handshake (see below).

6.1.1 Common Features

Most of the code in stream transports is common to all kinds of transports and therefore most features are also common. In this chapter you find all those features.

6.1.1.1 Handshake

When a stream client connects to a stream server, the two transport implementations use a handshake to agree on the available features and to identify sessions (see chapter 5).

The sequence of messages is as follows (if successful):
1. Client connects to server
2. Client sends handshake message to the server
3. Server sends agreed handshake message to the client
4. Client sends acknowledge to the server
5. Regular communication starts with the chosen codec

If the server can't understand the message from the client or doesn't support it, it sends a negative acknowledge instead of step 3 and closes the connection immediately.

If the client can't understand the message from the server or doesn't support it, it sends a negative acknowledge instead of step 4 and closes the connection immediately.

6.1.1.1.1 Handshake Message Structure

The handshake message looks as follows: `<{features},{codec},{session},{remote}>` where:
- `{features}` is a bit-field (written as decimal digits) of the following features:
    - 1: this stream is used to transmit regular messages
    - 2: this stream is used to transmit resource messages
    - 4: unused

   - 8: every message is prepended with a frame and acknowledge number (framing)
   - 16: currently unsupported, but in the future reserved for stream compression
   - 32: currently unsupported, but in the future reserved for stream encryption
   - 64: the connection is a gateway connection (see chapter 3.4.3)
- `{codec}` is a combination of one character codec type and one digit version number, currently the following codec identifiers are used:
   - B: BEC (only version 1 supported)
   - X: XML (only version 1 supported)
- `{session}` is a common identifier shared by both peers on a connection (see also chapter 5)
- `{remote}` is a local identifier which must be unique for each node; remote and session together guarantee that each connected session can be uniquely identified (since two session identifiers provided by different nodes might not be unique within a given node)

If the handshake message structure has to be changed in the future, it is important that the first part (“`<{features},`”) remains the same, so that the new structure can be deduced from the features bit-field.

6.1.1.1.2 Acknowledge Message Structure

The acknowledge of the handshake has the simple structure `<{status}>` where `{status}` is one of:
- 0: Failure: this value only exists for backwards compatibility and means a general failure
- 1: Success: positive acknowledgement: “everything is OK, proceed”
- 2: Version Not Supported: the three least significant bits (values 0..7) are not supported
- 3: Internal Error: something went wrong during handshake
- 4: No Connection: this is never found in an acknowledge, but used inside a client peer when it couldn’t connect to the server
- 5: Stream Error: this should never be found in an acknowledge, but is used within a peer when the connection was broken
- 6: Bad Content: the receiving peer doesn’t understand the structure of the handshake message
- 7: Codec Not Supported: the receiving peer doesn’t know how to handle the codec or its version defined in the handshake (this happens for example if one peer is set to BEC and the other to XML)

If a peer receives a value other than 1 (even if currently unknown), it should immediately close the connection. If the error is 2 or 7, it shouldn’t even try to reconnect again since those failures are permanent.

6.1.1.2 Message Streams

The message stream is used to send regular `MediMessage`s to the opposite peer. The message stream contains only payload data after the handshake. There are no begin or end message markers. This has the advantage of making this transport very efficient, but requires the upper layer (codec) to care about message boundaries. A message buffer returned when reading from a stream transport may contain parts of a message or one or more messages.

6.1.1.3 Resource Streams

Internally sending a resource message is split into multiple messages, some of them sent over the regular message channel, others sent over the resource channel.

1. The sending peer sends a `ResourceStateRequest` with the hash of the resource to the receiver (over the message channel)
2. The receiving peer replies to that request with a `ResourceStateResponse` containing:
   a. The hash of the resource (as received)
   b. A flag indicating if the receiving peer already knows the entire resource

