Programming Mobile Devices

Networking

University of Innsbruck

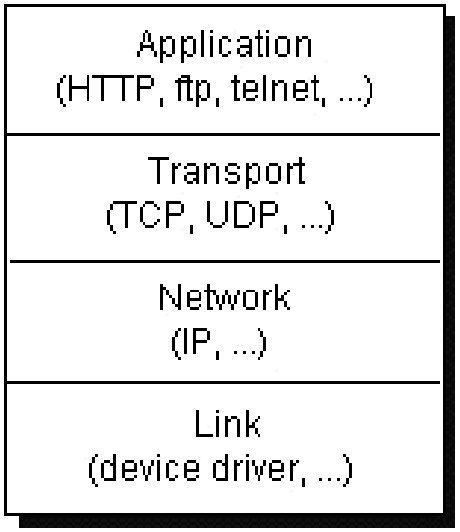
WS 2009/2010

thomas.strang@sti2.at

STI · INNSBRUCK

Networking in General

- Internet composed of millions of computers running the Internet Protocol (IP) stack
- Each machine (node) can be identified by a unique IP address, which is a 32 bit (IPv4) or 128 bit (IPv6) number, e.g. `204.68.152.68` (syntax)



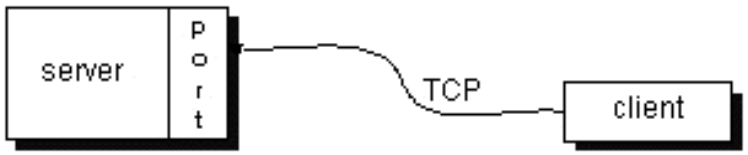
- IP address usually associated with a domain name for better readability by humans, e.g. `sti2.at`
- Ports are used to identify distinguished services on a node.

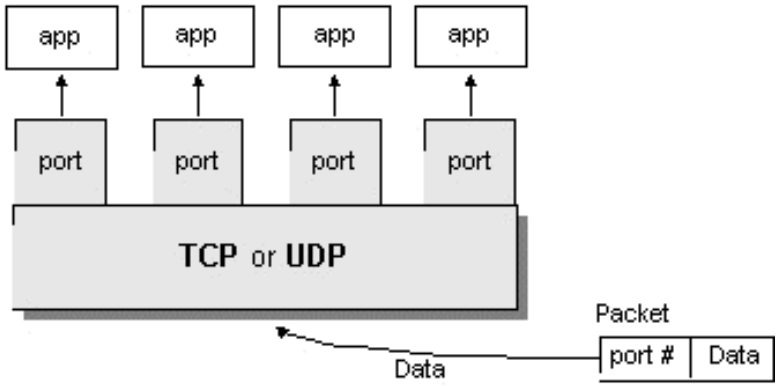
Transport Layer

- Connection-like Transport
  - TCP (Transmission Control Protocol) is a connection-based protocol that provides a reliable flow of data between two computers
  - TCP guarantees that data sent from one end of the connection gets to the other end and in the same order it was sent
- Packet-like Transport
  - UDP (User Datagram Protocol) is a protocol that sends independent packets of data, called datagrams, from one computer to another with no guarantees about arrival.
  - Can be compared to sending letters through postal service

Client-Server

- A node has (usually) only one physical connection to the network, but many applications – node is identified on the network by its IP address
- Dominant: Client-Server-Architectures
- The transport protocols (e.g. TCP, UDP) use ports to map incoming data to a particular application running on a computer

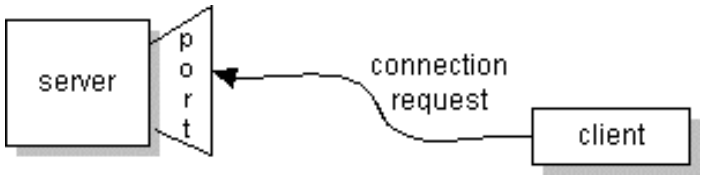


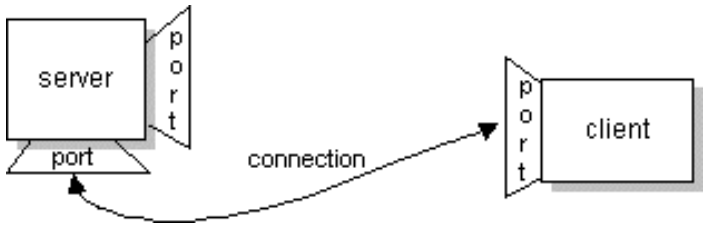


- HTTP, FTP, SMTP etc. are assigned standard ports.

