1. Background

In Palcom Services encapsulate functionality. A Service can provide some kind of computation (typically implemented in Java), provide an interface to hardware on a physical device, or interface to another software system (such as a database). A Palcom Service acts as a point of contact to such functionality for the Palcom world, and we will in this manual describe what it takes to create a Palcom Service. There are three major parts of a Palcom Service:

- Identity of the Service (its name, and type, both user readable and as unique identities).
- Description of the Service (what commands it can send, react to, how communicates 1-1, 1-n etc)
- Activities of the Service (reacting to incoming commands, sending commands for other Services to act on, handle new connections etc.).

Services on different devices, or computers, can be connected in a Palcom network, in which case we have a distributed system, or the Services may reside on the same device, in which case the notion of Palcom Services work as a system of components. For the Service there is no difference if it is connected to local, remote, or a combination of local and remote services.

The preferred way to deploy Palcom Services is by installing them in the generic Palcom application: theThing, which executes as an application on a computer. TheThing is serving the role as a Palcom device and is a simple and flexible way to put Services to use, with its facilities to dynamically install, start and stop Services also remotely. It is also possible to create a custom Palcom device for a single, or a group of services. This might be an alternative when installing a Palcom Service on a smaller device, where the facilities of Palcom, and theThing, are not possible to support, or otherwise are not wanted. Again, for the individual Service it does not matter if it is executed on a custom device, or an instance of theThing, given it does not explore specific features of one or the other.

![Diagram](Fig 1. Examples of use of Palcom Services. This manual describes how to implement the yellow “Service” parts.)

2. Software organization

Palcom Services are implemented as extensions of the class AbstractSimpleSevice, which is in its turn part of a class hierarchy. This hierarchy has historical reasons and is subject to revision. It is therefore recommended to use AbstractSimpleService as the extension point for custom services.
3. The three parts of a Palcom Service

3.1. Service Identification

The identification part of a Service is geared both towards programmatic use (unique ids) and users (readable names). When the Service eventually is executing as part of a Palcom Device (such as theThing) the Service is addressed with a triplet:

1. the DeviceID - unique ID of the Palcom Device (theThing) on which it executes
2. the ServiceID - unique ID of the Service type and its version
3. the InstanceName - local ID referring to one execution of the Service on this Device.

The first part, DeviceID, is handled by the used device (such as theThing) and is of no concern here. The second part, ServiceID, is an elaborate unique identity of the type of the Service including information of the version of the Service. If the same service is installed on several devices, they will all have this part in common. It is thus possible to identify other, similar services on the same, or other Devices. The inclusion of version information makes it impossible to use another version of a Service by mistake, but still detect that another version of the Service is available.

The third part, the InstanceName, is a name of the Service local to the device it is executing on. It makes it possible to have more than one instance of a given Service on the same Device. The favorite example is the situation with several cameras hosted by the same device, possibly with InstanceNames such as Left, Right and Behind. The Assignment of the InstanceName can be either using an explicit mechanism embedded in the implementation of the Service, using a simple numbering scheme, or most often left to the Device the Service is installed on, such as theThing.

From the users perspective an executing Palcom Service have the readable identification:

1. deviceName
2. the ServiceName & the VersionName
3. the InstanceName

The ServiceName is a readable name of the Service-type. The ServiceName need not be unique, since it is never used programmatically, but it is recommended to be chosen fairly descriptive to avoid confusion.

The VersionName is a readable name on the version of the Service that can be chosen according to some strategy such as, numbering, major.minor, date-of-change, revision system tag etc. ServiceName&VersionName together compares with ServiceID and denote a version of a Service type.

From the users perspective an executing Service is thus named by: the DeviceName, the ServiceName, the VersionName and the InstanceName.

3.2. Service Description

In the Service description part it is specified what command the Service can send, and receive, their parameters, and the type of the parameters. This information is distributed to other devices who asks for them (such as the PalcomBrowser). Using this information it is thus possible to communicate with the Service without any prior knowledge of its capabilities. The Description is built as a data-structure which is assembled during start-up of the Service, and is thus free to be constructed dynamically. One use of this is to interface to software systems that change dynamically for example based on some internal discovery mechanism. Interface to 1-wire, telstick, dlna and Anoto-pens have been built using this facility to generate suitable Services dynamically.