   c. The number of bytes already available in the receiving peer (this allows to continue an aborted transfer of a large resource)
3. Upon reception of the response the sending peer sends a `ResourceStateAck` with the hash so that the receiving peer knows that its response has arrived (currently this is only used to make sure frames are not repeated and thus resources resent when a connection was lost after a transfer)
4. If the resource is not yet completely available at the receiving peer (as reported by the `ResourceStateResponse`), the following header is sent on the resource channel:
   a. 1 byte "0": Header start
   b. 8 bytes: Length of this header (excluding a & b)
   c. 1 byte: Header version (currently "1")
   d. Variable length string (see .NET `BinaryWriter`): Source Medi address unit name
   e. Variable length string: Source Medi address application name
   f. Variable length string: Destination Medi address unit name
   g. Variable length string: Destination Medi address application name
   h. 8 bytes: Offset from where the data is sent (0 if the entire resource is sent)
   i. 8 bytes: Total length of the resource (not starting from the offset!)
   j. Variable length string: hash of the (entire) resource
5. After the header the unknown part (or the entire resource) is sent:
   a. 1 byte "1": Content start
   b. 8 bytes: Length of following data (excluding a & b) – this is now only the length starting at the offset provided in the header!

6.1.1.4 Stream Factory

To make the creation of new kinds of stream transports easy (like the Bluetooth transport below), usually it is enough to implement the `IStreamFactory` interface and provide it through (almost empty) client and server implementations. See the classes in `Gorba.Common.Medi.Bluetooth` as an example.

6.1.2 TCP Transport

The TCP transport is completely based on the common stream transport framework and simply uses .NET sockets for communication.

The TCP server is always bound to any local IP address and listens to connections with a backlog of 10 slots.

The TCP client supports domain name resolution before connecting a regular socket.

6.1.3 Bluetooth Transport (experimental)

The Bluetooth transport was mainly done as a proof of concept for the extensibility of the stream transport framework. In general it has been tested with a small dongle as well as a built-in Bluetooth radio in a laptop. The functionality is based on 32FEET.NET a Bluetooth (and IrDA) implementation which would also work on .NET Compact Framework (but was never tested).

The protocol uses the following service UUID: `A655A737-612D-4DB4-AA30-53AC0A565779`

Since there is no way of resolving names with Bluetooth, one needs to know the server's Bluetooth address to connect to it; so using this transport makes only sense in an environment where the address is discovered using directly the 32FEET.NET API and then Medi is configured dynamically to use the found address.

Bluetooth transport does not handle pairing in any way, this is done directly by the Bluetooth stacks on the devices that are connecting to each other and is thus not covered here.

6.1.4 Pipe Transport (testing only)

To simplify unit testing another transport was created: the Pipe transport. It is a very simple implementation of in-memory pipes: a client connects to a server and messages sent from one peer to the other are directly passed as buffers.This allows to have two independent peers on the same computer without having to use (public) TCP ports and waiting on the slower TCP/IP communication.

Receiving methods never get called directly by sending methods since this could cause deadlock situations – therefore most calls are routed through the thread pool.

6.2 File Transport (experimental)

The file transport is yet another experiment to validate the Medi architecture. It is an incomplete implementation of the basic transport interfaces and allows to exchange messages using files. The files can either be located in a local folder, a network folder or on an FTP server.

This transport should never be used in productive environments, but can be seen as an example of how to write a transport that is not based on streams and that has equal peers, so no client-server setup.

7 Codecs

Medi supports multiple codecs (or “encodings”). Each of them has its advantages and disadvantages.

Since codecs are not compatible between each other, an entire network should always use the same codec. It is currently for example not possible to send a Ximple message from an application using XML encoding through a Medi Server that supports XML and BEC to an application which is configured to use BEC. Reason for this is that types unknown to an intermediate Medi node (e.g. Medi Server) are simply passed along in binary form – and binary BEC data can’t be converted to XML and vice versa.

BEC is the preferred codec, unless there is an important reason for choosing XML.

Because codecs can change between different versions of Medi, but within the same codec should still be compatible, a codec is identified by its type and a version number. Currently for all codecs the version number is 1, but in the future it will be possible to implement new features or make “breaking” changes that are only used between peers that support a certain newer version.

7.1 Binary Enhanced Coding (BEC)

The Binary Enhanced Coding (BEC) was developed mainly to support slow connections, but it has proven to be a reasonable solution in all situations. BEC is proprietary to Gorba and is only meant to be used between Gorba applications.

The main advantages of BEC are:

- Binary data format uses less data than XML
- String tables allow to only send the same string once (in both directions)
- Schema exchange helps keeping messages compatible between different versions (similar to .NET XML serialization)
- Inheritance is supported without additional metadata (contrary to .NET XML serialization which requires attributes)

7.1.1 Encoding Sequence

BEC traverses the object tree of an object and serializes all its properties recursively. For each newly encountered type, a schema is created (see below) and all newly encountered strings are sent before the actual message.

