Mobile Systeme
Grundlagen und Anwendungen standortbezogener Dienste

Location Based Services in the Context of Web 2.0

Department of Informatics - MIN Faculty - University of Hamburg
Lecture Summer Term 2007

Dr. Thilo Horstmann

CLDC

NMEA

MIDP

Google Earth

OpenGIS

SQL

KML

Bluetooth

Mash-Ups

Web 2.0

J2ME

Loxodrome

Euler Spaces

RDMS

GPS

PostGIS

GPX

Maps

JSR 179

Polar Coordinates

API

Threads

Today: J2ME (VI)

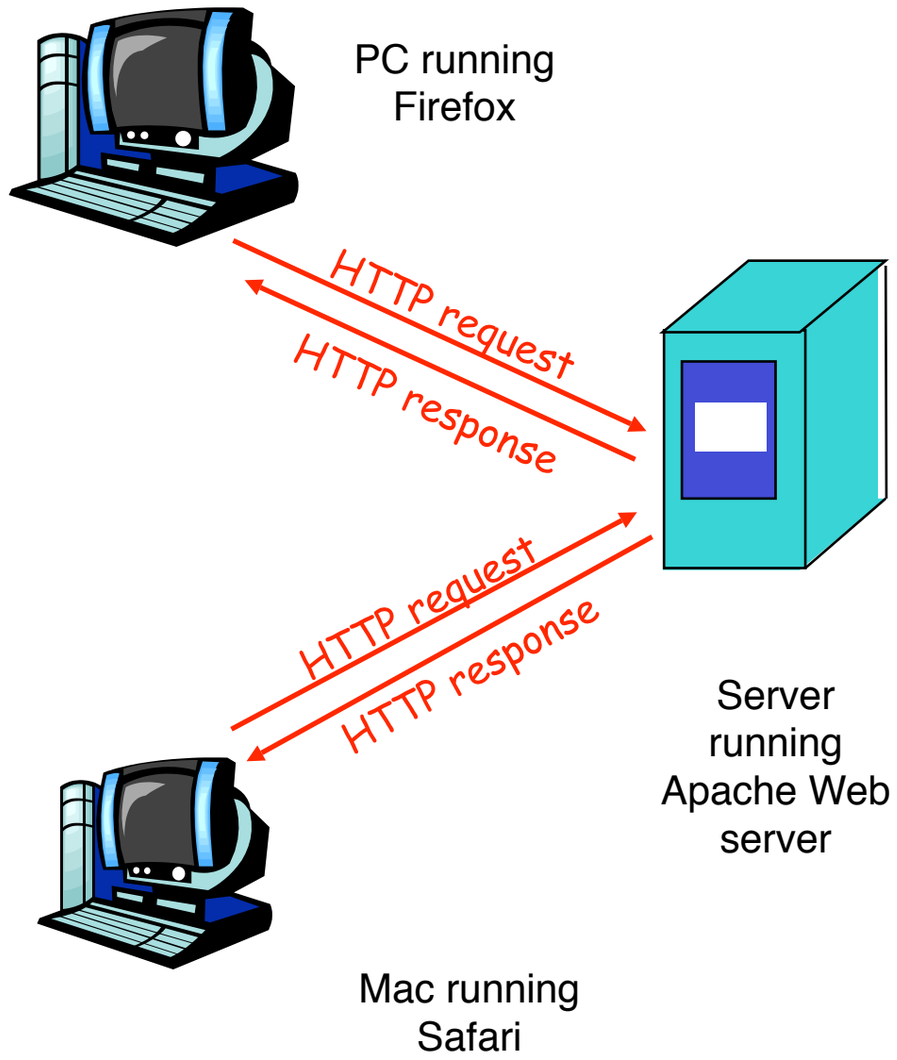
- Some HTTP basics
- J2ME and HTTP
- Walk through: Threaded MIDP Network application

The 5 layer TCP/IP model

Layer	Protocols
Application	DHCP • DNS• FTP • Gopher • HTTP • IMAP4 • IRC • NNTP • XMPP • MIME • POP3 • SIP • SMTP • SNMP • SSH • TELNET • RPC • RTP • RTCP • TLS/SSL • SDP • SOAP • …
Transport	TCP • UDP • DCCP • SCTP • RSVP • GTP • …
Internet	P (IPv4 • IPv5 • IPv6) • IGMP • ICMP • BGP • RIP • OSPF • ISIS • IPsec • ARP • RARP • …
Data Link	802.11 • ATM • DTM • Ethernet • FDDI • Frame Relay • GPRS • EVDO • HSPA • HDLC • PPP • L2TP • PPTP • …
Physical	Ethernet physical layer • ISDN • Modems • PLC • SONET/SDH • G.709 • …

The HTTP Protocol

- Web's application layer protocol
- client/server model
- client („W3 user agent"): browser that requests, receives, "displays" Web objects
- server: Web server sends objects in response to requests
- HTTP 1.0: RFC 1945
- HTTP 1.1: RFC 2068



HTTP Connection setup (1)