The Palcom network is designed to be able to support different protocols for Service interaction (command representation), such as text-based (for easier human reading and debugging) and binary (for higher efficiency) and custom protocols (for interfacing with existing systems). For the time being the the only protocol available is the built in default Service-Interaction protocol, "P1".

Palcom Services can use one of several different schemas for distribution of commands when communicating. The most commonly used one is "Unicast" where communication is 1-1 albeit it is possible to send the same command to all Services that are connected. More information on other distribution forms will be provided later in this document.

3.3. Service Actions

This part contains the active part of the Service. Here it responds to incoming commands, sends commands in response, or spontaneously. It can also react to incoming requests from services to connect, accept or reject such requests and maintain the "status" of the Service. The action part is usually the largest part of the implementation of a Service.
4. Setting up a Service

We use the simplest possible Service, EchoService, for illustration. The setting up of a Service can be done in the constructor.

```java
import ....;

public class EchoService extends AbstractSimpleService {

    public static final DeviceID YOURDEVICE =
            new DeviceID("C:5e3845de-0803-4143-a5e7-xxxxxxxxx");

    public EchoService(AbstractDevice deploymentDevice, String instanceName) {
        // Initialize superclass
        super(deploymentDevice,
                new ServiceID(YOURDEVICE, "YOUnumber", YOURDEVICE, "YOUnumber"),
                "P1",
                "v1.0",
                "Echo Service",
                instanceName,
                "Echoes input to output",
                new UnicastDistribution(false)
        );

        // Create description
        Command cmdIn = new Command("echoIn", null, "in", "Incoming message to be processed");
        cmdIn.addParam(new Param("inputValue", "text/plain", "the text-message to the echo-system");
        getProtocolHandler().addCommand(cmdIn);

        Command cmdOut = new Command("echoOut", null, "out", "Reply message - the same message sent back to all connected services");
        cmdOut.addParam(new Param("outputValue", "text/plain", "the echo-text") );
        getProtocolHandler().addCommand(cmdOut);
    }

    Comments:

    4.1. Parameters to constructor in the superclass
    1. deploymentDevice - link to the Palcom Device we are executing on, provided as parameter to us.
    2. new ServiceID(..) - constructed unique ServiceID for our Service type. See 4.2 below
    3. "P1" - the default Service-interaction protocol format
    4. "v1.0" - VersionName, our choice of readable name of the version of this Service
    5. "Echo Service" - ServiceName, the readable name of the Service type.
    6. instanceName - name of this particular instance of the Service, here using the provided parameter
    8. new UnicastDistribution(false) - communication method, see 6.1 for details.
    9. Specification of the commands the service can handle, see 4.3 below for details.

    4.2. ServiceID
    A ServiceID is both giving the Service-type a unique ID and works as a handle for version-control when Services are updated. It consists of 2, 3 or 4 parts where each part consists of a pair:
    a. deviceId
    b. local running identifier-number
```
The idea is that such a pair forms a unique identifier. For the deviceID use the deviceID of a Palcom device on your computer, preferably your theThing. For the second part use your initials and a running number that is different each time you create a ServiceID.

The four parts of the full ServiceID are:

A. Creation-ID - this part is always present and is never changed. It is thus shared among all existing and future versions of the Service.
B. Update-ID - this part is always present and reflect the current version. For the first version of a Service A and B can be equal.
C. Previous-ID - for an updated Service the B part is assigned a new value and the previous B part now becomes the C-part.
D. Merged-ID - for an update that is merging two existing versions of the Service, both B parts are included in the new ServiceID, one as C and the other as the D part.

The following is and example of a Service ID for a new Service, A an B parts being equal:

```
ServiceID SERVICE_ID = new ServiceID(
    A: new DeviceID("C:5e3845de-0803-4143-a5e7-xxxxxxxxxxxxxx"), "MJ1",
    B: new DeviceID("C:5e3845de-0803-4143-a5e7-xxxxxxxxxxxxxx"), "MJ1")
```

When updated the new ServiceID has (at least) three parts. A new "update-ID" (B-part) is constructed and the old part in position B is moved to position C (had there been an old C part it would have been dropped).

```
ServiceID SERVICE_ID = new ServiceID(
    A: new DeviceID("C:5e3845de-0803-4143-a5e7-xxxxxxxxxxxxxx"), "MJ1",
    B: new DeviceID("C:369736b8-7e06-44e0-b5ce-yyyyyyyyyyyyy"), "BM12",
    C: new DeviceID("C:5e3845de-0803-4143-a5e7-xxxxxxxxxxxxxx"), "MJ1")
```

Here the update was made on a another device and by another person, which is reflected in the Update-ID (B) part.