Sockets

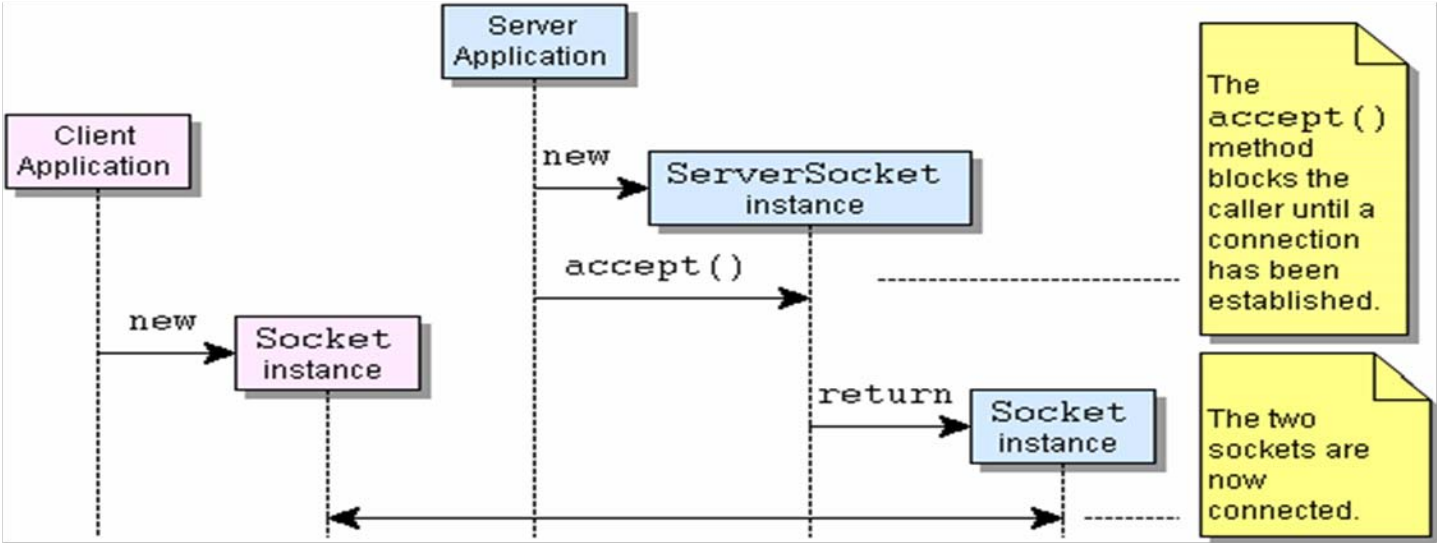
- Almost any programming language provides access to networking through the `socket` concept, also Java
  - A socket is one endpoint of a two-way communication link between two programs running on the network.
- In the nutshell:
  - Server app registers a socket that is bound to a specific port number.
  - The server just waits, listening to the socket for a client to make a connection request





Example: Client-Server over TCP

- Separation between `ServerSocket` and `Socket`
- Over time:



… now communicate…

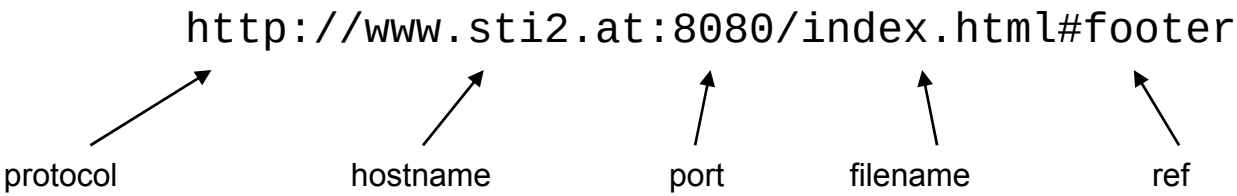
Networking in Java Standard Edition w/ Sockets

```
import java.net.*;
import java.io.*;

public class AddServer {
  ServerSocket server = new ServerSocket(10000);
  AddServer() throws IOException {
    while (true) {
      Socket client = server.accept(); //blocking
      IntputStream is = client.getInputStream();
      OutputStream os = client.getOutputStream();
      int input1 = is.read();
      int input2 = is.read();
      output.write(input1 + input2);
      output.flush();
      output.close();
      input.close();
    }
  }
  public static void main(String[] args) {
    try {
      AddServer myServer = new AddServer();
    } catch (Exception e) {}
  }
}
```

URL

- URL is an acronym for *Uniform Resource Locator* and is a reference (an address) to a resource on the Internet, e.g.