7.1.2 Mapping

In the following chapters there are two kinds of mappings: string mapping and schema mapping. A mapping is the assignment of a unique identifier to a value. Mappings are done for each session. They support reverse-mapping, which means that the two directions of a communication channel only require the mapping to be sent once over the channel.

Example: A sends the string “abc” to B, referring to it with the identifier 5; then B will refer to “abc” with the identifier -5. This means that B never has to transmit a mapping for “abc.” This improves the throughput of the codec even more.

7.1.3 Primary message types

BEC has three different types of primary messages exchanged:

- String Mapping Messages are used to exchange strings with their unique identifier
- Schema Messages hold the information about the serialization structure of types
- Regular Medi Messages contain the actual payload

The sequence of messages for a single encoded object is always:

1. Zero or one String Mapping Message
2. Zero or more Schema Messages
3. Exactly one Medi Message

7.1.3.1 String Mapping Messages

All newly mapped strings are put into a single String Mapping Message. This allows to refer to known strings by their integer unique identifier. The following two strings are never transmitted and have predefined constant identifiers:

- Null value has always the identifier 0
- An empty string has always the identifier 1

7.1.3.2 Schema Messages

All newly found types are described using one Schema Message per type. A schema is actually a regular object (implementing `ITypeSchema`) that is also serialized using BEC. To prevent issues at start-up (how do you transmit the schema of a schema?), the following types have predefined constant identifiers and can therefore never change their structure (except if a new version of the codec is used):

- ID 0: Used for null values
- ID 1: `TypeName` (name of a possibly unknown .NET type)
- ID 2: `BecSchema` (schema of the structure of a regular type)
- ID 3: `BuiltInTypeSchema` (built-in types like primitives, strings, Type, …)
- ID 4: `DefaultTypeSchema` (schema referring to a type and its schema)
- ID 5: `ListTypeSchema` (schema of lists and arrays)
- ID 6: `EnumTypeSchema` (schema of enums)
- ID 7: `SchemaMember` (property of a `BecSchema`)
- ID 8: `SchemaMessage` (message containing one schema)
- ID 9: `StringMappingMessage` (message containing multiple string mappings)

7.1.4 BEC Schema Structure

As mentioned above, BEC creates, caches and exchanges schema information to know the exact structure of exchanged objects. The `SchemaFactory` is responsible for creating a schema for a given type and caching it for the lifetime of the application. A single schema contains information about exactly one object; it doesn't describe the entire object tree but rather references other schemata if needed. There is no notion of inheritance in a schema, thus a schema always contains all properties of its type and base types.

Each kind of schema is represented by a type implementing `ITypeSchema`. The following kinds exist:

7.1.4.1 BecSchema

The `BecSchema` contains the name of the type and all its fields and properties. A property is represented by its type schema and name.

To be similar to XML serialization regarding the members serialized, the BEC schema has the following modifications to the list of fields and properties:

- If the member has an `XmlIgnoreAttribute`, it is ignored
- If the member has an `XmlElementAttribute` with a non-empty "Name", this name is used instead of the original member name
- If the member has an `XmlAttributeAttribute` with a non-empty "Name", this name is used instead of the original member name; the name is also (in both cases) prefixed with an "@" to distinguish it from elements and regular properties

7.1.4.2 BuiltInTypeSchema

The `BuiltInTypeSchema` refers only to the type name which must be known to the encoding and the decoding application. The following types are currently known:

- All primitives (`byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `double`, `float`)
- Strings
- Array of bytes
- Type
- `Guid`
- `DateTime`

7.1.4.3 DefaultTypeSchema

The `DefaultTypeSchema` is a placeholder schema for any type. It is used for properties and allows subclasses to be used instead of the class or interface defined in an object.

Example: class B extends A and C has a property of A; the property A (of C) will be described using the `DefaultTypeSchema` which then allows both A and B to be used as values for the property. This is a strong advantage over XML serialization where subclasses must be defined using .NET attributes.

7.1.4.4 ListTypeSchema

The `ListTypeSchema` contains the type of the list as well as the type of its elements. Lists are types implementing `IList` (which includes arrays). The type of elements is in most cases described using a `DefaultTypeSchema`, which again allows to use subclasses as elements of a list.

