Table of Contents

1. General description

1.1. Terms and abbreviations

ID	Identifier
SDL	Smart Device Link
TA	Transport Adapter
TCP	Transmission Control Protocol
TM	Transport Manager
UID	Unique identifier

1.2. Transport layer features

Several definitions:

1) '*Connection establishing* with the application or device:
   - Means that transport layer creates a physical interconnection for sending and receiving 'handshake' data.
   - Does not mean that this application or device shall be marked as "connected" for the user.
   - May have two options:
     - One-time connection: when the data related to a device is cleared up after disconnection
     - Persistent connection: when the information about the device is stored even in the power off mode.

2) '*Connection closing*':
   - Means that connection is not terminated until 'goodbye' data transmission is finished (all accumulated data is sent to / received from the application).
   - Physical disconnection happens when there is no more data left for transmission.

Table #1: List of features provided by transport layer

N	Feature
1.	**Searching for the applications on devices using all available transports**
a)	Notifying on applications list found on device
b)	Notifying on search finished in adapter
c)	Notifying on search done
d)	Informing for which transport the connection has failed
2.	**Connection establishing with the user-selected application for specific device using specific transport**
a)	Notifying on connection successful performance
b)	Notifying on connection failure
3.	**Connection establishing with the user-selected application for specific device on all transports supported by the application**
a)	Notifying on connection successful performance
b)	Informing for which transport the connection has failed
4.	**Connection establishing with all applications for the user-selected device (using specific or all supported transports)**
a)	Notifying on connection successful performance

b)	Informing for which application (and transport) the connection has failed
5.	**Connection establishing with all applications on all user-selected devices (using specific or all supported transports)**
a)	Notifying on connection successful performance
b)	Informing for which application, device and/or transport the connection has failed
6.	**Connection establishing with the application that requests the access to the system by all available transports**
a)	Notifying on connection request from the application on unknown device
7.	**Remembering the device (device is marked in a special way and is kept in the system until the user asks to clear all data connected with it)**
8.	**'Forgetting' the device (removing the mark from the device and removing the device from internal storage)**
9.	**Enabling/Disabling SDL visibility for devices (similar to BT visible/invisible)**
10.	**Connection closing for specific application on specific device on specific transport**
a)	Notifying on successful disconnection
b)	Notifying on disconnection failure
11.	**Connection closing for specific application on specific device on all transports**
a)	Notifying on successful disconnection
b)	Notifying on disconnection failure
12.	**Connection closing for specific device on specific transport**
a)	Notifying on successful disconnection
b)	Notifying on disconnection failure
13.	**Connection closing for specific device on all available transports**
a)	Notifying on successful disconnection
b)	Notifying on disconnection failure
14.	**Connection closing for all devices on all transports**
a)	Notifying on successful disconnection
b)	Notifying on disconnection failure
15.	**Transmitting data to a connected device**
a)	Transmitting data with the use of priority
b)	Notifying on successful data transmission
c)	Informing which part of transmitted data stream has failed
16.	**Receiving data from a connected device**
a)	Processing received data with the use of priority
b)	Notifying on successful data receipt
17.	**Reconnecting the unexpectedly disconnected application (device)**
a)	Accumulating data destined for unexpectedly disconnected device
b)	Transmitting all accumulated data in correct order if the application has reconnected
c)	Informing that the accumulated data has not been sent if there is no connection established after the timeout.
18.	**Restoring previous state**
a)	Saving the internal information about the connected devices in persistent memory before the shutdown
b)	Loading the information about connected devices from persistent memory
19.	**Reconnecting applications configured to auto reconnect on power on**
20.	**Automatically selecting transport according to priority for multi-transport applications (if appropriately configured)**
21.	**Automatically switching the data flow for unexpectedly disconnected transport to another transport for multi-transport applications (if appropriately configured)**
22.	**Providing current state information**

a)	Listing the connected devices providing the list of applications for each device
b)	Listing the existing connections providing the information on their status
c)	Listing the available transports

1.3. Transport Level Structure

The *Figure 1* demonstrates the structure of the Transport Level:

- Transport Manager has no limitations on transports number.
- Each transport has no limitations on devices number.
- Any device may be connected through unlimited number of transports. In this case each connection established between the application on a device and the Transport Manager has the unique ID.