Host Name	The name of the machine on which the resource lives.
Filename	The pathname to the file on the machine.
Port Number	The port number to which to connect (typically optional).
Reference	A reference to a named anchor within a resource that usually identifies a specific location within a file (typically optional).

URL class in Java Standard Edition

- absolute

```
URL sti2 = new URL("http://www.sti2.at/");
```

- relative

```
// e.g. when parsing <a href=“team.html”>STI2 Team</a>
URL sti2 = new URL("http://www.sti2.at/about/");
URL team = new URL(sti2, "team.html");
```

- other

```
URL sti2 = new URL("http”, “www.sti2.at/about“, “team.html”);
URL sti2 = new URL("http”, “www.sti2.at“, 80, “/about/team.html”);
```

Example – Parsing an URL

```
import java.net.*;
import java.io.*;

public class ParseURL {
    public static void main(String[] args) throws Exception {
        URL aURL = new URL("http://www.sti2.org/about/"
                           + “team/index.html");
        System.out.println("protocol = " + aURL.getProtocol());
        System.out.println("host = "     + aURL.getHost());
        System.out.println("filename = " + aURL.getFile());
        System.out.println("port = "     + aURL.getPort());
        System.out.println("ref = "      + aURL.getRef());
    }
}
```

Example: Reading directly from URL

```
import java.net.*;
import java.io.*;

public class URLReader {
   public static void main(String[] args) throws Exception {
       URL sti2 = new URL("http://www.sti2.at/");
       BufferedReader in = new BufferedReader(
              new InputStreamReader(sti2.openStream()));

       String inputLine;
       while ((inputLine = in.readLine()) != null)
              System.out.println(inputLine);
       in.close();
   }
}
```

Recap: Selective Remarkables

- KVM
  - classfile preverification required
- CLDC 1.0
  - no float, no double, no serialization, no custom classloaders
  - Generic Connection Framework (GCF), but no mandatory network protocol (no sockets / IP, no RMI)
- MIDP 1.0
  - no Swing or AWT, just own GUI: Liquid Crystal Display User Interface (LCDUI)
  - mandates HTTP as network protocol
  - Application Management