7.1.4.5 EnumTypeSchema

The `EnumTypeSchema` contains the type of the enum as well as the type of the underlying value.

7.1.5 Serialization Process

This chapter describes step by step how an object is serialized using BEC. These steps are more or less also followed when de-serializing an object.

1. The first value that is serialized is the identifier of the schema for the object being serialized. If the object being serialized is null, then the identifier 0 is used. See chapter 7.1.5.5 for the binary format of an integer.
2. If the object is not null, it will be serialized depending on its type (see below)

7.1.5.1 Regular Object

A regular object is simply serialized by serializing its properties in the order they appear in the schema according to the process described in the entire chapter 7.1.5; this means starting with the schema identifier of the value of the property (which doesn’t have to be the same as the type of the property) and so on.

7.1.5.2 Boolean

A Boolean is written as an 8-bit value (0 representing false, 1 representing true).

7.1.5.3 Byte

A byte is written as a regular 8-bit value.

7.1.5.4 Signed Byte

A signed byte is written as a regular 8-bit value.

7.1.5.5 Signed Integers

Signed integers (`short`, `int` and `long`) are serialized using a short format which will in many cases (especially for small numbers) uses less bytes than simply using the binary representation of the number:

- The highest bit of every byte written defines if there are more bytes coming.
- The second highest bit of the first byte is set if the value is to be inverted (2's complement) to represent a negative number.

Examples (green = more bytes, red = negative, other colors = value):

Value	Binary	BEC Serialized
19	`0001'0011`	`0001'0011`
-55	2's complement of -55 = 54: `0011'0110`	`0111'0110`
165	`1010'0101`	`1010'0101 0000'0010`
6515	`0001'1001 0111'0011`	`1011'0011 0110'0101`
45861	`1011'0011 0010'0101`	`1010'0101 1100'1100 0000'0101`

7.1.5.6 Unsigned Integers

Unsigned integers (`char`, `ushort`, `uint` and `ulong`) are serialized using a short format which will in many cases (especially for small numbers) uses less bytes than simply using the binary representation of the number:

- The highest bit of every byte written defines if there are more bytes coming.

Examples (green = more bytes, other colors = value):

Value	Binary	BEC Serialized
19	`0001'0011`	`0001'0011`
165	`1010'0101`	`1010'0101 0000'0001`
6515	`0001'1001 0111'0011`	`1111'0011 0011'0010`
45861	`1011'0011 0010'0101`	`1010'0101 1110'0110 0000'0010`

7.1.5.7 Floating Point Numbers

`Float` and `double` are serialized using their IEEE 754 representation, either using 4 or 8 bytes.

7.1.5.8 Array of Bytes

An array of bytes is serialized as its length (as unsigned integer, see chapter 7.1.5.6) followed by the bytes as regular 8-bit values.

7.1.5.9 String

If BEC finds a string value, it checks with its internal mapping whether that string is already known; if not, the next available identifier is taken. The identifier is serialized as a signed integer (see chapter 7.1.5.5).

7.1.5.10 Type

A .NET type is serialized using its full name (namespace and type name) as a string (see chapter 7.1.5.9).

7.1.5.11 Guid

A .NET `Guid` object is converted to its byte array equivalent (16 bytes) and then serialized as an array of bytes (see chapter 7.1.5.8).

7.1.5.12 DateTime

A .NET `DateTime` object is serialized with its tick count as a signed integer (see chapter 7.1.5.5) followed by its kind in a single byte.

7.1.5.13 Lists and Arrays

Lists and arrays are serialized by serializing their length as a signed integer (see chapter 7.1.5.5) followed by each element serialized according to the process described in the entire chapter 7.1.5; this means for each element starting with the schema identifier of the value of the element and so on. If the list or array is null, the value `-1` is serialized.

7.1.6 IBecSerializable

If a type requires special handling during serialization and deserialization, it is possible to implement the `IBecSerializable` interface. Use this interface only when there is absolutely no other solution.

Be very careful when creating the `BecSchema` returned by `GetSchema()` since it must match exactly the structure of the data, otherwise intermediate nodes will not be able to handle such messages which will cause the entire deserialization process for the given connection to fail.

