

Proseminar Mobile Java (J2ME)
Bluetooth API (JSR-82)

von Christoph Schall

I Bluetooth allgemein	3
1. Hintergrund	3
1.1 Motivation und Idee	3
1.2 Geschichte.....	3
1.3 Funktionsweise	3
2. Vorteile / Probleme.....	4
3. Ausblick	4
II Java Bluetooth API (JSR-82).....	4
1. Allgemeines / Ziele.....	4
2. Funktionsweise	4
2.1 Architektur	4
2.2 Protokolle / Profile	5
2.2.1 Generic Access Profile (GAP).....	6
2.2.2 Serial Port Profile (SPP).....	6
2.2.3 Logical Link and Adaption Protocol (L2CAP)	7
2.2.4 Object Exchange Protocol (OBEX).....	7
3. Möglichkeiten.....	9

I Bluetooth allgemein

1. Hintergrund

1.1 Motivation und Idee

Der Fortschritt in der elektronischen Branche ist kaum aufzuhalten. Immer mehr Geräte finden den Weg auf den heimischen Schreibtisch. Dabei benötigt man immer mehr Kabel und muss diese sauber verlegen um dem Chaos zu entgehen. Da ist der Gedanke natürlich nicht weit, eine kabellose Alternative zu schaffen, die die Übertragung zwischen den Geräten standardisiert. Diese Alternative sollte wenig Strom verbrauchen, keinen Sichtkontakt benötigen (wie es die Datenübertragung mit IrDA erfordert) und mit wenig Kosten in der Herstellung auskommen. Mit Bluetooth ist dieser Spagat gelungen. Viele große Unternehmen, darunter Handy- und PC-Hersteller, haben sich zur *Bluetooth SIG* (*Bluetooth Special Interest Group*) zusammengeschlossen. Jedes Gerät kann mit Hilfe dieses einheitlichen Standards mit einem anderen Gerät, womöglich sogar eines anderen Herstellers, kommunizieren. So können von PC zu PDA Dateien ausgetauscht, kabellos eine Freisprecheinrichtung am Mobiltelefon betrieben oder mittels eines Bluetooth-Druckers Fotos von der Kamera ausgedruckt werden. Da sich so viele Unternehmen an den offenen Bluetooth-Standard halten, gibt es immer weniger drahtlose Geräte, die nicht auf Bluetooth basieren.

1.2 Geschichte

Bluetooth wurde bereits 1994 von Ericsson entwickelt, als man eine Möglichkeit suchte, Komponenten von Mobiltelefonen miteinander kabellos zu verbinden. Im Frühjahr 1998 schloss man sich mit anderen Industriepartnern zur Bluetooth SIG zusammen, deren Ziel es war, eine billige Lösung für drahtlose Datenübertragung zu schaffen. Die SIG ist auch Eigentümer des Markennamens. Im Juli 1999 wurde dann die Spezifikation der Version 1.0 fertig gestellt. Bluetooth hält seitdem Einzug in immer mehr Geräte. Weitere 2 Jahre dauerte es, bis schließlich 2001 die ersten Produkte auf den Massenmarkt kamen, heute ist nahezu jedes Handy mit einem Bluetooth-Chip ausgestattet. Der Name Bluetooth stammt vom skandinavischen König Harald Blaatand (zu Deutsch: Blauzahn), der im Mittelalter große Bereiche Skandinaviens unter seiner Herrschaft vereinte. Ebenso soll Bluetooth die zahllosen Bereiche der Informationstechnik vereinen.

1.3 Funktionsweise

Bluetooth nutzt den lizenzfreien Frequenzbereich zwischen 2,402 und 2,480 Gigahertz, was den Vorteil hat, dass man sein Bluetooth-Gerät nicht bei der Post anmelden muss. Der Nachteil liegt allerdings darin, dass sehr viele Geräte, wie z.B. WLAN-Adapter oder Mikrowellen denselben Frequenzbereich nutzen und die Bluetooth-Geräte so gut gegen Fremdstrahlung gesichert sein müssen. Dieses Problem hat man gelöst, indem man ein sog. *Frequency-Hopping*-Verfahren anwendet. Bei diesem Verfahren wechseln die Geräte alle 625 μ s durch 79 verschiedene Kanäle im Abstand von 1 MHz. Dadurch ist ein Bluetooth-Chip nicht von einer bestimmten Frequenz abhängig und so störungsunempfindlicher.