4.3. Service description

For every Command that the Service can handle (send or receive) there need to be a data structure created, describing the signature of the Command.

```
Command cmdIn = new Command(
    - one for each Command the Service have.
    "echoIn", null, - its name (and a currently unused parameter - set to null)
    Command.DIRECTION_IN, - for commands the server can receive,
    - Command.DIRECTION_OUT for those it sends
    "Incoming message to be processed" - Help text
);
```

```
5. cmdIn.addParam(new Param(
    "inputValue", - Param-name,
    "text/plain", - Param-type (using mime types, see x.y),
    "the text-message to the echo-system")); - Help-text
```

6. getProtocolHandler().addCommand(cmdIn); - ... and finally register the command

Comments

This needs to be repeated for each Command the Service support, here is a second command written more compact:

```
Command cmdOut = new Command("echoOut", null, Command.DIRECTION_OUT, "Reply message - the same message sent back to all connected services");
```

```
cmdOut.addParam(new Param("outputValue", "text/plain", "the echoed-text")
}
```

```
getProtocolHandler().addCommand(cmdOut);
```

When exploring services with the PalcomBrowser it will present this information (Commands, Parameters, Types, help-texts) in the GUI. See also 6.3 for the use of grouping commands of large Services.
5. Activity of a Service

The activity of the Service is specified by implementing a number of call-back methods. They all have workable default implementations, so only those we want to interact with needs to be implemented. We start with the simple case of completing our hello-world EchoService example and address the full set of call-backs in section 8.

It should be noted that this part of a Palcom Service is executed in a separate thread. Doing time-consuming processing in the call-back methods can thus delay the handling of other call-backs to this same Service, but not delay other Services.

5.1. Accepting an input command

The essential event to react to is a call to the method "Invoked()" which is called when a connected class sends us a command. Implement this callback in order to react on incoming commands.

```java
protected void invoked(Readable conn, Command command) {
    if (command.getID().equals("echoIn")) {
        byte[] message = command.findParam("inputValue").getData();
        sendEcho(new String(message) );
    }
}
```

Comments
1. The parameter "conn" is a link back to the Service that sent us the Command. In case there are several services connected to our service and we want to send a reply exclusively to the sending Service this parameter is used.
2. Command is a representation of the incoming Command carrying its identity and parameter values.
3. Here the method "getID" enables us to check which command that has arrived (should be OK since there is only one inbound command in our case).
4. Here we fetch the value of the parameter "inputValue". Notice parameters are identified by their names, rather than positions. The value is represented as a byte-array (byte []) which enables us to handle arbitrary data. If you need to work on them you often have to convert the value to a suitable type (as indicated by the type of the parameter, "text/plain" in our case which should match a String.)
5. The semantics of our Service is to send an "echoOut" command, implemented in a separate method as of below. The parameter "message" is converted to a String since we here know its type ("text/plain"), and want to do that as soon as possible in order to take advantage of the Java type system.

5.2. Sending an output command

Sending a command involves three things, what to send, where to send it, and how to send it. In our case we know what command to send (we named it "echoOut") and we have the data to assign to its parameter. The semantics is also that an "echo" is heard by everybody listening, so we send the reply to all connected services.

```java
protected void sendEcho(String message) {
    Command reply = getProtocolHandler().findCommand("echoOut", "out");
    reply.findParam("outputValue").setData(message.getBytes());
    sendToAll(reply);
}
```

Comments:
1. This method is called as the input command "echoIn" is received.
2. We ask the ProtocolHandler to retrieve the echoOut-Command that we initially created.
3. Assign our data to its parameter "outputValue". Convert message to byte-array at this late point.
4. And send it to all connected services - including the one that sent us a message in the first place. See section 7 for alternatives to "sendToAll".

5.3. Replacing the start() call-back method.

In order to create a nicely behaving Service we should also set its status to "Fully_Operational" (or "Green"), rather than leaving it as "Not_Operational" ("Red") which is the default. The place to do this is in the call-back method start() which is called when the Service is started and thus presented to the Palcom network.
public void start() {
    setStatus(PRDService.FULLY_OPERATIONAL, "I'm ready");
    super.start();
}

5.4. Deployment

For a full listing of the example EchoService see Appendix A.