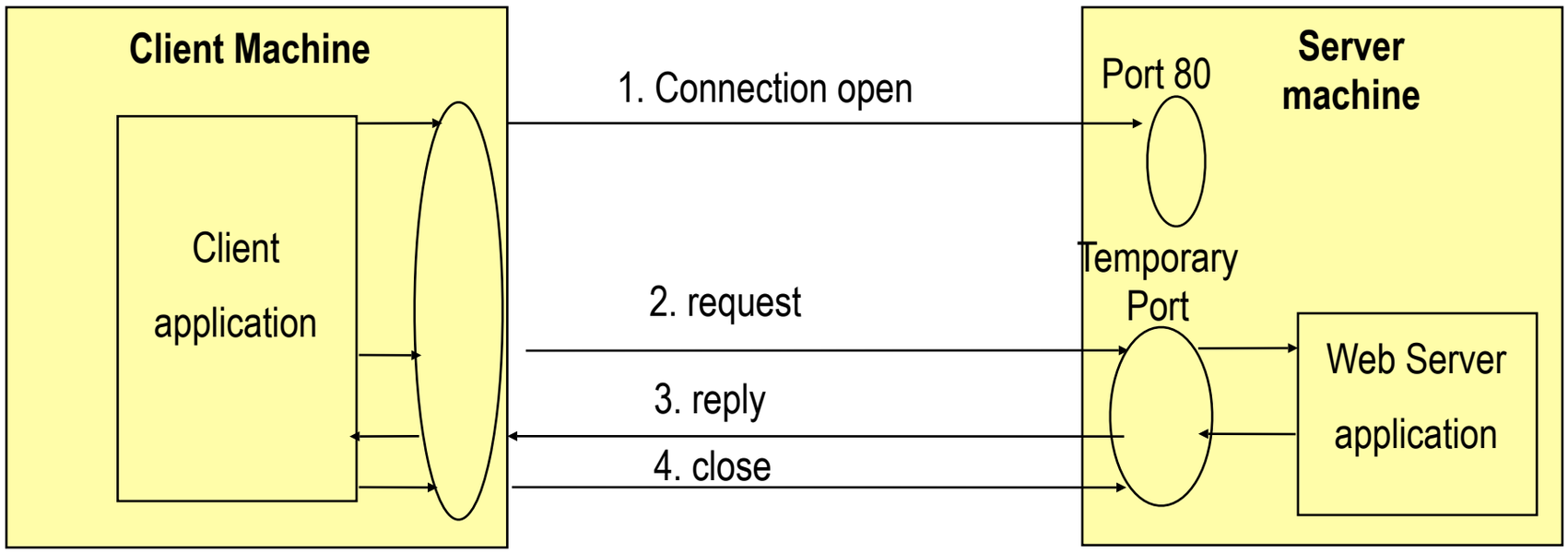
- The client web browser opens a TCP connection with the server specified in the URL the user typed.
- By default this connection initially connects to port 80. (the client can use any port. The IP packet will contain the clients port number so the web server will know which port to send to)
- TCP automatically creates a temporary port on the web server for the rest of the conversation leaving port 80 free for another client to connect to...

HTTP Connection setup (2)

- After the initial connect the client sends a URI request for a single web page or item („ressource“) on a web page to the web server
- The web server sends the client the contents of the page or item
- If the web server can not find the item an error message is returned instead
- When the conversation is over the connection is closed
- This type of communication is called request response because the client sends a single request and gets a single reply.
- However, this is inefficient because a connection has to be opened and closed for every page fetched (HTTP/1.0)

HTTP Request/Response

- Client opens connection to server on a port 80 (TCP assigns a temporary port for the duration of the conversation on the server)
- Client sends request with URL identifying resource required
- Server sends the client the resource or an error message
- The client closes the connection and the temporary port is de-allocated



Non persistent HTTP

1a. HTTP client initiates TCP connection to HTTP server (process) at www.uni-hamburg.de on port 80

1b. HTTP server at host www.uni-hamburg.de waiting for TCP connection at port 80. "accepts" connection, notifying client

2. HTTP client sends HTTP request message (containing URL) into TCP connection socket. Message indicates that client wants object /Info/neues.shtml

3. HTTP server receives request message, forms response message containing requested object, and sends message into its socket

4. HTTP server closes TCP connection.

5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

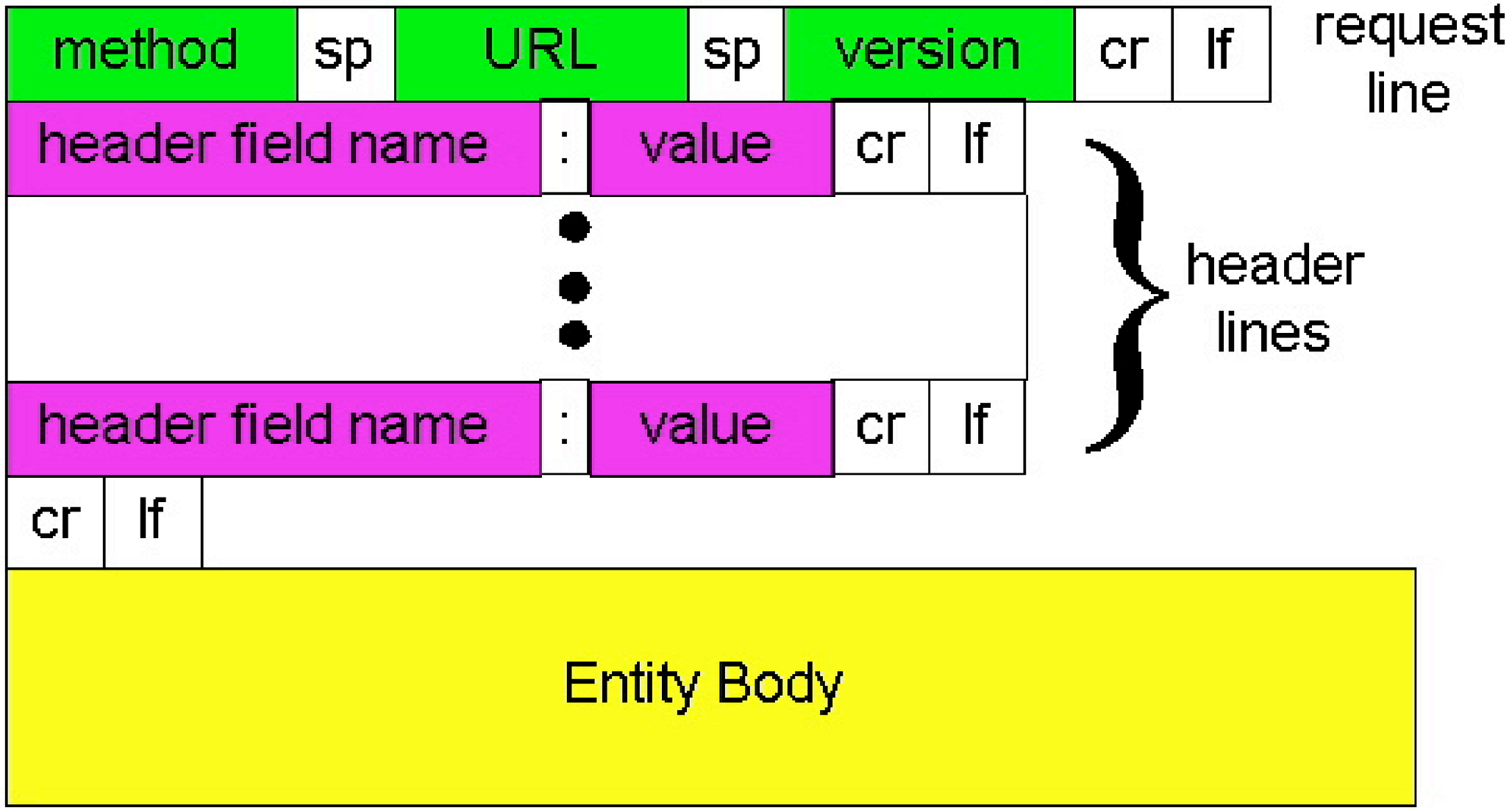
6. Steps 1-5 repeated for each of 10 jpeg objects