Zur Verbindung von Bluetooth-Geräten formen die Geräte eigene Kleinst-Funkzellen (sog. *Pikonetze*). Innerhalb eines solchen Pikonetzes ist Platz für acht Bluetooth-Geräte, ein Gerät kann jedoch mehreren Pikozellen angehören. Innerhalb eines Pikonetzes arbeitet ein Gerät als Leitstation (*Master*), die anderen Geräte sind Folgestationen (*Slaves*). Jeder Bluetooth-Chip hat die Fähigkeit sowohl als Master als auch als Slave im Pikonetz zu agieren. Des Weiteren gibt es noch *geparkte Geräte*, die gerade keine Verbindung innerhalb des Pikonetzes haben. Diese Geräte sind jedoch dem Master bekannt und können innerhalb weniger Millisekunden reaktiviert werden.

Um eine Verbindung herzustellen, werden folgende Schritte durchgeführt:

1. Will ein Gerät eine Verbindung herstellen und ist noch kein Pikonetz vorhanden, so wird ein Pikonetz gebildet und das Gerät wird zum Master.
2. Das Gerät, das an dieser Verbindung teilnehmen will, muss sich dem Master anpassen. Das heißt, es muss dieselbe Sprungfolge der Kanäle benutzen. Diese wird vom Master in der weltweit eindeutigen 48-Bit-Geräteerkennung (ähnlich der MAC-Adresse) festgelegt. Nun müssen sich die beiden Geräte synchronisieren. Dazu hat jeder Chip eine interne Uhr, die der Uhr an der Leitstation angepasst wird.

3. Jedem aktiven Gerät innerhalb des Piconetzes wird eine 3-Bit-Adresse (*Active Member Address (AMA)*) zugewiesen. Alle geparkten Geräte bekommen eine 8-Bit-*Parked-Member-Adresse (PMA)*. Will ein geparktes Gerät Daten senden, so bekommen sie eine AMA und werden damit zu Slaves. Sind alle AMAs belegt, so muss zuerst ein aktives Gerät in den parkenden Zustand geschickt werden, damit wieder ein freier Slave-Platz entsteht.

2. Vorteile / Probleme

Mit Bluetooth-Geräten ist es möglich, Kabelverbindungen (im mobilen und Desktop-Bereich) komplett zu ersetzen. Bluetooth wurde allerdings nicht als eine Alternative zu drahtlosen Netzwerken entwickelt, viel mehr sollte es von Anfang an dazu dienen, kleinere Kabelstrecken drahtlos zu ersetzen. Durch den ständigen Wechsel der Frequenzen ist Bluetooth sehr unempfindlich gegen äußere Störungen.

Leider wirft die neue Technik auch neue Probleme auf. So wird in Spanien zum Beispiel der Frequenzbereich zwischen 2,402 und 2,480 Gigahertz vom Militär genutzt, was also entweder eine Anpassung für die Geräte erfordert, die dort betrieben werden oder eine eingeschränkte Bandbreite zur Folge hat. Da auch z.B. WLANs in diesem Frequenzbereich senden, bedeutet ein Nutzen von solchen Geräten eine Einschränkung in der Bandbreite von Bluetooth, was zu Lasten der Übertragungsgeschwindigkeit ist.

3. Ausblick

Die Entwicklung von Bluetooth steckt im Moment noch in den Kinderschuhen. Wenn es im Moment auch nur sehr wenige Anwendungsmöglichkeiten gibt, so wird die Zukunft noch ungeahnte Möglichkeiten bieten. Es ist praktisch möglich, dass alle Geräte, die heute noch mühsam per Kabel verbunden werden müssen, in Zukunft drahtlos per Bluetooth kommunizieren. So reicht beispielsweise ein Bluetooth-Empfänger im PC aus um drahtlos die Tastatur, die Maus, den Drucker und das Modem anzuschließen. Den Ideen sind dabei keine Grenzen gesetzt.