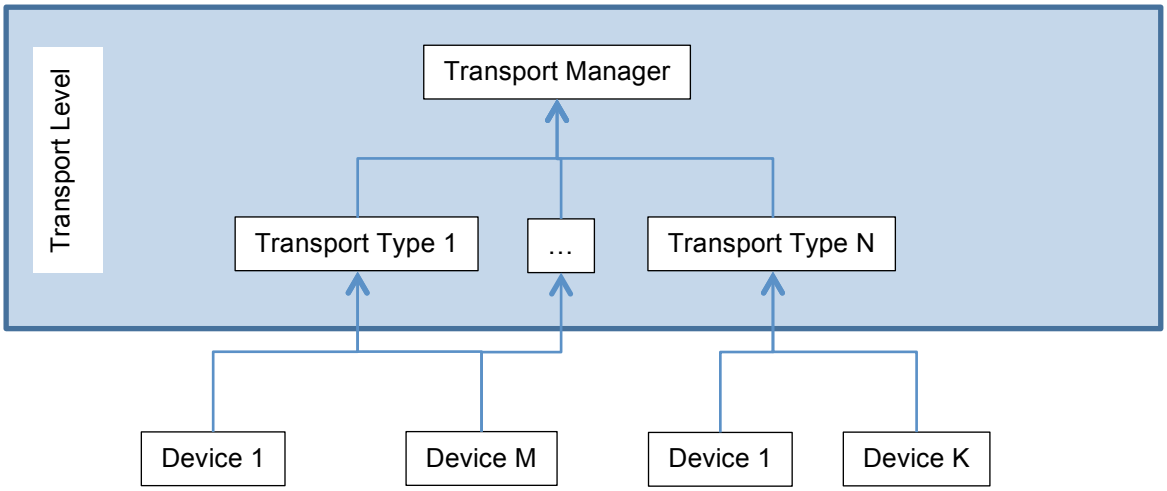

Figure 1: Transport Level Structure Diagram

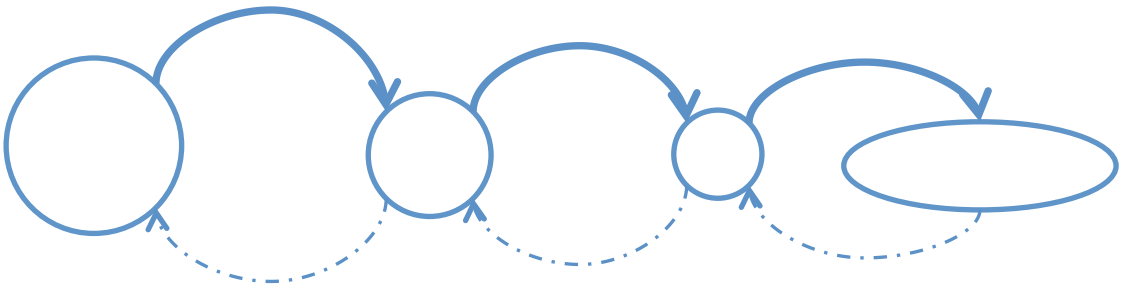

Figure 2: Transport level control and data flow (downstream – to device, upstream – from device)

The *Figure 2* demonstrates the data flows:

- Each abstraction level communicates only with the next and the previous abstraction level.
- Direct communication of two abstraction levels that are not the direct neighbors is prohibited.

For example:
The “Upper Level” cannot access the “Transport Adapter (TA)”
The “Transport Manager (TM)” cannot access the device and so forth.

The structure of the Transport Manager and the Transport Adapter is described in the below section 1.3.1 and section 1.3.2 correspondingly.

1.3.1 Transport Manager Structure

The following diagram represents the structure of the Transport Manger.

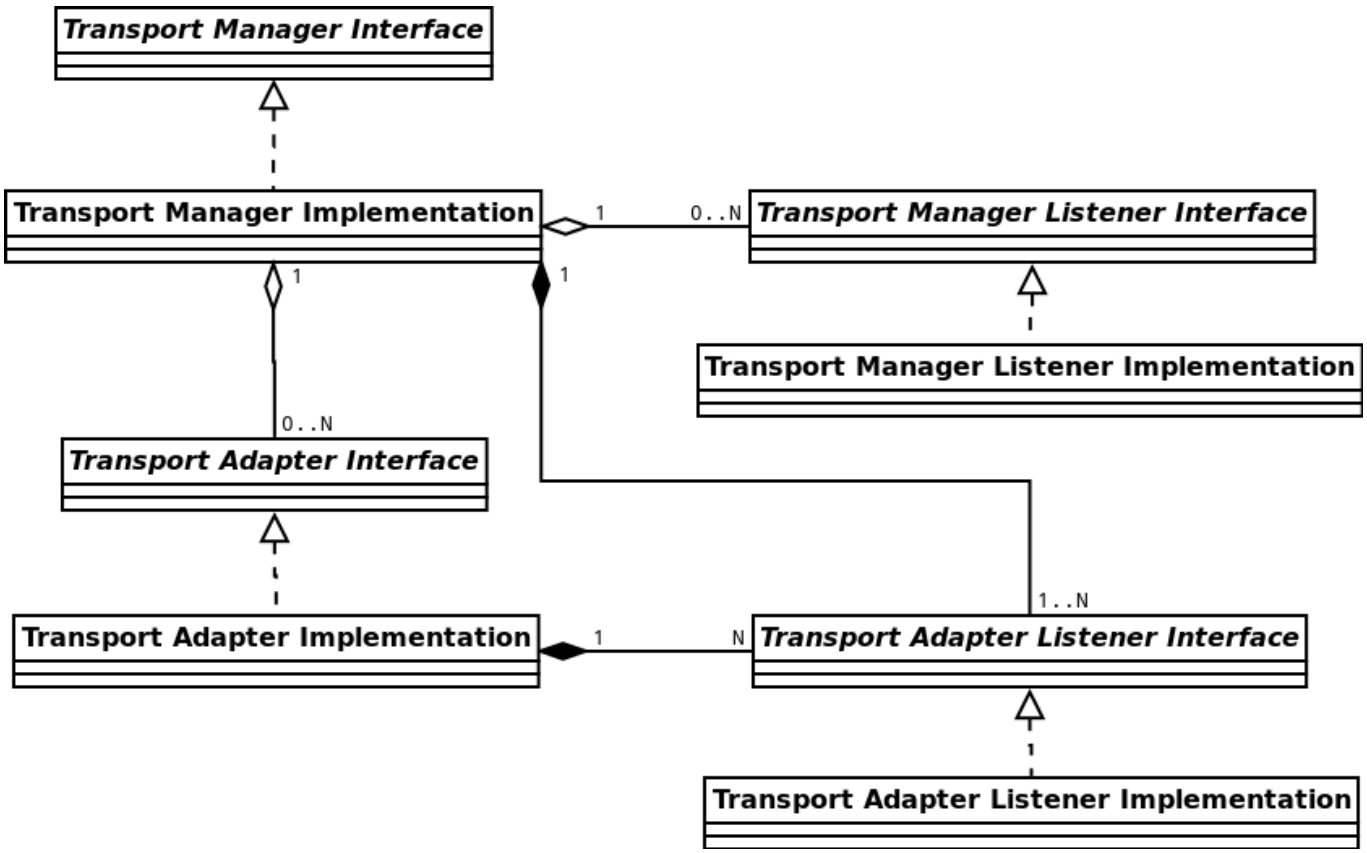

Figure 3: Transport Manager Class Structure Diagram

1.3.2 Transport Adapter Structure

The following diagram represents the structure of the Transport Adapter.