time

HTTP Connections

- Nonpersistent HTTP
  - At most one object is sent over a TCP connection.
  - HTTP/1.0 uses nonpersistent HTTP
- Persistent HTTP
  - Multiple objects can be sent over single TCP connection between client and server.
  - server leaves connection open after sending response
  - subsequent HTTP messages between same client/server sent over open connection (HTTP pipeling)
  - HTTP/1.1 uses persistent connections in default mode

Gerneric HTTP Request



HTTP Request

HTTP method

HTTP version

Request-URI

Host header

TCP

```
= Info: About to connect() to www.informatik.uni-hamburg.de port 80
== Info:   Trying 134.100.9.77... == Info: connected
== Info: Connected to www.informatik.uni-hamburg.de (134.100.9.77) port 80
=> Send header, 194 bytes (0xc2=)
0000: GET /Info/neues.shtml HTTP/1.1
0020: User-Agent: curl/7.13.1 (powerpc-apple-darwin8.0) libcurl/7.13.1
0060:  OpenSSL/0.9.7l zlib/1.2.3
007c: Host: www.informatik.uni-hamburg.de
00a1: Pragma: no-cache
00b3: Accept: */*
00c0:
```

Orange: Optional HTTP (request) headers

HTTP Methods

- The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI.
- The HEAD method is identical to GET except that the server MUST NOT return a message-body in the response.
- The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line.
- The PUT method requests that the enclosed entity be stored under the supplied Request-URI.
- The DELETE method requests that the origin server delete the resource identified by the Request-URI.
- The TRACE method is used to invoke a remote, application-layer loop- back of the request message.
- This specification reserves the method name CONNECT for use with a proxy that can dynamically switch to being a tunnel
- The OPTIONS method represents a request for information about the communication options available on the request/response chain identified by the Request-URI.

HTTP Request Headers

- The request-header fields allow the client to pass additional information about the request, and about the client itself, to the server.
- These fields act as request modifiers, with semantics equivalent to the parameters on a programming language method invocation.

```
request-header = Accept
                               | Accept-Charset
                               | Accept-Encoding
                               | Accept-Language
                               | Authorization
                               | Expect
                               | From
                               | Host
                               | If-Match
                               | If-Modified-Since
                               | If-None-Match
                               | If-Range
                               | If-Unmodified-Since
                               | Max-Forwards
                               | Proxy-Authorization
                               | Range
                               | Referer
                               | TE
                               | User-Agent
```

HTTP Response

HTTP version

HTTP Response Message

HTTP Response Code

```
0000: HTTP/1.1 200 OK
0000: Date: Wed, 06 Jun 2007 15:00:11 GMT
0000: Server: Apache/2.0.58 (Unix) PHP/4.4.2 mod_ssl/2.0.58 OpenSSL/0.9.7g
0000: Accept-Ranges: bytes
0000: Transfer-Encoding: chunked
0000: Content-Type: text/html; charset=ISO-8859-1
0000: 95
0004: <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
...
== Info: Connection #0 to host www.informatik.uni-hamburg.de left intact
== Info: Closing connection #0
```

TCP

Orange: HTTP (Response) headers

Btw, What violates the spec here?