Im Handy-Bereich könnte man die einzelnen Komponenten komplett trennen. Das heißt im Einzelnen, dass man das Display am Handgelenk, die Hörmuschel im Ohr, das Mikrofon am Hemd und das eigentliche Handy in der Jackentasche hat. Alle diese Komponenten könnten dann per Bluetooth kommunizieren.

II Java Bluetooth API (JSR-82)

1. Allgemeines / Ziele

Ziel der JSR-82-Spezifikation (*Java Specification Request 82*) ist es, eine Java API zu schaffen, mit der Kommunikation via Bluetooth möglich ist.

Die JSR-82-API arbeitet auf Geräten, die der CLDC-Klasse angehören. Sie soll also überwiegend auf Geräten eingesetzt werden, die sehr eingeschränkte Leistung und Speicher haben. Der vorwiegende Einsatz wird auf Handys, sowie PDAs sein. Die API ist deshalb so definiert, dass die o.g. Voraussetzungen erfüllt sind.

JSR-82 ist so konstruiert, dass ständig neue Profile hinzugefügt werden. Deshalb sind den Möglichkeiten für Entwickler kaum Grenzen gesetzt; so lange sich der Kern der API nicht ändert, kann sie ständig erweitert werden. Eigens hinzugefügte Protokolle, wie z.B. das HTTP oder FTP können einfach auf bestehende Profile, wie z.B. dem *Object Exchange Protocol (OBEX)* oder dem *Logical Link Control and Adaption Protocol (L2CAP)* aufgebaut werden, auf die unten noch näher eingegangen wird.

2. Funktionsweise

2.1 Architektur

Der *Bluetooth Protocol Stack* ist unterteilt in zwei Komponenten: den *Host* (die Hardware) und den *Controller* (die Software). Außerdem gibt es das *Host-Controller-Interface (HCI)*, welches die beiden o.g. Komponenten verbindet und so die Schnittstelle zwischen Hard- und Software bildet.

Es gibt vier unterschiedliche Schichten bei der Kommunikation via Bluetooth. Folgende Tabelle zeigt, zu welcher dieser Schichten die verschiedenen Protokolle gehören:

Aufgesetzte Protokolle	PPP, UDP/TCP/IP, OBEX, WAP, vCard, vCal, IrMC, WAE
Telephony Control Protocols	TCS-Binary, AT-commands
Cable Replacement Protocol	RFCOMM
Bluetooth Kernprotokolle	Radio, Baseband, LMP, L2CAP, SDP

Tabelle 1: Bluetooth-Protokolle

Auf einige der obigen Protokolle wird unten noch genauer eingegangen.

Die Protokolle der oberen Schichten benutzen zur Ausführung die Protokolle der unteren Schichten. Aus diesem Grund muss man nicht jedes neue Protokoll von Grund auf neu programmieren, sondern kann die bereits vorhandenen Protokolle wieder verwenden.

Abbildung 1 zeigt die Abhängigkeiten der Protokolle.

Zusätzlich zu den oben genannten Protokollen hat die Bluetooth SIG noch verschiedene Profile definiert. Diese Profile sind in verschiedene Kategorien der Kommunikation gegliedert. Für jedes Profil ist genau spezifiziert, welche Protokolle benutzt und wie sie eingesetzt werden. Die Unterteilung der Bluetooth-Geräte folgt auch aus den Profilen, da jedes Gerät ein Profil immer vollständig benutzen muss, eine teilweise Implementation ist also nicht erlaubt. Die vier gängigsten Profile sind das Generic Access Profile (GAP), das Serial Port Profile (SPP), das Service Discovery (Application) Profile (SDP) und das Generic Object Exchange Profile (GOEP). Diese Profile können voneinander abhängen, d.h. z.B. GOEP basiert auf dem SPP, welches wiederum GAP benötigt.