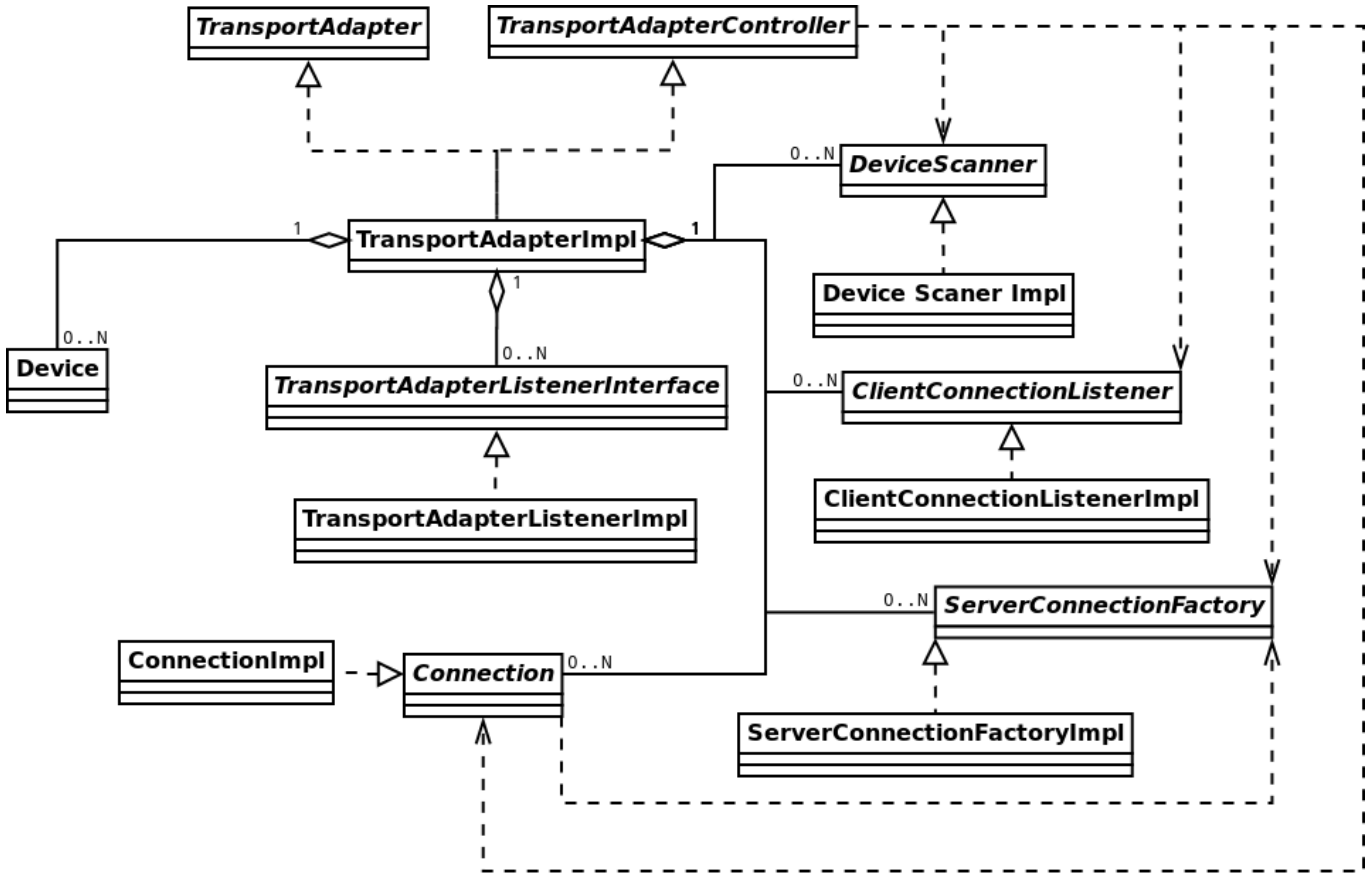

Figure 4: Transport Adapter Class Structure Diagram

1.4 Operation Examples (Message Sequence Chart)

1) New device search.