The simplest way to execute your service is to start it in the tool, theThing, where it can easily be installed and started. Written as above it will come up with a green dot signaling that it is fully operational.

Using the PalcomBrowser, you can now find your service offered as a service by your instance of theThing, easily surfed to in the Browser window. Next you can test your Service through the immediate GUI-mechanism of the PalcomBrowser, send it echoIn-command and observe that it answer properly with a echoOut-command. It is now ready to be included in Assemblies you might care to create.

*Creating services of your own will of course become more interesting!* 

6. Details, service description

6.1. Distribution

*Unicast* is used to communicate with connected services on a 1-to-1 basis. There are, however, several ways to choose which service to send to see section 7. Unicast have a parameter **Reliable** - set to true the delivery will be guaranteed to be delivered and in order (provided a network is available), while false means a best effort where commands are sent only once.

For a service using **Groupcast** all messages are communicated to all connected services on a n-to-n basis (all can send and all listen in similar to pub-sub). A Group can be distributed over Palcom-networks connected via Palcom Routers or Tunnels. To use groupcast in your service, replace UnicastDistribution with new GroupDistribution("xxx"), where the string ("xxx") is replaced with the name of the group. This string is to be known and used by all services connecting to the group.

For a service using **Broadcast** all messages are communicated to all connected services on a n-to-n basis. Broadcast is only distributed on the local Palcom network and will thus not be seen on remote networks. To use broadcast in your service, replace UnicastDistribution with new PubSubDistribution("yyy"), where the string ("yyy") is replaced with the name of the topic. This string is to be known and used by all services connecting to the topic. (On an IP-network, Palcom Broadcast can be efficiently implemented using the IP/UDP Broadcast mechanism.)

For a service using **Radiocast** every message is sent to all connected services (one sender, many can listen in). Note that in this distribution-form, messages are only sent one way and that the sending service is not aware of what services are listening in. To use radiocast in your service, replace UnicastDistribution with new RadioDistribution().

6.2. Delivery of commands

Normally commands are delivered in the order they were sent in to the receiving service. There are, however two features that can be used in the service-description, to control the delivery.

1. **Priority**
2. **Overwrite**

**Priority**

A service might have a mixture of commands with different demands on timing, for example commands that contain large quantities of data (such as an image) that takes a longer time to transfer and should not block (often short) commands that needs to be transfered as quickly as possible. Palcom provides three priority levels: High, Medium(=default), and Low. A command with a higher priority will be sent before a message with lower priority. This works also if parts of a large lower priority command has started to be sent.

Example usage:

```java
Command command1 = new Command(...);
command1.setPriorityHigh(); // only the last call will take effect
command1.setPriorityMedium();
command1.setPriorityLow();
```

- 6 -
Use `setPriorityLow` for occasional low priority commands among otherwise equally prioritized commands, and `setPriorityHigh` for occasional high priority commands. Please observe that when using the priority feature, the point is that commands are not necessarily delivered in the order they were sent, which is otherwise the case among commands with the same priority.

**Overwrite**

Some commands have the form of updates of information from a Service, such as sensor readings for example. In situations of congestion or bad connections it might be meaningless to send buffered old readings, while more recent readings have to wait. When using the overwrite feature there is only one copy of the command awaiting to be sent and when there is another sending of the same command it will replace the existing command in the output stream.

Example usage:
```java
Command command1 = new Command([...]);
command1.setOverwrite();
```

6.3. **Command parameters - Mime types**

The parameters are given a type in the mime classification style. When parameters are passed from one Command to the next, (e.g. by the PalcomBrowser when editing an Assembly) it will make sure that these types match exactly. One can thus invent custom types and be sure they are not mixed up by accident.

The PalcomBrowser interpret a small number of pre-defined mime-types:
- `text/plain` - Used for plain text, the content of such a parameter is presented as a string.
- `image/jpeg` - Used for photos, the content is displayed as an image.
- `application/x-jar` - Used for a parameter representing a jar-file. For an in-command, the PalcomBrowser interpret such a parameter type by opening a file-system browser in order for the user to specify the file-path. It then reads and place the content of that file as the parameter value (thus sends the file to the Service).

6.4. **Groups of Commands**

Parameters can be grouped in arbitrary levels. In a Service offering many Commands this can be used to make the Service easier to grasp and understand. Groups can for example be used to separate less frequently used commands, Commands supposed to be used in certain situations etc.