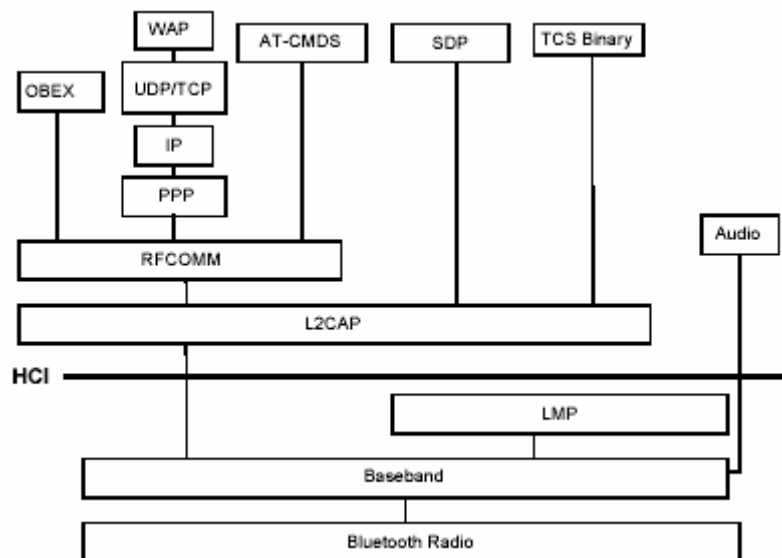


Abbildung 1: Bluetooth-Protokoll-Stack

2.2 Protokolle / Profile

Im Folgenden wird ein kurzer Überblick über die Kern-Protokolle gegeben:

- Im *Radio*-Protokoll sind die Spezifikationen der Funkschnittstelle, also die der physikalischen Verbindung zwischen den Geräten, festgelegt. Dort ist u.a. vereinbart, welche Frequenz die Bluetooth-Chips benutzen dürfen und mit welcher Sendeleistung sie senden sollen.
- Im *Baseband*-Protokoll ist festgelegt, welche Mechanismen für den Verbindungsaufbau und welche Paketformate verwendet werden sollen. Auch die Implementation des Frequency-Hoppings ist im Baseband-Protokoll verankert.
- Das *LMP (Link Manager Protocol)* stellt die Verbindungen zweier Geräte her und verwaltet sie. Außerdem ist es für die Sicherheit zuständig, das heißt es setzt den Verschlüsselungsmodus und erstellt Schlüssel, sowie Zufallszahlen. Des Weiteren synchronisiert es die Uhren der beiden Geräte, so dass die Geräte zur selben Zeit die gleiche Frequenz nutzen. Über das LMP kann die Verbindung überwacht werden. Das heißt, bei Bedarf kann das empfangende

Gerät beim Sender eine stärkere Sendeleistung anfordern bzw. die Sendeleistung drosseln lassen.

2.2.1 Generic Access Profile (GAP)

Das *Generic Access Profile (GAP)* kommt ursprünglich von DECT-Telefonanlagen, mit dessen Hilfe sich dort schnurlose Geräte herstellerübergreifend an Basisstationen anschließen lassen. Bei Bluetooth kann GAP allerdings noch etwas mehr.

GAP ist Bestandteil jedes Bluetooth-Gerätes. Mit Hilfe dieses Profils können sich die Bluetooth-Geräte gegenseitig erkennen und herausfinden, ob und in welcher Weise sie miteinander kommunizieren dürfen. Selbst wenn zwei Geräte von unterschiedlicher Art sind und nicht miteinander kommunizieren können, so stellt GAP dies fest und teilt es dem Nutzer mit. Des Weiteren wird mit GAP die Übertragung gegenüber einem dritten Gerät gesichert. Die meisten Profile haben aus diesen Gründen GAP als Grundlage.

In J2ME wird der o.g. Sachverhalt mit Hilfe der Klassen `javax.bluetooth.LocalDevice` sowie `javax.bluetooth.RemoteDevice` realisiert. In der `LocalDevice`-Klasse sind Informationen über das eigene Gerät zu finden, während in der `RemoteDevice`-Klasse Informationen über das Gerät sind, mit dem man ggf. Verbindung aufnehmen möchte. Dabei werden Geräte-Informationen (Hersteller, Software-Version usw.), so wie die Netzwerk-Adresse des Gerätes übermittelt. Die Konstanten, die dabei verwendet werden sind in der Klasse `javax.bluetooth.DeviceClasses` zu finden. Taucht dabei ein Fehler auf, so wird eine `javax.bluetooth.BluetoothStateException` geworfen, die von `java.io.IOException` abgeleitet ist.