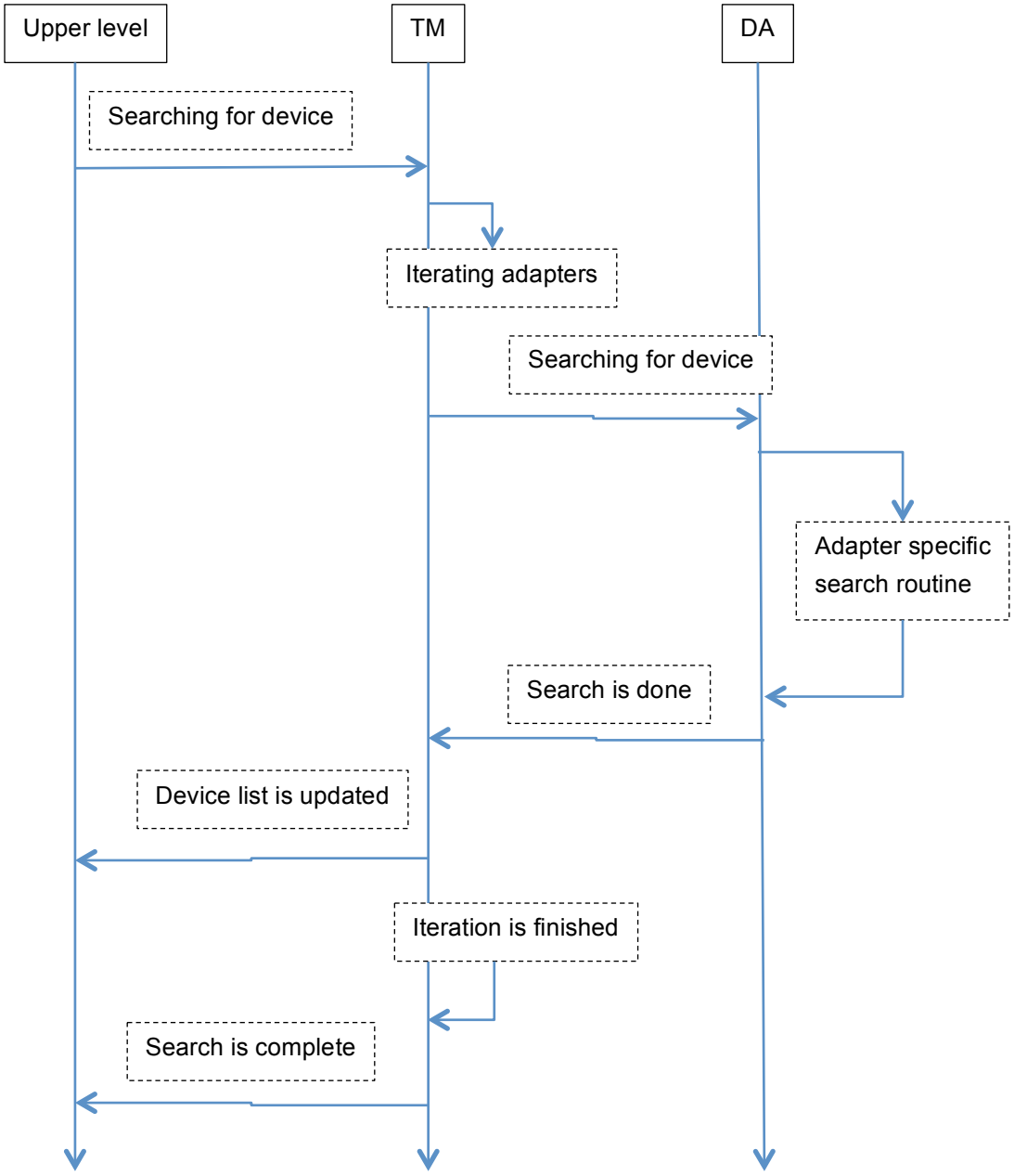

Figure 5: Transport Manager Message Sequence Chart for a New Device Search

2) Device-originating connection.

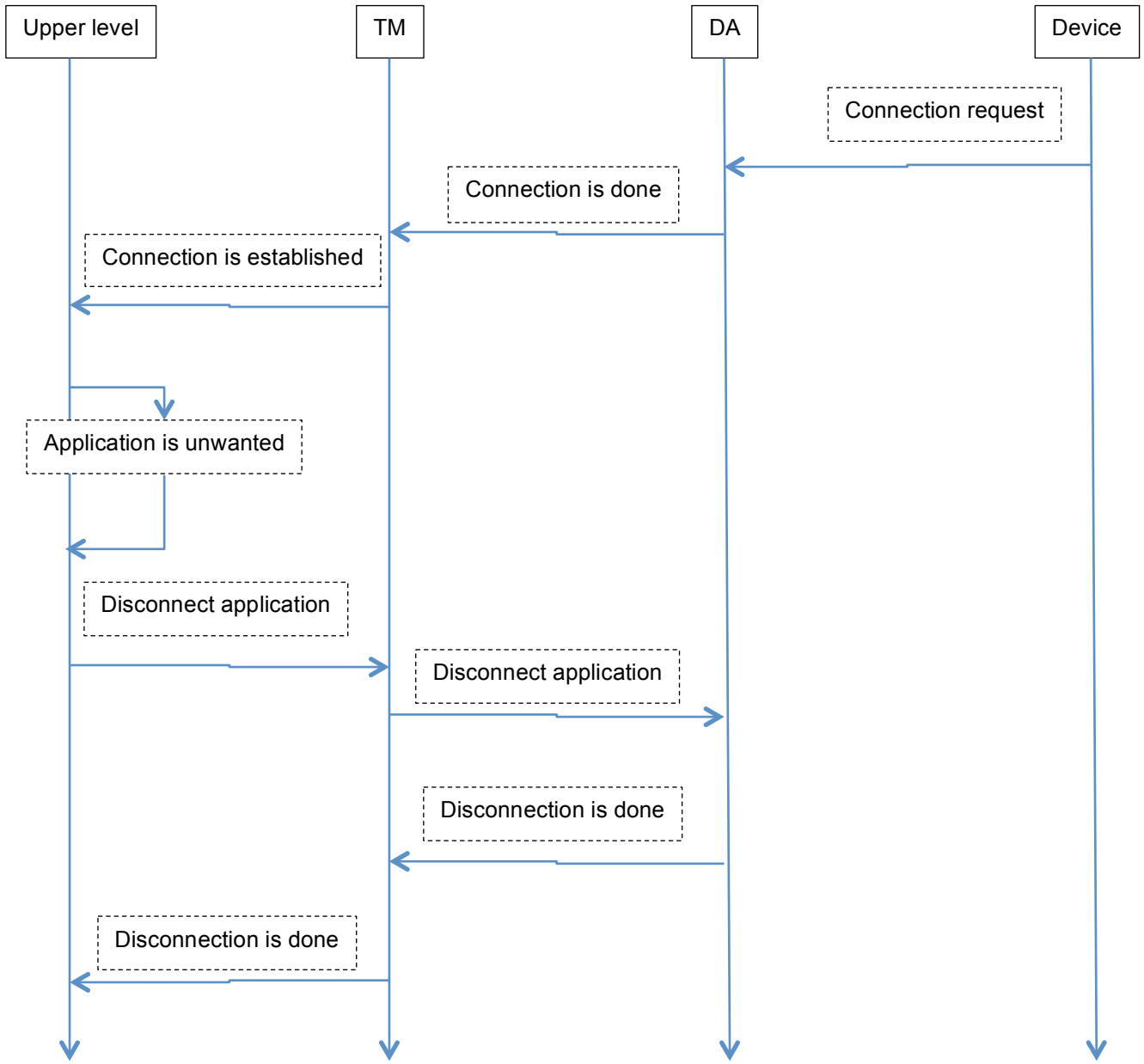

Figure 6: Transport Manager Message Sequence Chart for Device-Originated Connection