7.1.7 Frame Handling

If Frame Handling is enabled, each message has the following two values appended:

- Unsigned integer (see 7.1.5.6) send frame number
- Unsigned integer (see 7.1.5.6) acknowledge frame number

7.2 XML

The XML codec uses the .NET XML serialization to serialize MediMessage objects. It is therefore very simple and the TCP streams are (mostly) human readable.

To make parsing easier, each message is terminated with a null byte (0x00).

7.2.1 Frame Handling

If Frame Handling is enabled, each message is prefixed with a 17 byte ASCII header:

```
<8 byte hex send frame number>-<8 byte hex acknowledge frame number>
```

7.3 Dynamic Codec

The dynamic codec can be used in servers only and allows to choose the codec automatically during the handshake. As mentioned above, a conversion between different codecs is not possible, but at least an application can speak with different clients using different codecs for each of them.

8 Services

Medi provides a simple way of accessing shared services that are related to Medi. Most services need to be configured using the `MediConfig.Services` configuration property and are then available through `IRootMessageDispatcher.GetService<T>()`.

Services are started and stopped by Medi during configuration respectively disposal.

In the future it is very well possible to add more services. Services need to implement `IServiceImpl` and the corresponding service interface. If a service is placed in its own project, of course the DLL must be set to "Internals Visible" in Medi.Core. Please do not place any services outside the `Gorba.Common.Medi` namespace – this is not meant to be a general service discovery component.

8.1 Resource Service

The resource service is the most complex of all services. Since it partially also interacts with transports, some classes (besides configuration) are located in `Gorba.Common.Medi.Core.Resources` while all others are in the separate DLL `Gorba.Common.Medi.Resources`.

The resource service is responsible for storing and managing resources in a system. It also supports sending resources from one node to another. Since multiple applications might want to access the same resources on the same system (computer), there are two different kinds of resource services:

- The `LocalResourceService` stores resources in a (local) storage and provides direct access to those files and is configured using the `LocalResourceServiceConfig`.
- The `RemoteResourceService` requires a `LocalResourceService` running on the same system and mostly just forwards requests to the local service. It is configured using the `RemoteResourceServiceConfig`.

8.1.1 Functionality

8.1.1.1 Register Resources

Registering a resource adds the given file to the resource store and saves its information in the data store. A resource is uniquely identified by its MD5 hash and can always be retrieved at a later time by providing the `ResourceId` (which is its hash). If the resource is no longer required, it can be removed from the service as well. If an already registered resource hash is registered again, this is not seen as an error, but the new file is simply ignored (or deleted, depending on the provided parameters).

8.1.1.2 Export Resources

When an application wants to have a copy of a registered resource, it can simply export it from the store. This will always create a copy of the resource and no longer keep track of the exported file.

8.1.1.3 Checkout Resources

The concept of checkout and check-in is not used anywhere for now (instead export is used), but it allows to use registered resources outside of the store without creating copies. Especially for large resources this can be a big improvement since each resource is only available exactly once on the disk.

The disadvantage of this method is that if a checked out resource is deleted or modified (and therefore never checked in again), the resource service has "dead" references. This can lead to major issues if other components depend on the availability of the given resource.

8.1.1.4 Send Resources

Resources can be sent from one node to another simply by providing the reference to the resource as well as the address to which to send the resource. It is not possible to multicast or broadcast a resource. For the resource to pass from the first to the last node in the communication, all intermediate node (as well as the end node, of course) need to have a resource service configured.

Sending resources is optimized in a way that a resource is only sent from one node to another if it doesn't exist yet on the second node.

A sent resource will be stored on every intermediate node to prevent having to resend it later again.

8.1.1.5 Send Files

Sending a file is similar to sending a resource with two main differences:

- The file does not need to (but can) be registered before sending
- The resource created from that file is defined as "temporary" and will thus be deleted immediately after the transfer; this is also the case for all intermediate nodes

The remote node will then raise the `FileReceived` event in which the file must be handled.

8.1.1.6 Request Files

If one node wants to download a file from a remote node, it can request the file over Medi. The file is then transferred from the remote node to the local node using the "send file" mode (see 8.1.1.5) which results in the `FileReceived` event being risen.

8.1.1.7 Resource Locking

Internally resource services make sure that a resource can only be deleted when it is no longer required by any service on the local system. Resources are marked for deletion when a lock exists while removing the resource and are only deleted once the last lock on the given resource is released.