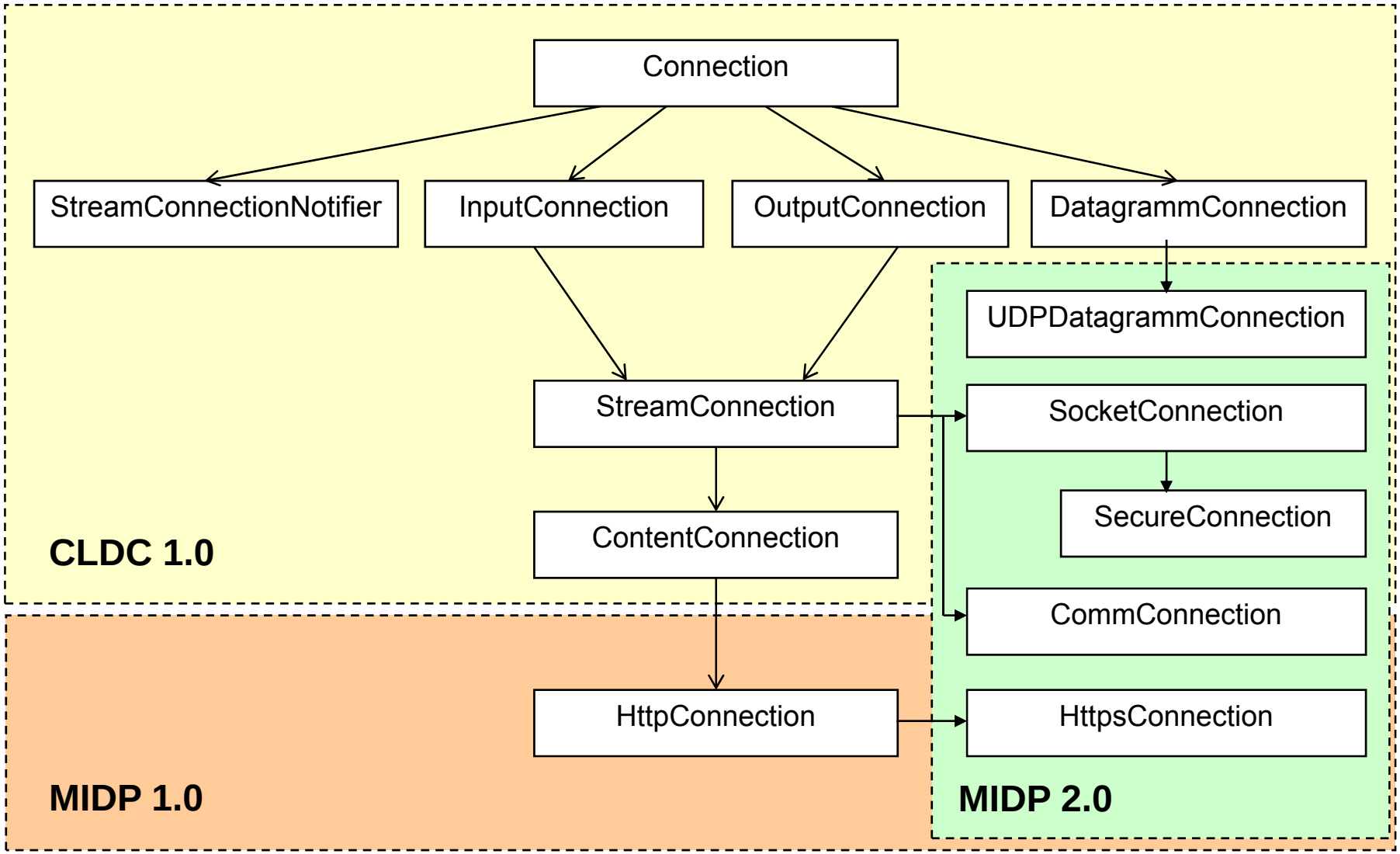
Generic Connection Framework

- Replacement for J2SE network and I/O classes with much smaller footprint
- Binding of protocols to program is done at runtime through a call to `Connector.open("<protocol>:<…>")`
  - `Connector.open("socket://129.144.11.222:8000")`
  - `Connector.open("comm:0;baudrate=9600")`
  - `Connector.open("datagram://129.144.111.123")`
  - `Connector.open("file:/readme.txt")`
- No specific protocol is mandated in CLDC 1.0
- HTTP 1.1 [RFC 2616] is mandated in MIDP 1.0

Changes by MIDP 2.0 on top of CLDC 1.1

- MIDP 2.0 (MIDP_NG, JSR-118)
  - released November 2002
  - Backward compatible with MIDP 1.0
  - Integrates *Over The Air User Initiated Provisioning Specification* as part of MIDP 2.0 spec
  - Added PKI support for security/trust of MIDP applications, similar to MExE trust model
  - Added HTTPS (Secure HTTP, Transport Level Security)
  - Push-Connections (e.g. triggered by SMS)
  - Inter-suite API & RecordStore access
  - Extended LCDUI (e.g. media player support)

GCF Interface Hierarchy



GCF usage at a glance…

```
// prepare the connection
HttpConnection c =
   (HttpConnection)Connector.open("http://www.sti2.at");

c.setRequestMethod(HttpConnection.GET);
c.setRequestProperty("If-Modified-Since",
   "01 Jan 2006 09:00:00 GMT");

// establish the connection
InputStream is = c.openInputStream(); // incl. connect

// read and process the data …

// behave friendly
is.close();
c.close();
```

Connector

- `Connector` is a factory class, used to bind a protocol implementation at run-time to the GCF
- Protocol is described just through an URL of the form

```
<scheme>:[<target>][<params>]
```

where

`<scheme>` is the protocol identifier (e.g. `http`)
`<target>` is some kind of address (e.g. `www.sti2.at`)
`<params>` are a series of name-value-pairs (e.g. `;id=0x13;lang=en`)