2.2.2 Serial Port Profile (SPP)

Mit dem *Serial Port Profile (SPP)* können serielle Kabelverbindungen (RS-232) emuliert werden. Das Profil muss gemäß des Standards Datenraten bis zu 128 kb/s unterstützen, höhere Datenraten sind optional.

Unter J2ME wird eine sog. SPP-Session eindeutig durch die Adresse des verbundenen Gerätes identifiziert. Das hat zur Schlussfolgerung, dass immer nur eine Sitzung zwischen zwei Geräten bestehen kann, jedoch kann eine Sitzung mehrere Verbindungen beinhalten. Abweichungen zu diesen Einschränkungen sind möglich, es wird jedoch nicht weiter darauf eingegangen, da diese nicht dem Bluetooth-Standard angehören.

Eine Anwendung, die ein SPP-Service anbietet, ist ein SPP-Server, die Anwendung, die diesen Dienst in Anspruch nehmen will, nennt man SPP-Client. Der Server registriert seinen Dienst in der *Service Discovery Database (SDDB)* und fügt einen sog. *Server Channel Identifier* hinzu. Ein Client kann mit Hilfe der *Service Discovery API* die verfügbaren Dienste herausfinden und sich dann an Hand der Server-Adresse und dem Server Channel Identifier mit dem Server verbinden. Sobald die Verbindung besteht, können Daten bidirektional übertragen werden.

a) Clientseitige Verbindung

Um eine Verbindung mit dem Server aufzunehmen wird die sog. Connection URL benötigt. Diese besteht aus dem Protokoll (*bt_spp*), der Server-Adresse und dem Server Channel Identifier. Der Aufbau dieser URL sieht folgendermaßen aus: `bt_spp://<adresse>:<identifizier>`. Sie muss angegeben werden, wenn man ein Verbindungsobjekt erstellen möchte, welches vom Typ `javax.microedition.io.StreamConnection` ist. Mit der Methode `Connector.open()`, die die URL als einziges Argument hat, kann man ein solches Objekt erstellen. Da diese Methode jede Art von Verbindung erstellt, muss ein Typecast auf die gewünschte Verbindungsart erfolgen. Nun kann man mit diesem Objekt arbeiten wie mit jedem anderen Java-Stream-Objekt auch, also Zeichen lesen und schreiben.

Mit der Methode `close()` des `StreamConnection`-Objekts kann man die Verbindung wieder beenden.

b) Serverseitige Verbindung

Um eine Anwendung zum Server werden zu lassen, ist etwas mehr Arbeit nötig. Ich werde das an Hand des folgenden Beispiels erläutern:

```
StreamConnectionNotifier service =
    (StreamConnectionNotifier)Connector.open(
        "bt_spp://localhost:102030405060708090A1B1C1D1E100;name=SPPEX");

StreamConnection con =
    (StreamConnection) service.acceptAndOpen();
```

Wie schon bei der clientseitigen Verbindung wird mit der `Connector.open()`-Methode ein Objekt erstellt; dieses Mal allerdings ein `StreamConnectionNotifier`-Objekt. Dieses Objekt ist noch kein `Stream`-Objekt, da natürlich auch noch keine Verbindung zu einem Client vorhanden ist. Der Server wartet jetzt lediglich auf eine ankommende Verbindung, er hat jetzt also nur den Dienst erstellt. Dieser ist jetzt für Clients in der SDB sichtbar.

In der nächsten Code-Zeile wird festgelegt, was die Anwendung machen soll, wenn eine Anforderung auf den Service vorhanden ist, in diesem Fall `acceptAndOpen()`, er soll die Verbindung also akzeptieren und den Kanal öffnen. Diese Methode liefert nun ein Objekt vom Typ `StreamConnection` zurück, mit dem auf dieselbe Weise gearbeitet werden kann, wie bei der clientseitigen Verbindung.