A Group has a descriptive name and a Help-text. Group objects can be inserted where Command objects can be inserted. In the other end, a Group-object have a addCommand() method where Command, or new Group-objects can be added. Grouping of Commands have a benefit for user readability, but do not effect the semantics of the Service.

Example:
```java
Group grp1 = new Group("grp1","first group of commands");
Command cmdA = new Command("A", null, "in", "A-command");
grp1.addCommand(cmdA);
Command cmdB = new Command("B", null, "in", "B-command");
grp1.addCommand(cmdB);
getProtocolHandler().addCommand(grp1);

Group grp2 = new Group("grp2","second group of commands");
Command cmdC = new Command("C", null, "in", "C-command");
grp2.addCommand(cmdC);
Command cmdD = new Command("D", null, "in", "D-command");
grp2.addCommand(cmdD);
getProtocolHandler().addCommand(grp2);
Command exit = new Command("exit", null, "in", "Exit-command");
getProtocolHandler().addCommand(exit);
```

Which could be presented e.g. by the PalcomBrowser along the lines indicated below:
7. Communicating with other Services

7.1. Handling user-addressing

Specifying a userId is needed by this service

Optionally, a service can require that a UserID is set before accepting connections. To do this, override the method requiresUserId to return true, and implement the callback method acceptUserId to perform the actual authentication (return true if you accept the connection).

Example authenticate implementation:

```java
protected boolean requiresUserId() {
    return true;
}

protected boolean acceptUserId(DynamicConnection connecting) {
    // acceptable() is your criterium
    return acceptable(connecting.getUserId());
}
```

If you only want to accept unique userId:s you can implement acceptable to refuse a connection with an existing userId, by using the method:

```java
checkIfUser(String userId); - Check if there already exists a Connection which used the specified userId when it connected. If so return true, else false.
```

Example:

```java
protected boolean acceptUserId(DynamicConnection connecting) {
    // Only accept unique userId:s
    return !checkIfUser(connecting.getUserId());
}
```

Depricated:

```java
requiresIdentification -use-> requiresUserId
authenticate -use-> acceptUserId
```

Providing a userId when connecting to remote service

In most cases, connecting to a service is handled in an Assembly. The editor in the PalcomBrowser supports supplying a userId when needed by a service, either with a fixed value in the Assembly, or as a parameter provided at start of the Assembly. Parameters can be supplied by the Thing. But if there still is a need to "manually" connect to a service from code you write you can provide the userId (when required by the Service connecting to) by calling (assuming declaration: DynamicConnection conn):

```java
conn.setUserID(userId); // where userId is your id-string.
...
conn.open();
```

7.2. Sending Commands

There are several ways for a service to send commands, which can be used to control which (of possibly many) connected Services the command should be delivered to. All these requires that there is an open connection to the Service in question. Most of these methods return a status, one of:

Status (these constants available in this class):
- SEND_OK -- the command was successfully placed in the output-buffer (to be sent soon)
Palcom Services - Users manual

- SEND_BUFFER_FULL -- the command was not sent because the output buffer is full.
- SEND_ERROR -- the command will not be sent for some other reason.

The "All" send methods return the count of the sends returning SEND_OK actually performed.

```
int sendToAll(Command cmd); - send cmd to all connected services, returns the count of successful deliveries to output-buffers.

<status> replyToWith(Writable connection, Command request, Command reply); - used to send a reply to a single service, in response to a previously received request-command, also provide the parameter conn from invoked(conn) needed to address the receiving Service.

<status> blockingReplyToWith(Writable connection, Command request, Command reply); - similar to replyToWith but would block if output-buffer is full.

<status> sendToUserOne(String userId, Command command); - send to one connected service which has supplied "userId" as the userId. Returns SEND_ERROR if no such connection was found.

<status> blockingSendToUserOne(String userId, Command command) - similar to replyToWith but would block if output-buffer is full.

int sendToUserAll(String userId, Command command); - send to the connected services which have supplied "userId" as their userId. Returns count of services sent to.

<status> sendToDevice(deviceID deviceID, Command command); - send to a single device of which you know its deviceID. Returns SEND_ERROR if no such connection was found.

<status> blockingSendToDevice(DeviceID deviceId, Command command); - similar to sendToDevice but would block if output-buffer is full.
```

Usage:

For a command that is sent spontaneously by a service (such as GPS-coordinates, measurements etc), it might be common that these are sent with sendToAll to reach all connected services.