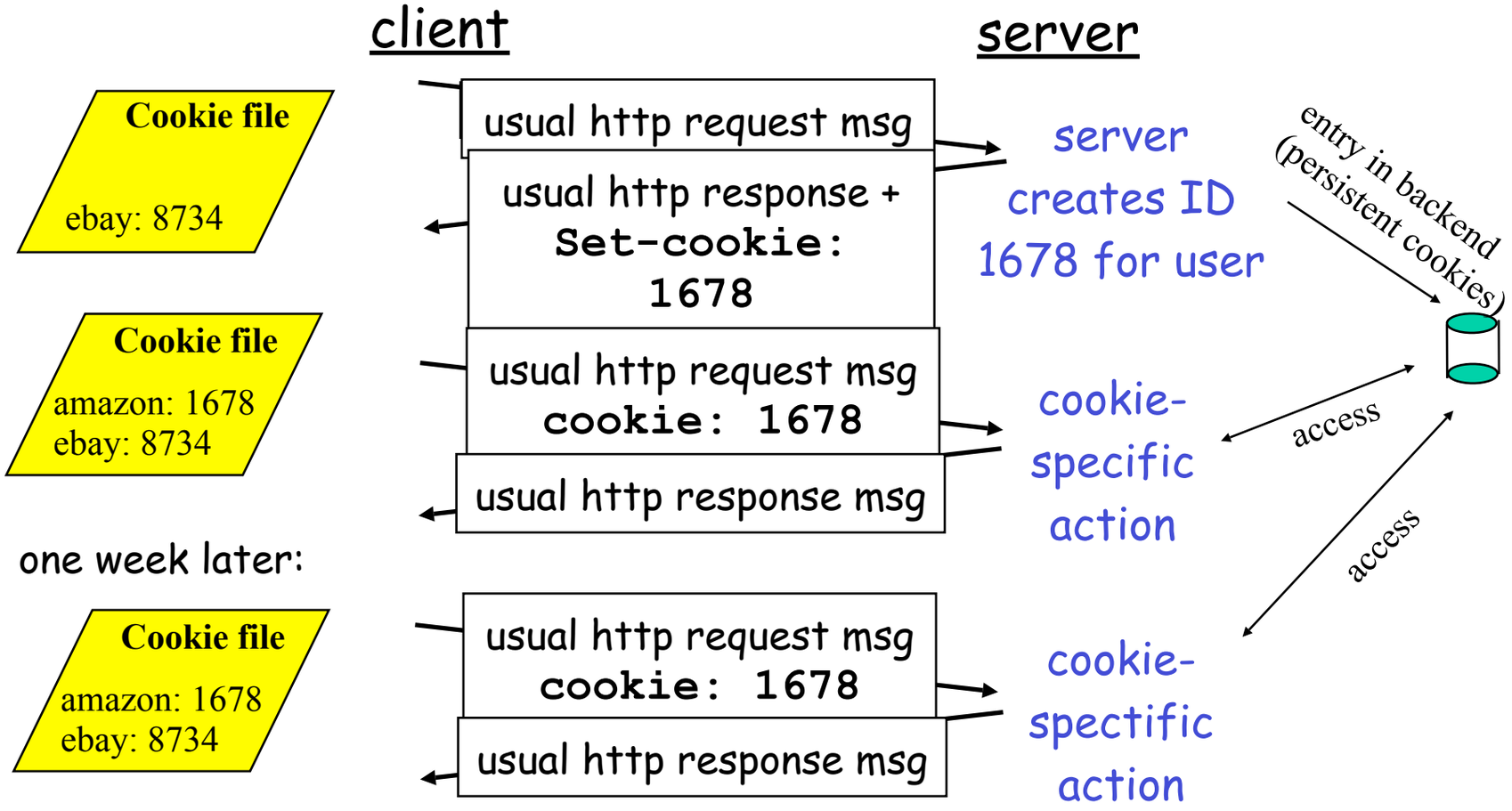
HTTP Response Codes

- 1xx: Informational
  - Request received, continuing process
- 2xx: Success
  - The action was successfully received, understood, and accepted
- 3xx: Redirection
  - Further action must be taken in order to complete the request
- 4xx: Client Error
  - The request contains bad syntax or cannot be fulfilled
- 5xx: Server Error
  - The server failed to fulfill an apparently valid request

HTTP is stateless! What?

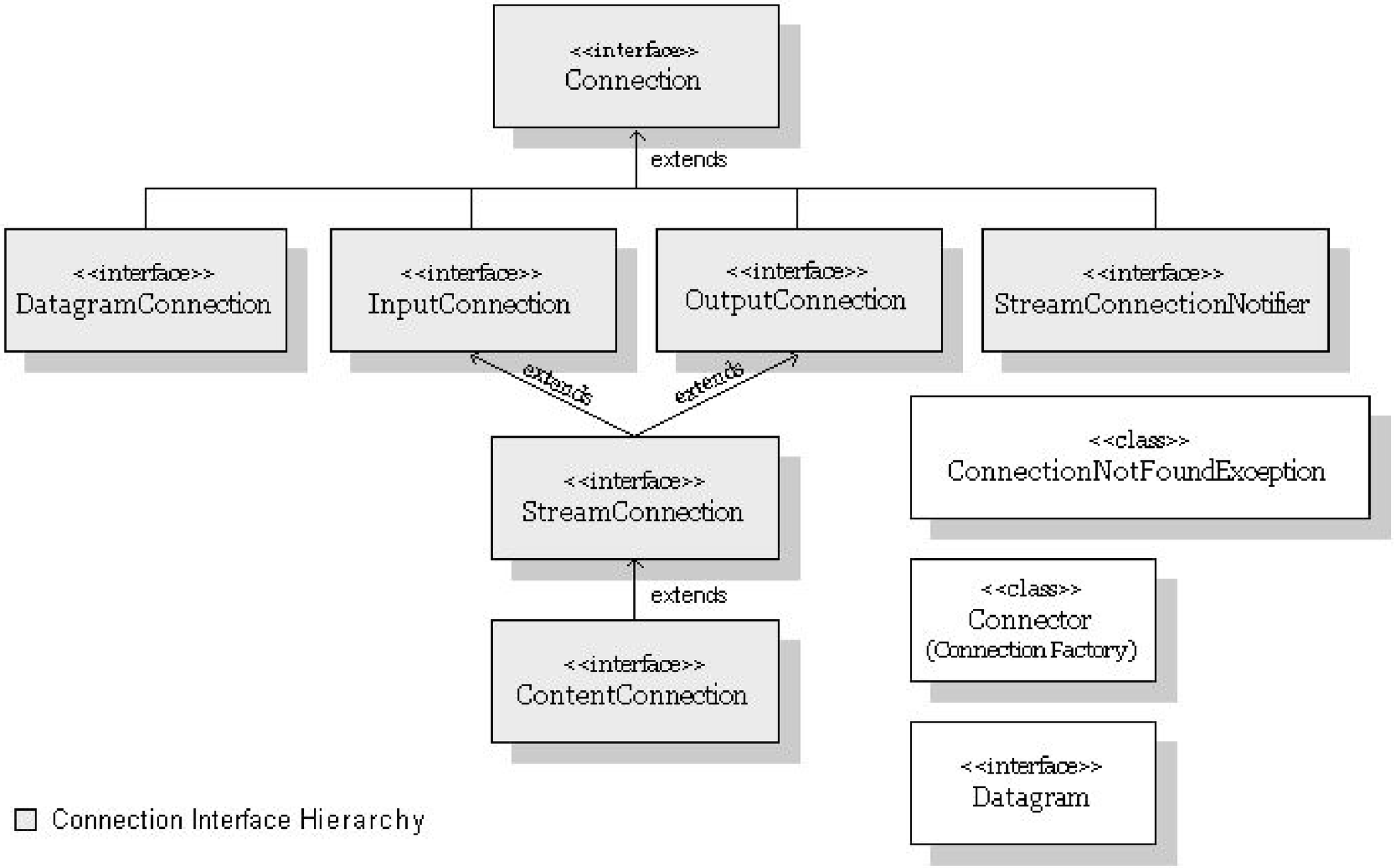
- server maintains no information about past client requests
- Protocols that maintain “state” are complex!
  - past history (state) must be maintained
  - if server/client crashes, their views of “state” may be inconsistent, must be reconciled
- But how do we store „state“, e.g. the contents of a shopping cart?
  - Cookies (*not* part of HTTP!)
  - initially introduced by Netscape (still de-facto standard)
  - RFC: http://www.faqs.org/rfcs/rfc2965.html