2.2.3 Logical Link Control and Adaption Protocol (L2CAP)

Mit dem *Logical Link Control and Adaption Protocol (L2CAP)* wird eine Möglichkeit geboten, bequem eigene Protokolle zu definieren. Es vereinfacht mit seiner Architektur das Implementieren neuer Protokolle, so dass man nicht jedes Mal direkt auf der Hardware programmieren muss, sondern schon eine Software-Schnittstelle verwenden kann. L2CAP ist diese unterste Software-Schnittstelle

Grundsätzlich unterscheidet man zwischen zwei Verbindungsarten, die mit Bluetooth möglich sind, nämlich *SCO (synchronous connection-oriented)* und *ACL (asynchronous connection-less)*. SCO-Verbindungen werden überwiegend bei Sprachübertragung eingesetzt, während ACL-Verbindungen bei Datenübertragungen zum Einsatz kommen. Mit L2CAP sind nur ACL-Verbindungen möglich, weshalb man bei Sprachübertragungen direkt auf das Baseband-Protokoll zurückgreifen muss.

L2CAP gehört zur Sicherungsschicht des OSI-Modells. Das heißt, es ist für das Aufteilen der Daten in Pakete verantwortlich. Auch das evtl. Anfordern fehlerhaft übertragener Pakete gehört zu seinem Aufgabenbereich. Insbesondere gehören folgende Begriffe hinzu:

- Die *Maximum Transmission Unit (MTU)* definiert die maximale Paketgröße, die übertragen werden kann. Das Paket kann kleiner sein, darf aber unter keinen Umständen größer als der MTU-Wert sein.
- Der *Flush Timeout* bestimmt wie lange der Empfänger brauchen darf, bis er das Paket bestätigt. Kommt in dieser Zeit keine Bestätigung beim Sender an, wird das Paket nochmals gesendet.
- Die *Quality of Service (QoS)* legt Prioritäten für den Datenverkehr fest. So kann z.B. Sprachübertragung stärker bevorzugt werden als Datenübertragung.

Die Werte Flush Timeout und Quality of Service werden bereits vom Bluetooth-Stack gesetzt. Lediglich die MTU kann man mit der JSR-82-API einstellen. Wird die Einstellung nicht vorgenommen, so wird ein *DEFAULT_MTU* von 672 Bytes benutzt.

Der Aufbau einer L2CAP-Verbindung läuft im Wesentlichen so ab, wie mit SPP. Mit der `Connector.open()`-Methode wird bei der clientseitigen Anwendung eine Verbindungsanforderung geöffnet. Dabei muss wieder die URL des Servers übergeben werden (`bt2cap://<adresse>:<port>;ReceiveMTU=<wert>;TransmitMTU=<wert>`). Allerdings muss hier auf ein `L2CAPConnection`-Typ gecastet werden. Daten können dann mit der `send()`-Methode dieses Objekts an den Server geschickt werden. Mit `close()` kann diese Verbindung wieder beendet werden.

Serverseitig wird zuerst ein `L2CAPConnectionNotifier`-Objekt erstellt, das dann mit `acceptAndOpen()` die ankommende Verbindung aufbauen soll.

2.2.4 Object Exchange Protocol (OBEX)

Ein sehr wichtiges Protokoll ist das *Object Exchange Protocol (OBEX)*. Ursprünglich von der IrDA entwickelt, wo es als IrOBEX bezeichnet wird, hat es nun auch seinen Weg in die Bluetooth-Welt gefunden. Es enthält dieselben grundlegenden Basisfunktionen wie HTTP und ist weitgehend damit ver-

wandt. Ziel dieses Protokolls ist es, einen Standard für den Austausch von Objekten, wie z.B. Dateien oder vCards festzulegen.