3) Connection close command

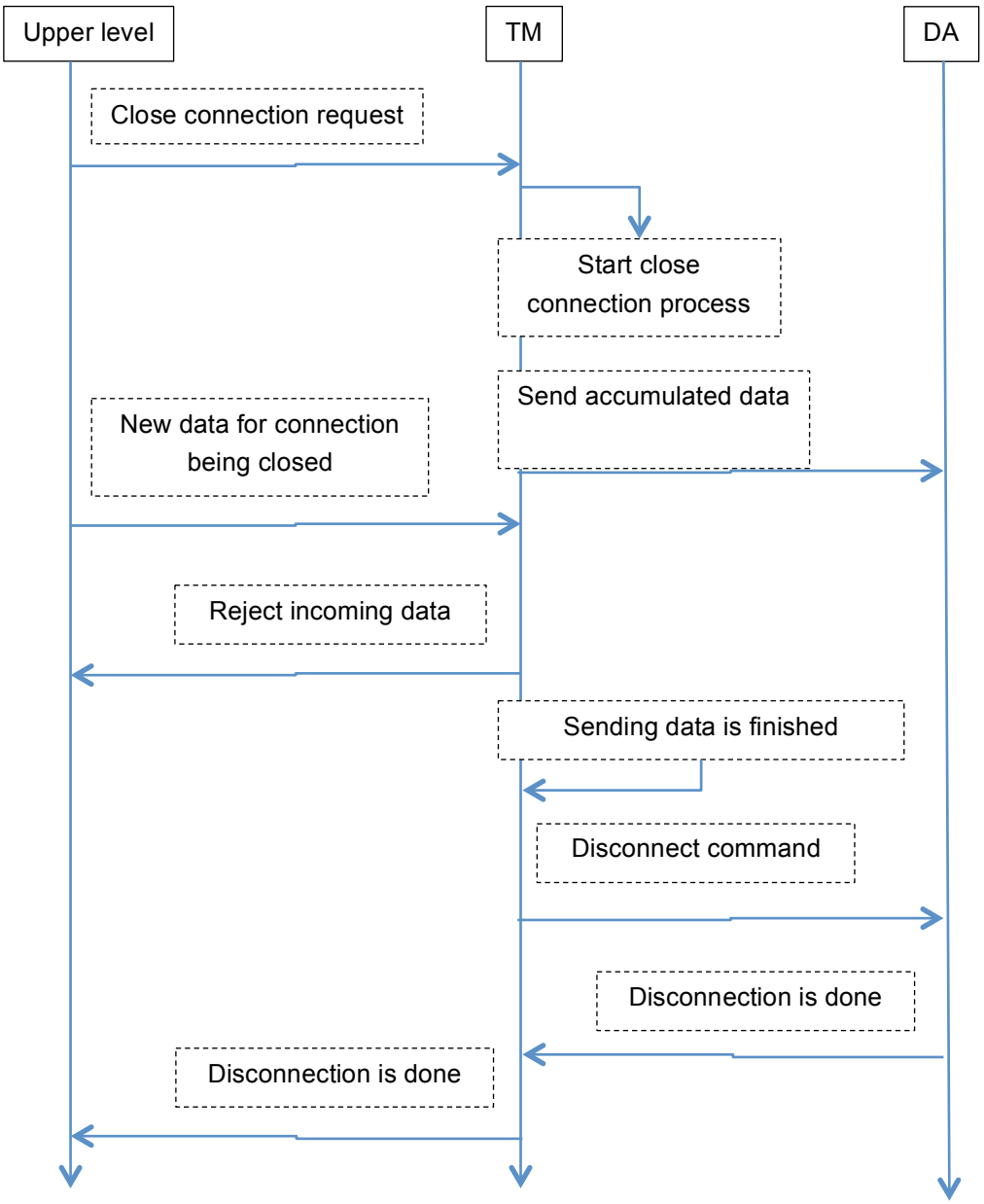

Figure 7: Transport Manager Message Sequence Chart for Connection Close Command

2. Transport Manager (TM) Usage

2.1. General Processing Description

1) TM initialization:
   - Every time SDL starts working it calls the creation and initialization of TM (the default or developer-defined one).
   - TM uses singleton pattern and the instance of TM is created and initialized while being retrieved. And the pointer is provided to the developer.
   - If TM is initialized once it must not be initialized again.
   - The developer may initialize the customized TM by calling the appropriate 'init()' function:
     - If the customized TM is based on default implementation, 'init()' must NOT be called twice for one and the same TM
     - If the customized TM is created from scratch, it is up to developer to choose the initialization mechanism.
   - During the initialization process TM creates two threads:
     - For processing the message queue of commands from the upper level
     - For processing the events coming from devices.
   - Also TM creates and initializes all available Transport Adapters (the default and/or customized ones).
   - If appropriately configured TM may load the information about previous state and perform necessary actions (e.g. reconnect the last connected application).
   - TM becomes initialized:
     - When its internal threads are created and are ready for working (i.e. the 'init()' function is serial and if it returns the control, the initialization has completed successfully and TM is ready for working).
     - And it does not watch whether the Transport Adapter(s) has initialized by this time:
       a) If any of TAs failed to initialize, this is informed into the log file. TM answers with erroneous messages to the requests related to such TA.
       b) If there are NO TAs (either not initialized or not defined), TM answers with the error messages to all of the requests.

2) TM structure:
   - *TM Core*: it is a TransportManagerImpl class. It contains the processing mechanisms (e.g., for procedures of sending/receiving the data, for connecting/disconnecting procedures, etc.).It does not have the embedded default Adapters and Listeners. And it would not work apart of the Wrapper.
   - *TM Wrapper*: it is implemented in TransportManagerDefault class. It adds the default Transport Adapter(s) and Listener to the Core completing the fully-featured functionality.