Maintaining server state via cookies

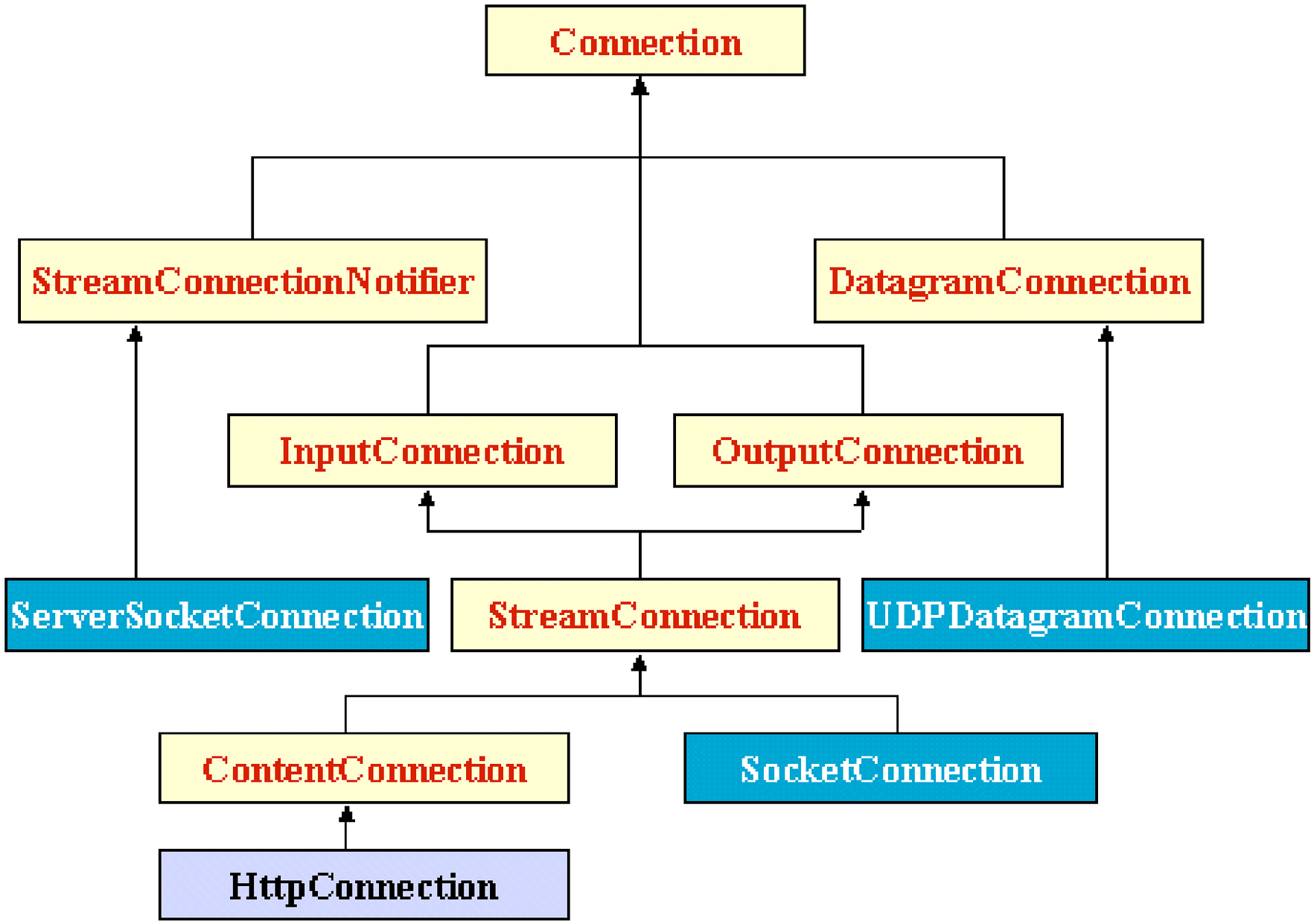


J2ME and HTTP

The CLDC GCF Interface Hierarchy



GCF Interfaces (CLDC) & Implementation (MIDP)



Interface HttpConnection (javax.microedition.io)

- defines the necessary methods and constants for an HTTP connection
- The connection exists in one of three states:
  - *Setup*, in which the request parameters can be set
  - *Connected*, in which request parameters have been sent and the response is expected
  - *Closed*, the final state, in which the HTTP connection as been terminated

How does the state change?