Jede OBEX-Session beginnt damit, eine Verbindung aufzubauen (CONNECT) und endet damit eine Verbindung zu beenden (DISCONNECT). Ist die Verbindung aktiv, so kann der Client dem Server ein Objekt schicken, indem er das PUT-Kommando verwendet oder mit dem GET-Kommando ein Objekt anfordern. Mit Hilfe von SETPATH kann der Client das aktuelle Verzeichnis auf dem Server wechseln. Sind Objekte zu groß um verschickt zu werden, werden diese automatisch in kleinere OBEX-Pakete aufgeteilt. Eine OBEX-Operation wird vom Client gestartet und endet nicht bevor die PUT bzw. GET-Operation abgeschlossen ist oder ein Fehler auftritt. Bei einem PUT-Kommando werden die Objekte vom Client in Pakete aufgeteilt. Kein Paket wird verschickt, sofern nicht das vorherige vom Server bestätigt wurde. Bei einem GET-Kommando ist es ähnlich, nur dass die Objekte hier vom Server geteilt und vom Client bestätigt und zusammengesetzt werden.

OBEX stellt außerdem noch Methoden zur Verfügung um zusätzliche Informationen über das Objekt zu verschicken. Hierbei werden Header verwendet, die Länge, Name usw. enthalten.

a) Clientseitige Verbindung

Um in J2ME eine OBEX-Verbindung aufzubauen, benutzt man die `javax.obex`-Klasse. Mit Hilfe von `Connector.open()` wird vom Client eine Verbindung erstellt und ein `javax.obex.ClientSession`-Objekt zurückgegeben. Der `Connector.open()`-Methode muss auch hier ein Connection-String übergeben werden, der das Protokoll, das Ziel, sowie etwaige sonstige Parameter enthält.

Es existiert jetzt nur das Verbindungs-Objekt, die Verbindung an sich ist noch nicht hergestellt. Um die Verbindung herzustellen, wird zuerst mit Hilfe der `createHeaderSet()`-Methode ein `javax.obex.HeaderSet`-Objekt erstellt. Mit Hilfe dieses Objekts kann der Client Details über die gewünschte Verbindung im Header festlegen. Um ein CONNECT-Request zu vervollständigen, wird das `HeaderSet`-Objekt der `connect()`-Methode übergeben. Wurde der CONNECT erfolgreich abgeschlossen, so wird vom Server ein Header zurückgeschickt. Dieser wird in das `HeaderSet`-Objekt des Servers eingebunden und kann mit Hilfe der `getResponseCode()`-Methode abgefragt werden. Diese Methode gibt den Status-Code des Servers zurück, welcher in der Klasse `javax.obex.ResponseCodes` definiert ist.

Ein DISCONNECT wird in genau derselben Weise abgeschlossen, nur dass statt der `connect`-Methode die `disconnect`-Methode ausgeführt wird.

Um ein PUT bzw. GET-Kommando auszuführen wird mit Hilfe von `createHeaderSet()` ein `HeaderSet`-Objekt erstellt. Nachdem die Header-Werte spezifiziert wurden, wird vom Client die `put()`- bzw. `get()`-Methode aus dem `ClientSession`-Objekt ausgeführt. Der Header wird dann zum Sender geschickt und bekommt dann Antwort. Die `put()`- bzw. `get()`-Methode gibt ein `javax.obex.Operation`-Objekt zurück. Mit diesem Objekt kann der Client feststellen, ob die Anfrage erfolgreich war und dann mit Hilfe von Streams das OBEX-Objekt senden bzw. empfangen. Um ein PUT- bzw. GET-Request abzubrechen wird die `abort()`-Methode aus dem `javax.obex.Operation`-Objekt benutzt. Diese Methode schließt alles Streams und beendet die Operation mit der `close()`-Methode aus dem `Operation`-Objekt.

b) Serverseitige Verbindung

Vom Server kann nicht selbstständig eine Verbindung aufgebaut werden. Er muss abwarten, bis eine Anfrage eines Clients kommt und kann diese dann entweder akzeptieren oder zurückweisen. Weitere Operationen, wie z.B. PUT- oder GET-Kommandos sind nur dem Client vorbehalten. Lediglich das Unterbrechen der Verbindung und das Verweigern mancher Kommandos kann der Server vornehmen (vgl. HTTP-Protokoll).