Connector ≠ Connection

- Caution: The static `Connector.open()` does not open (establish) a connection, but creates a protocol dependent `Connection` object (as `Connector` is a factory)
- As `Connector.open()` returns a `Connection` (the top level interface of all connections), it must be casted to a sub-interface matching the protocol in use, e.g.

```
HttpConnection c =
 (HttpConnection)Connector.open("http://www.sti2.at");
```

or

```
CommConnection c =
 (CommConnection)Connector.open("comm:com1");
```

Time of Connection Establishment

- CLDC/MIDP implements a two-phase connection establishment:
  1. create a `Connection` object with `Connector.Open()`
  2. establish the `connection` by calling either
     - connection.openInputStream
     - connection.openDataInputStream
     - connection.openOutputStream
     - connection.openDataOutputStream

- Between step 1 and step 2, the programmer can set/modify the properties of the connection object, e.g.

```
httpConnection.setRequestProperty("Accept-Language","en-us")
```

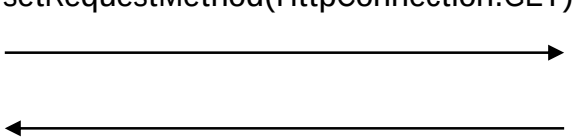
Low Level IP Networking

- Since MIDP 2.0, there is also support for *TCP/IP sockets* and *UDP/IP datagrams*
  - A `SocketConnection` is returned from `Connector.open("socket://remotehost:port")`
  - A `ServerSocketConnection` is returned from `Connector.open("socket://:port")`
  - An `UDPDatagramConnection` is returned from `Connector.open("datagram://remotehost:port")`
- If `remotehost` is omitted in URL, server behaviour is assumed: `Connector.open("datagram://:port")`

Server vs. Client Application on MID

- Mobile Device as Client


```
HttpConnection c =
  Connector.open("http://www.google.de")
c.setRequestMethod(HttpConnection.GET)
```



```
is = c.openInputStream()
// now read from is
```

- Mobile Device as Server


```
ServerSocketConnection ssc =
  Connector.open("socket://:80")
SocketConnection sc =
  ssc.acceptAndOpen() // blocking
```

```
URL url = new URL("10.11.12.23")
URLConnection c = url.openConnection()
```

```
dis = sc.openDataInputStream()
// now read from dis ... process ...
// .. and finally answer
```

Exercise 10 – text-based browser MIDlet

- Write a MIDlet, which connects to

  http://news.google.at/m/news?topic=h&hl=de

  and set the "Accept-Language" HTTP header field to
  - German
  - English

  [check RFC 2616 for encoding]

  in that or opposite order and put the content (body) to a textfield

Sessions

- Many network applications require a sequence of related Request-Response-Cycles, e.g.
  - shopping application
  - picture development service etc.
- Cycles are bound to "Sessions"
- Problem for "stateless" protocols such has HTTP. So how to maintain a state over several cycles?

Four variants to create a Session

- Cookies
  - use HTTP cookies to store values at client associated to a session [RFC 2109]
- URL rewriting
  - extend the URL with a session identifier which is used for subsequent interactions, client must be informed proprietarly
- Piggybacking Transport Layer Security
  - use HTTPS internal session management
- Form variables
  - add (usually hidden) fields to a HTML form whose value is used to identify the session
  - `<INPUT TYPE="HIDDEN" NAME="session" VALUE="...">`

MIDlet interacting with Server

- MIDlets are no browsers!
  - Cookies not supported, but easy to implement
  - URL rewriting not supported, also easy to implement
  - Piggybacking HTTPS, requires extra server functionality, supported in general since MIDP 2.0
  - Forms are HTML, not supported
- Server (e.g. Servlet) should assign session-id
- MIDlet should react on added session-id by adding it to further requests

Session handling on Server side

- Depends on server framework, e.g. Servlets:
  - Servlets provide comfortable class for handling sessions:
    - `HttpSession session = (HttpServletRequest)request.getSession(true)`
  - session objects can be used to store key-value-pairs of data associated to a session, e.g. for user-id:
    - `session.setAttribute("user-id", userName)`

How Cookies are used for Sessions

- On demand, the server initiates a (logical, not physical) session by adding a `set-cookie` HTTP header in its reply, e.g.

```
set-cookie: JSESSIONID=A47110815; Path="/tomcat"
```