- The following methods may be invoked only in the Setup state:
  - `setRequestMethod, setRequestProperty`
- The following methods cause the transition to the Connected state when the connection is in Setup state.
  - `openInputStream, openDataInputStream, getLength, getType, getEncoding, getHeaderField, getResponseCode, getResponseMessage, getHeaderFieldInt, getHeaderFieldDate getExpiration, getDate, getLastModified, getHeaderField`
- The following methods may be invoked while the connection is in Setup or Connected state.
  - `close, getRequestMethod, getRequestProperty, getURL, getProtocol, getHost, getFile, getRef, getPort, getQuery`

Open a HTTP Connection using a StreamConnection

```
void getViaStreamConnection(String url) throws IOException {
        StreamConnection c = null;
        InputStream s = null;
        try {
            c = (StreamConnection)Connector.open(url);
            s = c.openInputStream();
            int ch;
            while ((ch = s.read()) != -1) {
                ...
            }
        } finally {
            if (s != null)
                s.close();
            if (c != null)
                c.close();
        }
    }
```

Open a HTTP Connection using a ContentConnection

```
void getViaContentConnection(String url) throws IOException {
        ContentConnection c = null;
        DataInputStream is = null;
        try {
            c = (ContentConnection)Connector.open(url);
            int len = (int)c.getLength();
            dis = c.openDataInputStream();
            if (len > 0) {
                byte[] data = new byte[len];
                dis.readFully(data);
            } else {
                int ch;
                while ((ch = dis.read()) != -1) {
                    ...
                }
            }
        } finally {
            if (dis != null)
                dis.close();
            if (c != null)
                c.close();
        }
    }
```

Open a HTTP Connection using a HttpConnection

```
void getViaHttpConnection(String url) throws IOException {
        HttpConnection c = null;
        InputStream is = null;
        int rc;

        try {
            c = (HttpConnection)Connector.open(url);

            // Getting the response code will open the connection,
            // send the request, and read the HTTP response headers.
            // The headers are stored until requested.
            rc = c.getResponseCode();
            if (rc != HttpConnection.HTTP_OK) {
                throw new IOException("HTTP response code: " + rc);
            }

            is = c.openInputStream();

            // Get the ContentType
            String type = c.getType();
```

Open a HTTP Connection using a HttpConnection

```
// Get the length and process the data
            int len = (int)c.getLength();
            if (len > 0) {
                int actual = 0;
                int bytesread = 0 ;
                byte[] data = new byte[len];
                while ((bytesread != len) && (actual != -1)) {
                   actual = is.read(data, bytesread, len - bytesread);
                   bytesread += actual;
                }
            } else {
                int ch;
                while ((ch = is.read()) != -1) {
                    ...
                }
            }
        } catch (ClassCastException e) {
            throw new IllegalArgumentException("Not an HTTP URL");
        } finally {
            if (is != null)
                is.close();
            if (c != null)
                c.close();
        }
    }
```

Using HTTP Post (1)

```
try {
            c = (HttpConnection)Connector.open(url);

            // Set the request method and headers
            c.setRequestMethod(HttpConnection.POST);
            c.setRequestProperty("If-Modified-Since",
                "29 Oct 1999 19:43:31 GMT");
            c.setRequestProperty("User-Agent",
                "Profile/MIDP-2.0 Configuration/CLDC-1.0");
            c.setRequestProperty("Content-Language", "en-US");

            // Getting the output stream may flush the headers
            os = c.openOutputStream();
            os.write("LIST games\n".getBytes());
            os.flush();             // Optional, getResponseCode will flush

            // Getting the response code will open the connection,
            // send the request, and read the HTTP response headers.
            // The headers are stored until requested.
            rc = c.getResponseCode();
            if (rc != HttpConnection.HTTP_OK) {
                throw new IOException("HTTP response code: " + rc);
            }
```

Using HTTP Post (2)