Um ein Verbindungs-Objekt zu erstellen, wird die `Connector.open()`-Methode ausgeführt, die im Gegensatz zum Client ein `javax.obex.SessionNotifier`-Objekt zurückgibt. Auch hier wird wieder ein Connection-String benötigt. Das erstellte Objekt wartet bis vom Client eine Verbindung angefordert wird und stellt mit `acceptAndOpen()` diese Verbindung her. Diese Methode gibt ein Objekt vom Typ `javax.obex.Connection` zurück. Wollen mehrere Clients eine Verbindung zu diesem Server, so kann die `acceptAndOpen()`-Anweisung mehrmals erteilt werden. Damit bekommt man für jeden Client ein eigenes `Connection`-Objekt.

Nun muss auf dem Server eine Klasse erstellt werden, welche die Klasse `javax.obex.ServerRequestHandler` beerbt. In dieser neuen Klasse werden die Ereignis-Behandlungen definiert, die eintreten, wenn der Server eine Aktion vollzieht. Jedoch müssen nur die Methoden überschrieben werden, die vom Server auch unterstützt werden. Wenn er z.B. keine SET-PATH-Operation zulässt, muss die `onSetPath()`-Methode auch nicht näher definiert werden. Am Ende einer jeden solchen Methode müssen die Status-Codes zurückgeschickt werden, die in der `javax.obex.ResponseCodes`-Klasse zu finden sind.

Eine Server-Applikation sollte keine `abort()`-Anweisung absetzen. Wird diese Methode innerhalb einer `onPut()`- bzw. `onGet()`-Behandlung ausgeführt, führt dies unweigerlich zu einer `java.io.IOException`.

3. Möglichkeiten

Es gibt unterschiedliche Einsatzgebiete für die Bluetooth-API. Folgend möchte ich ein paar davon nennen:

Auf Grund des Bluetooth-Standards können mit den einheitlichen Protokollen bekanntlich alle Bluetooth-Geräte untereinander kommunizieren. Leider muss sich nicht nur die Hardware verstehen, auch die Software muss die gleiche Sprache sprechen, wie der Bluetooth-Partner. Aber es wäre sehr aufwändig, dieselbe Software für mehrere Handy-Typen zu schreiben.

Hier kommt der Vorteil der J2ME-Architektur zum Tragen. Eine Anwendung wird einmal mit Hilfe der JSR-82 entwickelt und kann durch die Portabilität auf verschiedene Geräte unterschiedlicher Hersteller geladen werden, die dann untereinander problemlos kommunizieren können.

Eine Anwendung wären z.B. Handy-Spiele, mit denen Multiplayer-Spiele ausgetragen werden können. So muss der Hersteller das Spiel nicht auf einen anderen Handy-Typ portieren.

Eine andere Anwendungsmöglichkeit: Mit der JSR-82-API kann ein Online-Händler Software schreiben, die es seinem Kunden vereinfacht, bei ihm einzukaufen. Der Händler muss dabei nicht verschiedene Versionen für mehrere Handy-Typen schreiben, es reicht lediglich eine J2ME-Version zu entwickeln, die auf allen Handys läuft. So können Automaten konstruiert werden, an denen der Benutzer mit seinem Handy in der Hand davor steht und mit einem Druck auf OK die Ware bezahlt. In Japan wird diese Bezahlungsmöglichkeit derzeit bereits an Fahrkartenautomaten getestet, leider jedoch ohne Einsatz von J2ME.

Im Moment sind allerdings noch kaum Handys auf dem Markt, die das neue MIDP 2.0 und damit die JSR-82-API unterstützen. Lediglich Nokia bietet zurzeit ein Handy damit an. Ein Developer-Kit kann bereits auf der Homepage von Nokia heruntergeladen werden.

Die anderen Hersteller werden dann wahrscheinlich bald nachziehen und dann steht unserer neuen schönen drahtlosen J2ME-Welt nichts mehr im Weg!

Quellen:

Informationen:

- Java APIs for Bluetooth Wireless Technology (JSR-82) – Specification Version 1.0
- Jochen Schiller - Mobilkommunikation

Abbildungen:

- Java APIs for Bluetooth Wireless Technology (JSR-82) – Specification Version 1.0