- If the client desires, it may store and/or use the cookie information in subsequent requests to the same server by adding a `cookie` HTTP header, e.g.

```
cookie: JSESSIONID=A47110815
```

  - Key for session-id used by Servlets is JSESSIONID (uppercase)

Example – Cookies in MIDP (Client Side)

```
// To get cookie together with first response:

HttpConnection hc =
    (HttpConnection) Connector.open("http://www.amazon.de");
// ...
InputStream is = hc.getInputStream();
// Retrieve session ID from response
String cookie = hc.getHeaderField("set-cookie");
String sessionId;
if (cookie != null) { // sessionId is the first name=value
    sessionId = cookie.substring(0, cookie.indexOf(";"));
}

// To send cookie in subsequent requests:

hc = (HttpConnection) Connector.open("http://www.amazon.de");
if (sessionId != null) {
    hc.setRequestProperty("cookie", sessionId);
}
// ...
InputStream is = hc.getInputStream();
// ...
```

Pseudo-Example (Server Side, based on Servlets)

```
public class ShowSession extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    HttpSession session = request.getSession(true);
    response.setContentType("text/ascii");
    PrintWriter out = response.getWriter();
    String answer;
    Integer accessCount = new Integer(0);;
    if (session.isNew()) {
      answer = "Welcome, Newcomer";
    } else {
      answer = "Welcome back";
      Integer oldAccessCount =
        (Integer)session.getAttribute("accessCount");
      if (oldAccessCount != null)
        accessCount = new Integer(oldAccessCount.intValue() + 1);
      answer += " for the " + accessCount + ". time";
    }
    session.setAttribute("accessCount", accessCount);
    out.println(answer);
  }
```

Alternative: Sessions with URL Rewriting

- If cookies are not supported by or deactivated at client, a server may (also) add a session-id to the URL. As the server initiates a (logical, not physical) session, the extended URL must be communicated to the client, e.g. using the `Location` HTTP header in its reply, e.g.
  `Location: http://www.sti2.at/;jsessionid=A4711`
- If the client desires, it may use the URL extended with session information in subsequent requests to the same server by just connecting to this URL, e.g.
  `http://www.sti2.at/;jsessionid=A4711`
- Note that `";jsessionid=A4711"` is a path parameter, not a variable which would follow a `"?"` as in
  `http://www.sti2.at/;jsessionid=A4711?lang=en`

URL Rewriting on Server (based on Servlets)

```
public class ShowSession extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    HttpSession session = request.getSession(true);
    response.setContentType("text/ascii");
    String answer;
    Integer accessCount = new Integer(0);;
    if (session.isNew()) {
      answer = "Welcome, Newcomer";
    } else {
      answer = "Welcome back";
      // as some slides ago…
    }
    session.setAttribute("accessCount", accessCount);
    String urlWithSessionId = response.encodeURL();
    response.setHeader("Location", urlWithSessionId);
    // response.setStatus( 302 ); // optional, or just…
    response.getWriter().println(answer);
  }
```

Using rewritten URL on Client (MIDP)

```
// To get new URL together with first response:

String url = "http://www.amazon.de";
HttpConnection hc = (HttpConnection) Connector.open(url);
// ...
InputStream is = hc.getInputStream();
// Retrieve URL with session ID from response
String urlWithSessionId = hc.getHeaderField("location");
if (urlWithSessionId != null)
   url = urlWithSessionId;

// Connect to new URL in subsequent requests:

hc = (HttpConnection) Connector.open(url);
// ...
InputStream is = hc.getInputStream();
// ...
```