3) TA initialization:
   - The default Adapters are created and initialized by the default TM Wrapper.
   - The customized Adapters should be initialized by the developer himself:
     - TM is initialized first
     - Then the init() function for the custom Adapter is called
     - Then the initialized custom TA and the Listener are added to the initialized TM.
   - Transport adapter in its turn creates and initializes all available workers.
     - 'Search Device' worker creates a separate thread and waits for the 'Start' command.
     - 'Client Listener' creates a separate thread and waits for connections from devices.
     - 'Server Connection' worker executes all actions on caller's thread (does not create a thread).

4) Getting started:
   - When the initialization is complete, TM starts waiting for:
     - The user`s command
     - The device connection.
     - Resumption from “last state” singleton (if such resumption is specified via profile)
   - When one of the above happens, TA:
     - Creates a separate thread for each connection with device’ application(s)
     - Notifies TM on connection created.
   - When the connection is established, the Upper Level:
     - May start a handshake routine with the application and then notify the user about the application is connected.
     - May close the connection by sending an appropriate command to TM if the application or device is unwanted.

5) Errors in TM:
   - ‘*Immediate*’ error:
     - When TM is not able to execute any command it will immediately return the appropriate error code. For example: when `connect(app_id)` is called and TM is not initialized yet this type of error occurs.
   - ‘*Postponed*’ error:
     - TM is able to execute a command and there is some error occurred in downstream.
     - The component, where the error has occurred, sends the appropriate information to TM.
     - Then TM provides a notification to the Upper Level.

6) Messages in TM:
   - TM is ready to send and receive data after the connection is opened,
   - Sending messages:
     - Each message destined for the device is posted into the message queue.
     - The message is removed from the queue after it is successfully sent to the device.
     - The message(s) is returned to the caller if for some reason TM is not able to send it (e.g., unexpected device disconnection),
   - Receiving messages:
     - TM redirects messages from the Transport Adapter to the Upper Level via notification mechanism.

7) Connection identifiers:
   - TM uses the pair “`Device ID`” and “`Application ID`” for accessing the application on a device and internally for connection establishing.
   - `Device ID`:
     - Stands for global unique identifier based on MAC address for network adapters and MAC address like for BT.
     - In the default implementation `Device ID` is logically split into two parts:
       a) Internal device ID – that is a MAC address string
       b) External device ID – that is an integer value.
     - When the new MAC address is found, the integer value is assigned to it correspondingly (starting from 1 and incremented with every new assignment).
     - For persistent connection,
       a) When the device is marked as “known”, the correspondence of internal and external IDs is stored (even after power off) until the user explicitly asks to “forget about <this> device”.

b) When the user requests to 'forget' the known device, the MAC address may be assigned with the new integer value on being connected for the next time.

- For one-time connection this correspondence is not saved in long term storage. And if connected later the same MAC address may get the new integer value in correspondence.
- It is not defined what happens if two devices with the same MAC address would connect.

- `Application ID`: is the application unique identifier. It is the incremental integer value assigned and used internally in TM.
- `Connection ID`:
  - Connection is represented with a unique pair of "`Device ID`" and "`Application ID`".
  - Connection ID is a system wide unique identifier (incremented integer value) assigned to each new connection.
  - This ID is used for sending/receiving data and for closing the connection.
  - There are no certain rules to define how exactly this ID is assigned.

8) Connection closing:
   - When TM receives '`close connection`' or '`disconnect`' commands it tries to finish sending accumulated messages and then closes the connection.
   - If connection is lost TM will drain all accumulated messages and confirm connection closure.

9) SDL shutting down:
   - When SDL is going to shutdown, TM clears all objects which it has created (e.g. default transport adapters).
   - Objects created by developer (e.g. developer's transport adapter or listener) are not removed by TM. The developer must take care of destructing his customized objects by himself.

2.2. Default TM Instance Creating

For creating the instance of TM with default configuration it is necessary to use the following code:

```
#include "transport_manager.h"
#include "transport_manager_default.h"

{
…
  TransportManager* transport_manager =
    TransportManagerDefault::instance();
…
}
```

Initialization of TM may take some time due to threads creation and initialization routine. After all is set and done Transport Manager is ready to be used. Till that time any command will be rejected with the error code "NOT INITIALIZED" in return value.

TM implementation uses singleton pattern, thus only one instance of TM will exist at a time. Nevertheless, this is not the rule and may be changed in custom TM.

2.3. Adding Custom Listener to TM

1) A Listener allows monitoring the events that take place in TM.
2) The number of TM Listeners is not limited and may be zero.

3) These listeners are provided to TM and are used by any module that needs to receive the notifications from TM.
4) The list of Listeners that are called upon any event occurred in TM is stored in TM.
5) TM does not create the own listener by default.
6) Custom Listener:
   - Its logic should be related with implementation of the module that uses this Listener.
   - Should implement Transport Manager Listener Interface.

To set up the Listener the following code should be used:

```
#include "transport_manager.h"
#include "transport_manager_default.h"

class MyTransportManagerListenerImpl1 : public TransportManagerListener
{
//implement here entire interface
}

{
…
  TransportManager *tm_impl = TransportManagerDefault::instance();
  TransportManagerListener *my_tm_listener = new MyTransportManagerListenerImpl();
  tm_impl->addEventListener(my_tm_listener);
…
}
```

To remove the Listener `removeEventListener` should be used.

2.4. Adding Custom Transport Adapter to the Default TM

1) About TA in general:
   - TM may contain zero to N Transport Adapters (TM with zero Adapters returns the erroneous messages for all of the requests).
   - Each Adapter corresponds to specific type of transport (e.g. Bluetooth, LAN, USB, etc.).
   - TA implements transport specific search, connect, disconnect and data transfer routines.

2) Several instances of TA:
   - It is allowed to have several adapters of the same type in one TM.
   - The results of using several instances of default TAs are not defined.
   - It is developer's responsibility to create a custom Adapter that operates well under conditions of using several instances of it.