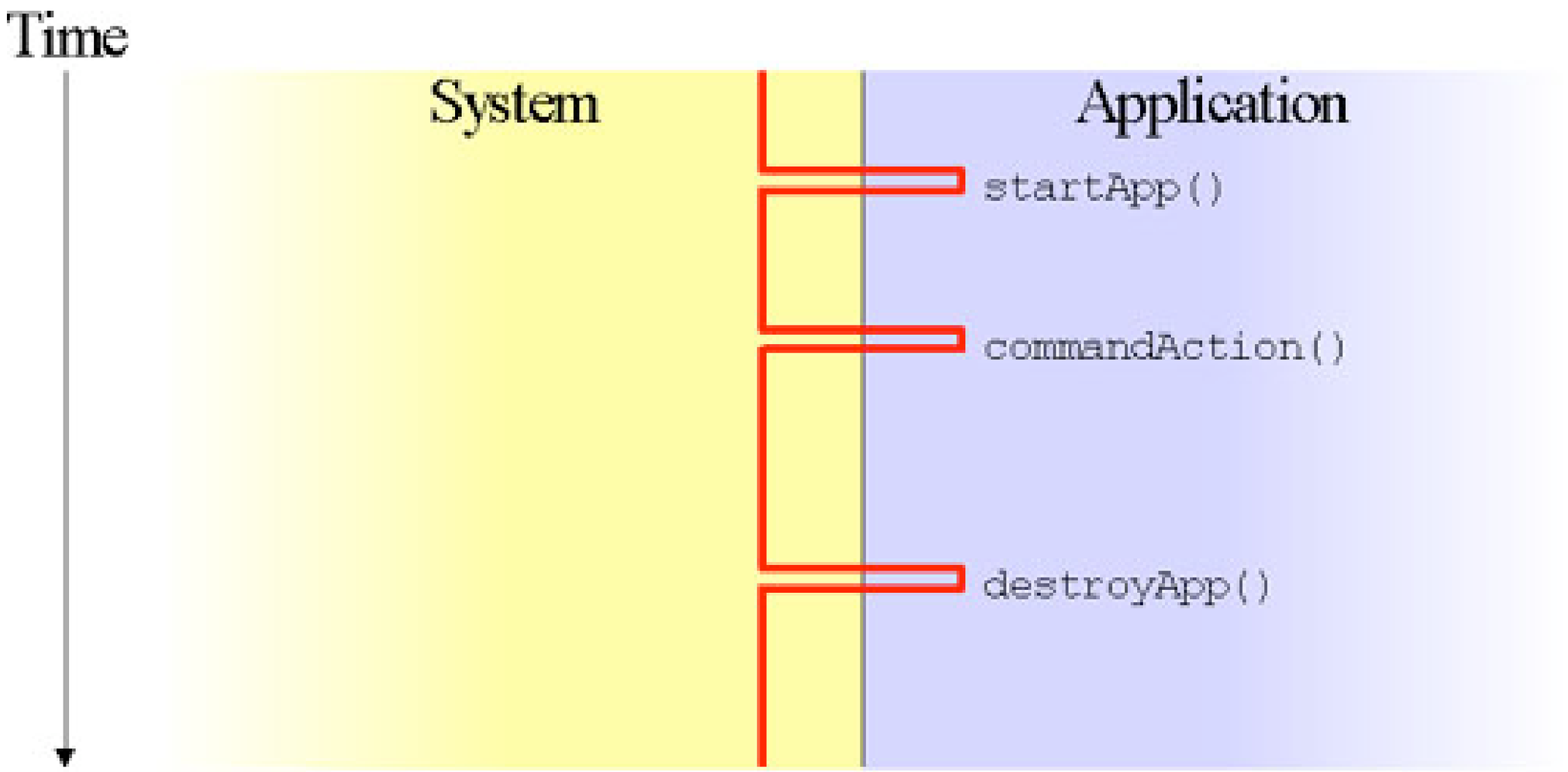
```
is = c.openInputStream();
// Get the ContentType
String type = c.getType();
processType(type);
// Get the length and process the data
int len = (int)c.getLength();
if (len > 0) {
     int actual = 0;
     int bytesread = 0 ;
     byte[] data = new byte[len];
     while ((bytesread != len) && (actual != -1)) {
        actual = is.read(data, bytesread, len - bytesread);
        bytesread += actual;
      }
        process(data);
    } else {
        int ch;
        while ((ch = is.read()) != -1) {
            process((byte)ch);
        }
    }
} catch (ClassCastException e) {
    throw new IllegalArgumentException("Not an HTTP URL");
} finally {
    if (is != null)
        is.close();
    if (os != null)
        os.close();
    if (c != null)
        c.close();
}
}
```