For commands in a request-reply pattern, one would use replyToWith to return the reply to the requesting services. In the case the message-stream is strictly in order it might be useful to use the version blockingReplyToWith delay the processing when there is a congestion. The non-blocking versions will return SEND_BUFFER_FULL in the situation when the blocking-versions will block.

For situations where large data-sets are to be sent (such as transferring a very large file in chunks) it might be practical to use the blocking-versions above since they handle flow-control. This will block the service-thread of the service. When this is not acceptable, the sending can be placed in a separate thread.

Spontaneous commands can also be sent to a particular device. sendToDevice will thus reach the named device if connected. This might be useful for functions such as managing devices.

An alternative is to send commands directed to a named "user". A "user" might be a person or something more artificial. Such a user is not bound to a particular device, but can for example migrate between a pool of mobile devices. One use could be to implement a chat function among users.

Deprecated: In most cases, it is advised to use the method replyToWith instead.

```
sendTo(Writable connection, Command command); - used to spontaneously send a Command to a single Service over a particular connection. connection might have been saved at the time of establishing the connection (see connectionOpened) or in a previous call to invoked.

<status> blockingSendTo(Writable connection, Command command); - similar to sendTo, but would block if output-buffer is full.
```

8. Setting service status

Each service has an associated status, one of:

- Red (PRDService.NOT_OPERATIONAL) - Not operational, something is wrong
- Yellow (PRDService.PARTIALLY_OPERATIONAL) - Working but not everything is in place to provide full operation.
- Green (PRDService.FULLY_OPERATIONAL) - Everything is working properly
In order to change status, the method `setStatus()` can be used. Since a service defaults to being not operational, a good thing might be to set it to `FULLY_OPERATIONAL` when it starts:

```java
public void start() {
    setStatus(PRDService.FULLY_OPERATIONAL);
    super.start();
}
```

`setStatus()` comes in two varieties: With and without a help text. A help text is an explanatory message telling the user more about the state. For instance:

```java
setStatus(PRDService.PARTIALLY_OPERATIONAL, "No SIM card inserted");
```

The Status help-text of a Service is displayed by the `Thing` and can be used as a tracing or debugging mechanism.

### 9. Additional callbacks in `AbstractSimpleService`

Callback are used to make a Service react on certain events. In many cases they are needed to implement in order to perform certain actions at the right time. They all have default implementations which are often suitable when control are not needed.

#### Start-up/Close-down

- **start()** - is called first thing - when the Service is started and made visible to the Palcom network, thus before any Palcom connections to the Service has been established. Use this call for general initialization, set the status of the Service (as of section 7) depending on the outcome.

- **stop()** - is called when the Service (or the complete device) is about to stop/close down. At this point there might be more events scheduled for us which we might want to service. Use this call to stop initiating new activities, but in many cases this call might have little work to do.

- **afterstop()** - is called last thing, the last call-back we will get. Use this call to save information for next time we are started, terminate any external dependencies etc., undo what you did in `start()`.

#### Handling Connections

- **connectionRequested(DynamicConnection conn)** - This is called when another Palcom Service wants to create a connection to us. This method should call either `conn.accept()` or `conn.reject()`. Default implementation calls `conn.accept()` to accepts all connections. Reasons to not do so could be to limit the number of simultaneous connections (maybe only one at a time can be handled). Through the parameter `conn`, one can find out more about the Service trying to connect, e.g. `getRemoteDeviceID()`.

- **connectionOpened(Connection conn)** - In the case the call to `connectionRequested` above resulted in a call to `conn.accepted()` this method is called when the connection has actually been established. Any initialization needed because of the new connection can thus be done through this call-back. It is only called if the opening was successful, and `conn` is now in the state "Open" - can actually be used for communication.

- **connectionClosed(Connection conn)** - This is called when a connection has been successfully closed. This could be triggered by the remote Service closing it, there is a loss in communication with the remote Service, or our Service has called `conn.close()` for some reason. Use this method to undo what you set up in the `connectionOpened()` method.