3) Adding the Custom TA:
   - The simplest way to add a custom Transport Adapter is to derive the existing implementation of TCP or BT Adapter, adjust some behavior in derived class and add it to TM:

```
#include "transport_manager.h"
#include "transport_manager_default.h"

{
…
  //note: the default implementation of TCP adapter is used in this example.
  //developer is free to create his own implementation of transport adapter from
scratch
  TransportAdapter* my_transport_adapter =
    static_cast<TransportAdapter*>(new TcpTransportAdapter);
  my_transport_adapter->init();//note: TA must be initialized before getting
instance of TM.
  TransportManager *tm_impl =
    TransportManagerDefault::instance();
  //note: when custom TA will be added to TM, the default listener will be
assigned to it automatically
  tm_impl-> addTransportAdapter(my_transport_adapter);
…
}
```

4) Initialization:
   - By default TA is initialized during TM initialization.
   - If TA needs to be added to the TM already initialized, this TA should be initialized first.

Notice:

- TM has a TCP adapter by default. Adding the new instance of the same Adapter may result the unexpected behavior (because default Adapters are not designed for working in such configuration).
- The developer must change the standard behavior of TCP adapter to eliminate the potential problems.

2.4.1. Custom Transport Adapter implementation from scratch

If default implementation of adapter is not suitable for particular case one can create new adapter from scratch:

- Create custom class that implements transport adapter interface. Note that custom TA shall use defined interface TransportAdapterListener for notifying Transport Manager.
- The instance of transport adapter listener will be set by Transport Manager automatically when custom transport adapter is added. If custom adapter does not store its listener then adapter will not be able to notify Transport Manager about events such as OnDataReceiveDone or OnConnectDone.

```
#include "transport_adapter.h"

class MyTransportAdapterImpl : public TransportAdapter
{
// implement all interface functions here
// use TransportAdapterListener interface to notify transport manager about event
happening
}
```

- Add the following code where the Transport Manager is initialized

```
#include "transport_manager.h"
#include "my_transport_adapter_impl.h"

{
…
  TransportAdapter* my_transport_adapter =
    static_cast< MyTransportAdapterImpl*>(new MyTransportAdapterImpl);
  //note: TA must be initialized before getting instance of TM
  my_transport_adapter->init();
  TransportManager *tm_impl =  TransportManagerDefault::instance();
  //note: when custom TA will be added to TM, the default listener will be
assigned to it automatically
  tm_impl-> addTransportAdapter(my_transport_adapter);
…
}
```

All internal logic implementation is up to developer. Developer is responsible to implement:

- Communication with device
- Notification of state changes
- Error handling and so forth.

2.5. Adding a New Listener to the Transport Adapter

1) A Listener allows monitoring the events that take place in TA.
2) The number of TA Listeners is not limited.
3) The list of Listeners that are called upon any event occurred in TA is stored in TA.
4) Custom Listener for TA:
   - The developer can add the Listener to TA through the customizing procedure only. I.e., the developer needs to create his own TA on the base of the default one, and then to add the Listener to it.
   - Should implement Transport Manager Listener Interface.

Note!

1) Working with TA Listener by-passing the TM is dangerous and may lead to the asynchronous behavior of TM and the Adapter.
2) The custom Listener should be added only together with the new custom Adapter and/or the new custom Transport Manger.

To set up the Listener the following code should be used:

```
#include "transport_manager.h"
#include "transport_manager_default.h"

class MyTransportAdapterListener :public TransportAdapterListenerImpl
{
//customize listener here if necessary
}
…
  //note: the default implementation of TCP adapter and the default implementation
of adapter listener are used in this example.
  //developer is free to create his own implementation of transport adapter from
scratch
  TransportAdapter* my_transport_adapter =
    static_cast<TransportAdapter*>(new TcpTransportAdapter);
  my_transport_adapter->init();
  TransportManager *tm_impl =
    TransportManagerDefault::instance();
  TransportAdapterListener* my_ta_listener =
    new MyTransportAdapterListener(tm_impl);
  my_transport_adapter-> addListener(my_listener)
  tm_impl-> addTransportAdapter(my_transport_adapter);
…
}
```

The developer should implement "`TransportAdapterListener`" interface with the custom logic before the Listener can be used.

2.6. Creating TM with Custom TAs Only (with No Default Adapter)

If for some reason the default Adapters are not good enough they can be replaced with developer's defined Adapter(s). To do this the developer

1) Must implement the Adapter's Interface using own logic
2) Provide this new adapter to TM:

```
#include "transport_manager.h"
#include "transport_manager_impl.h"

class MyTransportManager : public TransportManagerImpl {
  virtual int init();
  virtual ~MyTransportManager();

  MyTransportAdapter *my_adapter_;
  explicit MyTransportManager(const TransportManagerAttr &config)
    : TransportManagerImpl(config),
      my_adapter_ (nullptr){
  }
public:
  static MyTransportManager* instance();
};
//note: the implementation of all methods above is not defined here just for
making the code look simplier
//Obvious MyTransportAdapter should be created and initialized somewhere.
{
…
  TransportManager *tm_impl = MyTransportManagerImpl::instance();
…
}
```

More detailed information on custom Transport Adapter creation is provided in section 2.8.

2.7. Transport Manager Customizing

1) Basic information:
   - TM is responsible for all complex logic and decisions, while Transport Adapter is a primitive entity that operates only with transport specifics.
   - Communication interface between TM and TA:
     - TM sends a command to TA.
     - If TA is unable to execute this command it returns the error code not processing the command.
     - Otherwise, TA starts executing the command. Then TA notifies TM on executing completion by using the appropriate callback function.

2) Two queues are used as a fundamental of TM:
   - Message Queue: for commands coming from the Upper Level.
   - Event Queue: for events coming from devices.