8.1.2 Local Resource Service

The local resource service stores all information about the resources in a "data store" and the resources themselves in a "resource store". Both stores can be configured independently allowing to have specific behavior depending on the use case.

8.1.2.1 Data Store

The data store stores all information required by the resource service in a "database". The default implementation (`FileResourceDataStore`) keeps track of that information in XML files called ".rin".

8.1.2.2 Resource Store

The resource store is the place where the actual resources are kept. Usually it is a `FileResourceStore` which simply stores files in a local directory.

8.1.3 Remote Resource Service

The remote resource service requires a local resource service to run on the same system. It will then forward most requests directly to that service which does the actual handling of files.

8.2 Port Forwarding Service

The port forwarding service provides a way to tunnel regular TCP traffic through a Medi connection. This can for example be used to access a port on a remote unit to which there is no direct connection (i.e. the connection goes through the Background System).

The basic framework of port forwarding would also support other kinds of data transfer like UDP or serial ports; but at the time of this writing, this is not implemented (as there is no need for it at the moment).

8.2.1 Configuration

Like all services, the port forwarding must be enabled by configuring it in the Medi configuration. There are no configuration parameters required since port forwardings are created at runtime using the `IPortForwardingService` interface.

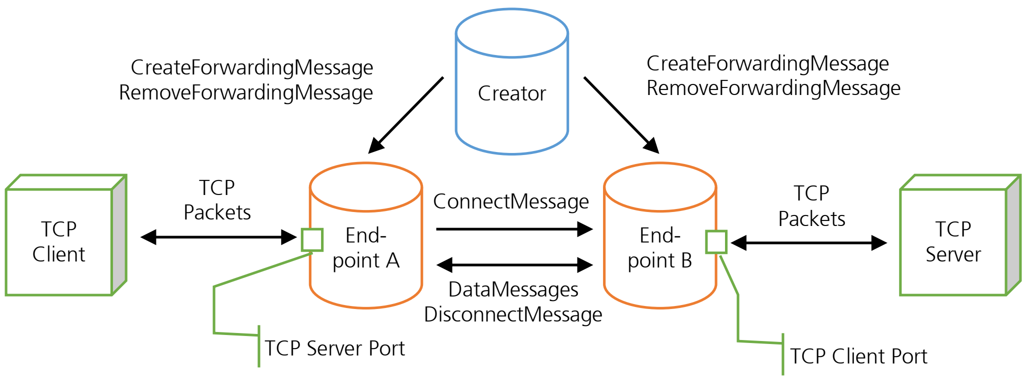
8.2.2 Setup of a Port Forwarding

In the creation of a port forwarding, three roles are involved:

- The creator
- The first endpoint (A)
- The second endpoint (B)

All these roles can be on the same or on different nodes. This allows for example node A to create a forwarding going from B to C. Often the first two roles are on the same node. All involved nodes need to have the port forwarding service enabled.

The creator is the one calling `IPortForwardingService.BeginCreateForwarding()`, the two endpoints are the roles involved in the actual forwarding of data. The following picture shows the logical connections between the roles and the messages exchanged:



8.2.3 Stream Forwarding

In some applications it might be desirable to directly access a remote TCP port using a .NET Stream. This can be achieved with the method `BeginConnect()`. The corresponding end method will return a stream which can then be used to read from and write to the remote port.

8.3 Broadcast Subscription Service

Contrary to other services, the broadcast subscription service is automatically available in a Medi peer, so there is no configuration required.

As explained in chapter 3.4.4, there is sometimes the need to get multicast and broadcast messages even if they are not transmitted over a gateway connection.

The broadcast subscription service sends a request for a certain type of message to a remote broadcast subscription service. The remote service will then forward (by changing the destination address) the requested messages. Those messages are then received like regular messages on the message dispatcher from which the broadcast subscription service was requested.

To prevent load issues when connections are suddenly dropped, a broadcast subscription only has a limited lifetime (currently 5 minutes) and needs to be renewed regularly. This guarantees that if a connection is lost, a broadcast subscription only "survives" for a short amount of time.

In general, broadcast subscriptions should only be used for a short period of time and not for regular operation of the system.

9 Appendix A: Glossary

Term	Description
Medi	Message Dispatcher component described in this document
BEC	Binary Enhanced Coding – codec optimized for minimal message size