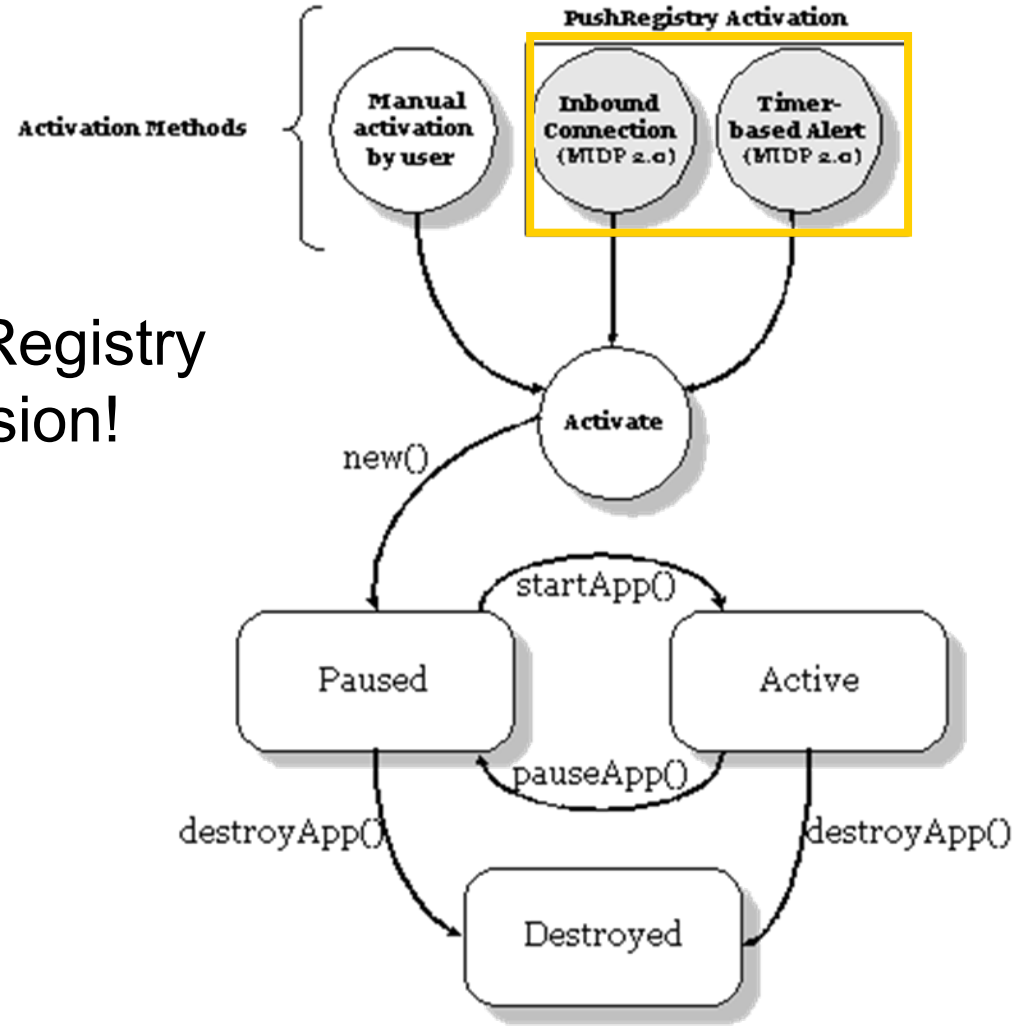
PushRegistry

- How to reach a Server-MIDlet on a mobile device which is currently not running?
- How to avoid network polling?
- How to start MIDlets automatically, i.e. at a specific time?

Answer: Using the PushRegistry (since MIDP 2.0)

Network- or Timer-initiated MIDlet Activation

Activation by PushRegistry is a life-cycle extension!