3) Customizing TM, the developer:
   - Should implement transport manager interface.
   - Should use the Transport Adapter Interface and the Transport Listener Interface for making his implementation work with default Adapters and Listeners.
   - Has two options:
     - Creating the TM from scratch.
     - Deriving from the default implementation:
       a) If not particularly changed, the default Adapters and Listeners will be used.

2.8. Transport Adapter Customizing

1) Basic information:
   - TA is highly adaptable to any specific of a real transport.
   - TA consists of
     - So called *worker classes* that perform a single operation (e.g., device search)
     - *Controller* that
       a) Accumulates event handling from all worker classes,
       b) Controls the state of all internal data and
       c) Notifies the Upper Level via callbacks
     - *Internal data structures* that contain the information about the device, the connection and other necessary information.

2) Workers of TA:
   - *Device Scanner*:
     - Implements transport dependent search procedure initiated by the appropriate command from the Transport Manger.
     - It is developer's responsibility to implement this worker.
     - It may be absent for transport types that do not support searching.
     - When the device is found (and in the current default implementation when all the devices are found) this worker:
       a) creates a notification and directs this notification to the Controller
       b) The Controller notifies TM using the TA Listener
       c) TM receives the notification and sends a command to TA for connecting all available applications
       d) TM and TA perform a chain of notifications

e) TM notifies the Upper Level using TM Listener with the information on each application connected: the connection ID, the application name, the device name.

- *Client Connection Listener*:
  - Implements the transport dependent connection that was originated by device.
  - It is developer's responsibility to implement this worker.
  - If transport does not support such ability this worker may be absent.
  - Working procedure:
    a) This worker waits for the connection of a mobile device.
    b) When the connection request from any of the mobile devices arrives, TA establishes a connection (creates the data path) with this device and the Connection Listener sends a notification through the Controller to the Upper Level with the device and application IDs.
- *Server Connection Factory*:
  - Implements transport dependent connection that was originated by the user.
  - It is developer's responsibility to implement this worker.
  - If transport does not support such ability this worker may be absent.
  - Creates a connection with the device and the application specified by the user:
    a) Both the device and the application must be already known to TA by this moment.
    b) TA may know about the device after 'search' routine or after 'restore previous state' routine.
    c) If device is not known Transport Adapter returns the error immediately.
  - When the connection is created TA sends a notification through the Controller to the Upper Level.

3) Connection:
   - The main responsibility of Client and Server connection workers is to create a **Connection**.
   - Connection is the entity that is responsible for data transmitting between the core and the device.
   - Workers and Connection they use are closely related.
   - Customizing:
     - Custom implementation of Connection must be used in custom worker(s) only.
     - It is not possible to use other types of data exchange in default workers.
     - It is possible to use default Connection implementation in custom worker.
   - Default connection implementation is based on sockets isolated in separate thread (threaded socket connection).
   - Each transport specific worker shall use transport dependent initialization of threaded socket connection.
   - If the default implementation is not convenient to developer he can create his own Connection implementing any suitable way of data exchanging (e.g. shared memory connection).
   - In this case custom workers shall be also created.

4) The Descriptor: is used for manipulating with devices and connections inside of the adapter.

5) Custom Transport Adapter creating: using the Adapter Concept provided by SDK, the developer should do the following:

   - Create a class that is derived from "`TransportAdapterImpl`" and add the implementation of necessary virtual methods. Let it be "`getDeviceType`".

```
#include "transport_adapter_impl.h"

class MyTransportAdapter : public TransportAdapterImpl
{
protected:
  virtual DeviceType getDeviceType() const;
}
```

- Create a connection class deriving from "`ThreadedSocketConnection`" and implement virtual methods.

```
#include "transport_adapter_impl.h"

class MySocketConnection : public ThreadedSocketConnection
{
  virtual ~MySocketConnection();
 protected:
  virtual bool establish(ConnectError** error);
}
```

- Create a class for the device that will be used by Controller to manage devices and implement all virtual methods.

```
#include "transport_adapter_impl.h"

class MyDevice : public Device
{
  virtual ~Device();

  virtual bool isSameAs(const Device* other_device) const;

  virtual ApplicationList getApplicationList() const;
}
```

- Create the necessary worker classes by deriving from appropriate basic worker and fill them with necessary functionality.

```
#include "transport_adapter_impl.h"

class MyDeviceScanner : public DeviceScanner
{
}

class MyServerConnectionFactory : public ServerConnectionFactory
{
}

class MyClientListener : public ClientConnectionListener
{
}
```

These workers should use connection and device created in previous steps. For instance scanner should add a list of devices to controller. This list will be used later when 'connect' request will be received. To create a data path the connection class should be used.

When connection actually starts it will update Controller with the pointer to this connection.

- When everything is created it is time to combine all together.

```
#include "transport_adapter_impl.h"

MyTransportAdapter::MyTransportAdapter()
    : TransportAdapterImpl(
          new MyDeviceScanner(),
          new MyServerConnectionFactory(),
          new MyClientListener())
{
}
```

- Create an instance of the new adapter and provide it to the Transport Manager.

```
#include "transport_manager.h"
#include "transport_manager_impl.h"
#include "my_transport_adapter_impl.h"

{
…
  TransportAdapter* my_transport_adapter =
    static_cast<TransportAdapter*>(new MyTransportAdapter);
  TransportManager *tm_impl =
    TransportManagerDefault::instance();
  tm_impl-> addTransportAdapter(my_transport_adapter);
…
}
```

This adapter will use the default connection implementation and default Adapter Listener but three worker classes will implement developer's logic. Also Transport Adapter will provide device type in developer's defined way.

`TransportAdapterImpl` virtual (but not pure virtual) methods `Store()` and `Restore()` can be reimplemented to provide resumption mechanism. Default implementations for both methods do nothing.

3. References

4. Change History

Version	Date	Status	Change description	Author/Editor
0.1	17.07.2013	draft	Initial draft	YKazakov
0.2	31.07.2013	draft	Update according to Anastasia's comments	YKazakov
0.3	05.02.2014	draft	Some notes on resumption added	NVaganov