When a connection is "Open" there might come calls to the call-back `invoked()` as described in 5.1.
Appendix A: Hello-world example Palcom Service: EchoService

```java
package you.palcom.examples;

import java.io.IOException;
import ist.palcom.resource.descriptor.Command;
import ist.palcom.resource.descriptor.DeviceID;
import ist.palcom.resource.descriptor.PRDService;
import ist.palcom.resource.descriptor.ServiceID;
import ist.palcom.resource.descriptor.PRDService;
import se.lth.cs.palcom.common.NoSuchDeviceException;
import se.lth.cs.palcom.common.collections.Iterator;
import se.lth.cs.palcom.common.message.tree.DataLeaf;
import se.lth.cs.palcom.common.message.tree.MessageData;
import se.lth.cs.palcom.common.message.tree.MultipartNode;
import se.lth.cs.palcom.communication.connection.Connection;
import se.lth.cs.palcom.communication.connection.DynamicConnection;
import se.lth.cs.palcom.communication.connection.Readable;
import se.lth.cs.palcom.communication.connection.Writable;
import se.lth.cs.palcom.device.AbstractDevice;
import se.lth.cs.palcom.discovery.proxy.Resource;
import se.lth.cs.palcom.logging.Logger;
import se.lth.cs.palcom.service.AbstractSimpleService;
import se.lth.cs.palcom.service.distribution.UnicastDistribution;

public class EchoService extends AbstractSimpleService {

    public static final DeviceID MYDEVICE = new DeviceID("C:5e3845de-0803-4143-a5e7-xxxxxxxxx");

    public EchoService(AbstractDevice deploymentDevice, String instanceName) {
        // Initialize superclass
        super(deploymentDevice,
            new ServiceID(MYDEVICE, "ME1", MYDEVICE, "ME1"),
            "P1","v1.0","Echo Service", instanceName, "Echoes input to output",
            new UnicastDistribution(false)
        );

        // Create description
        Command cmdIn = new Command("echoIn", null, Command.DIRECTION_IN, "Incoming message to be processed");
        cmdIn.addParam(new Param("inputValue", "text/plain", "input string"));
        getProtocolHandler().addCommand(cmdIn);

        Command cmdOut = new Command("echoOut", null, Command.DIRECTION_OUT, "Reply message - the same message sent back to all connected services");
        cmdOut.addParam(new Param("outputValue", "text/plain", "echoed-string"));
        getProtocolHandler().addCommand(cmdOut);
    }

    // Set status after start-up
    public void start() {
        setStatus(PRDService.FULLY_OPERATIONAL, "I'm ready");
        super.start();
    }
}
```
protected void invoked(Readable conn, Command command) {
    if (command.getID().equals("echoIn")) {
        byte[] message = command.findParam("inputValue").getData();
        sendEcho(new String(message));
    }
}

protected void sendEcho(String message) {
    Command reply =
        getProtocolHandler().findCommand("echoOut", "out");
    reply.findParam("outputValue").setData(message.getBytes());
    sendToAll(reply); // All listening will hear the echo coming back.
}
Open ends

10. AbstractService

(Give an introduction to managing your own service events with a custom event loop, streaming services etc

11. Interesting API:s

11.1. Connection

Some of the callbacks mentioned above have a parameter conn which provides a link to relevant information in the communication layer. These are for example:

connectionRequested(conn), connectionOpened(conn), ConnectionClosed(conn), invoked(conn, cmd)

The information one can retrieve is different depending on the kind of Distribution chosen for the Service.

Unicast

The parameter conn is an object of class DynamicConnection which implements several interfaces DirectedConnection=( RWConnection=(Readable, Writeable Connection), RemoteAware, SelectorAware) and ReliableListener.

Among the interesting methods available is:

public DeviceID getRemoteDeviceID() -- returns the deviceID of the remote device we are talking to.

Callback for finer interaction with reliable communication:

void feedbackReliableFail(Connection conn, MessageNode node) {} -- the command was NOT delivered

void feedbackReliableAck(Connection conn, MessageNode node) {} -- the command was acked by the receiving service - which might be part of an Assembly - possible further communication might still fail.

Blocking flow-control:

void waitForBufferAvailable() throws InterruptedException -- call this to block until buffer available

Callback for flow-control:

protected void feedbackBufferAvailable(Connection conn) {} -- called when re-start sending possible.

---

The following callbacks are meaningful when a Service itself takes initiative to establish a connection to a remote Service. This is the case in tools managing Assemblies or providing GUIs, but not in common Services, and are listed here only for completeness.

connectionAccepted(Connection conn)
connectionRejected(Connection conn)
connectionReopen(Connection conn)

The situations where these call-backs are interesting will be described in the Device implementers manual.

What about these - valid for a common Service??

resourceChanged(Resource r)
resourceAvailable(Resource r)
resourceUnavailable(Resource r)