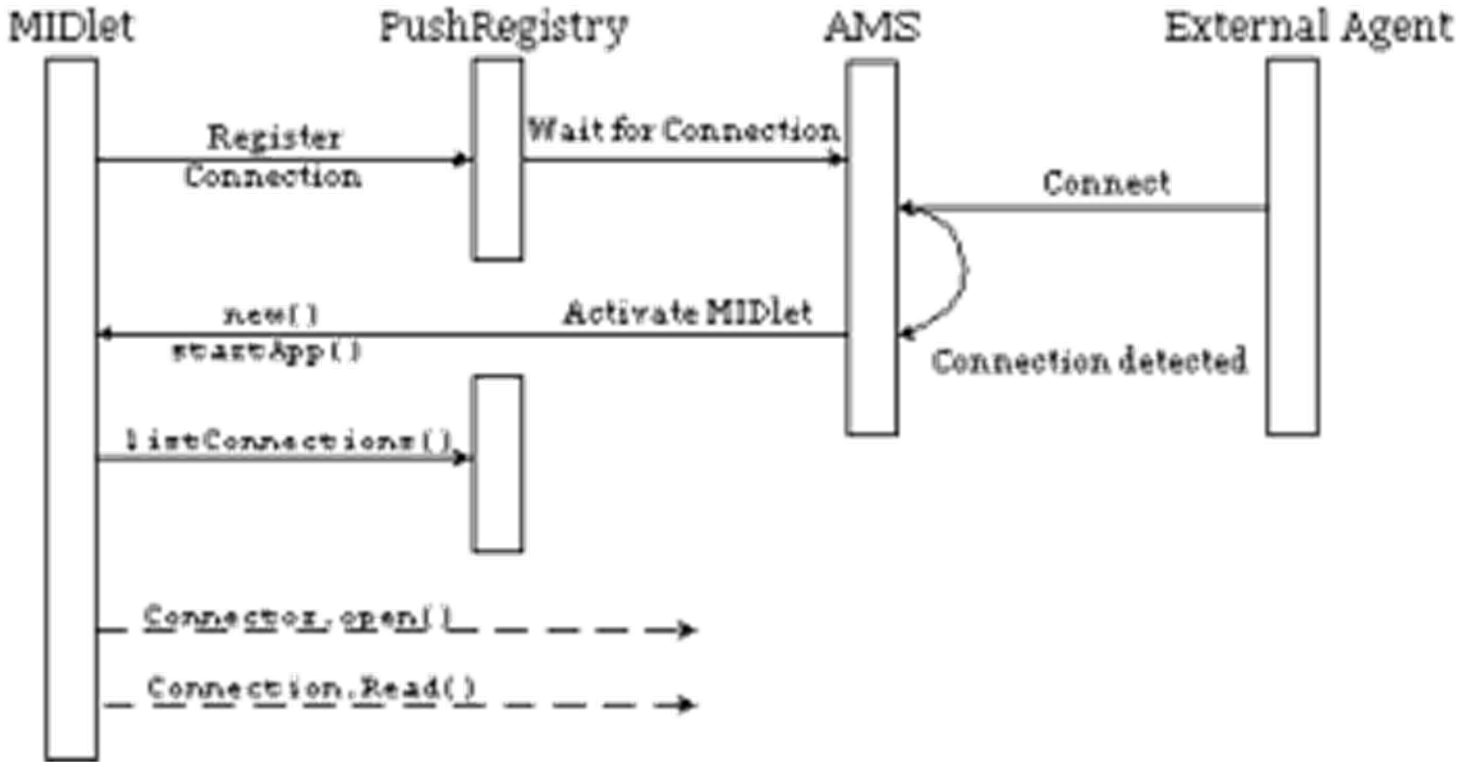
Source: Sun Developer Network

PushRegistry

- The `PushRegistry` maintains a list of possible inbound connections / conditions about when to start a MIDlet
- Inbound connections have to be registered using either
  - static info in the application descriptor
    - connection only, no timers
  - dynamically the PushRegistry API functions
    - connections and timers
- PushRegistry activates a MIDlet if it is not currently running and an inbound connection arrives for one of the registered connections; if MIDlet is already running the PushRegistry behaves transparent

Responsibility Split

- When the MIDlet is not active, the AMS monitors for incoming connections



Source: Sun Developer Network

Static Registration

- Register inbound connection within JAD:

MIDlet-Push-<n>: <ConnectionURL>, <MIDletClassName>, <AllowedSender>

where

<ConnectionURL> = the connection string used in Connector.open()
<MIDletClassName> = the class to be instantiated
<AllowedSender> = a filter that restricts the set of allowed senders ("*" is valid)

Example:

MIDlet-Push-1: socket://:8000, at.sti2.midp.TheMIDlet, *

Dynamic Registration

- Register inbound connections or timer alarms at run-time using
  - `PushRegistry.registerConnection()`
  - `PushRegistry.registerAlarm()`
- Parameters are similar to static registration
- Every MIDlet with registered inbound connections should check in startApp(), if it was started due to an inbound connection notification

```
String conn[] = PushRegistry.listConnections(true)
if (conn.length > 0)
    // proceed any input in conn[i]
    // (usually each in its own thread)
```

Exercise 11

- make sure WTK is set to MIDP 2.0
- Write a MIDlet, which listens on port 10.005 for incoming HTTP requests
- Register this MIDlet with the PushRegistry
- Make a test connection with a Web Browser to your simulated phone (http://localhost:10005) where the MIDlet is initially not running
- Let your MIDlet say 'hello' to the Browser