Walk through: From a blocking toward a multithreaded midp network application

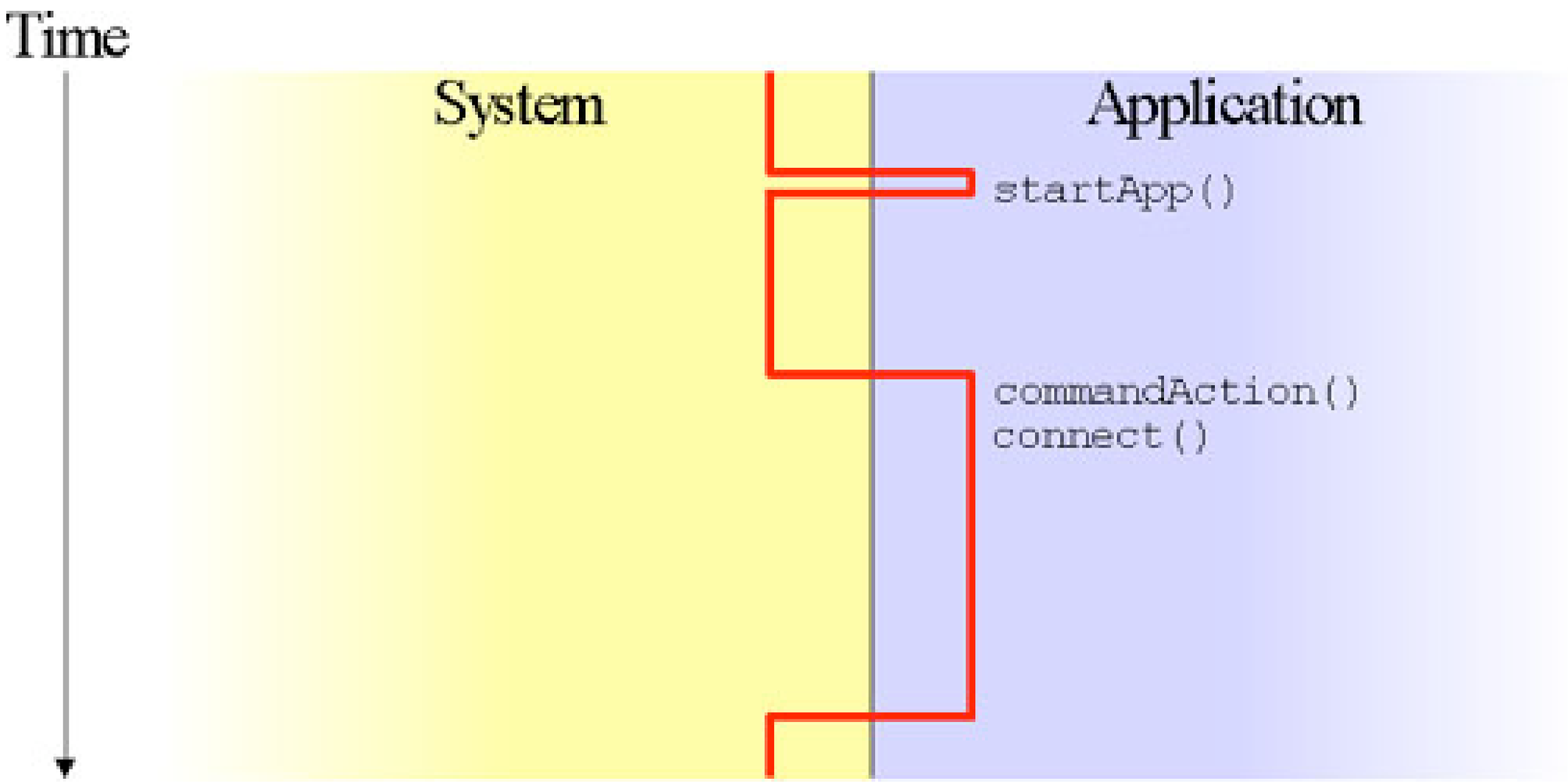
Software design issues

- Calls to HTTPConnection block!
  - Do not hijack a system thread for his own lengthy processing, e.g. in a CommandAction
  - The user interface may become unresponsive
  - Network operations usually take much longer on a mobile device
    - lower CPU, less bandwidth

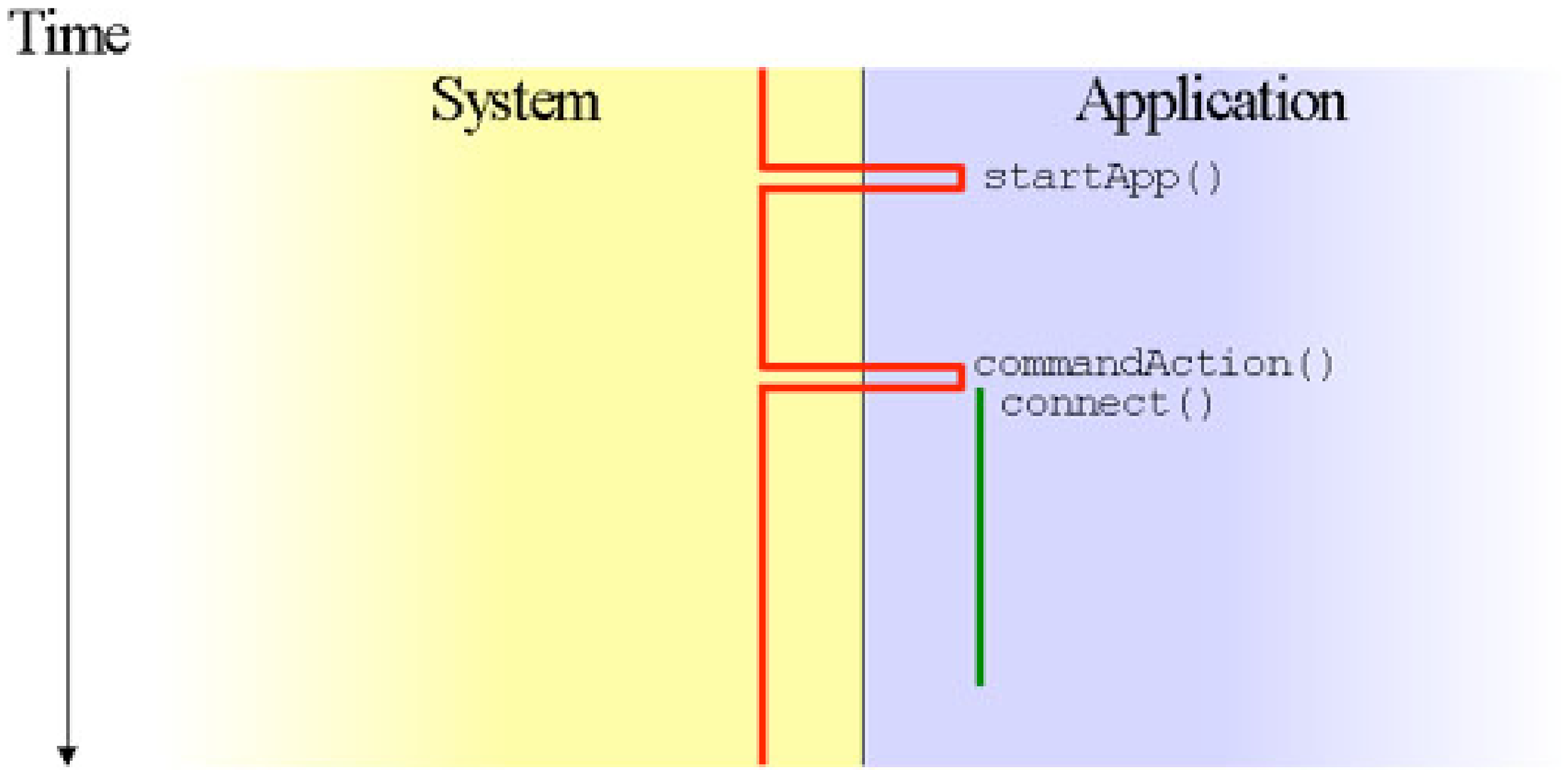
Normal threading



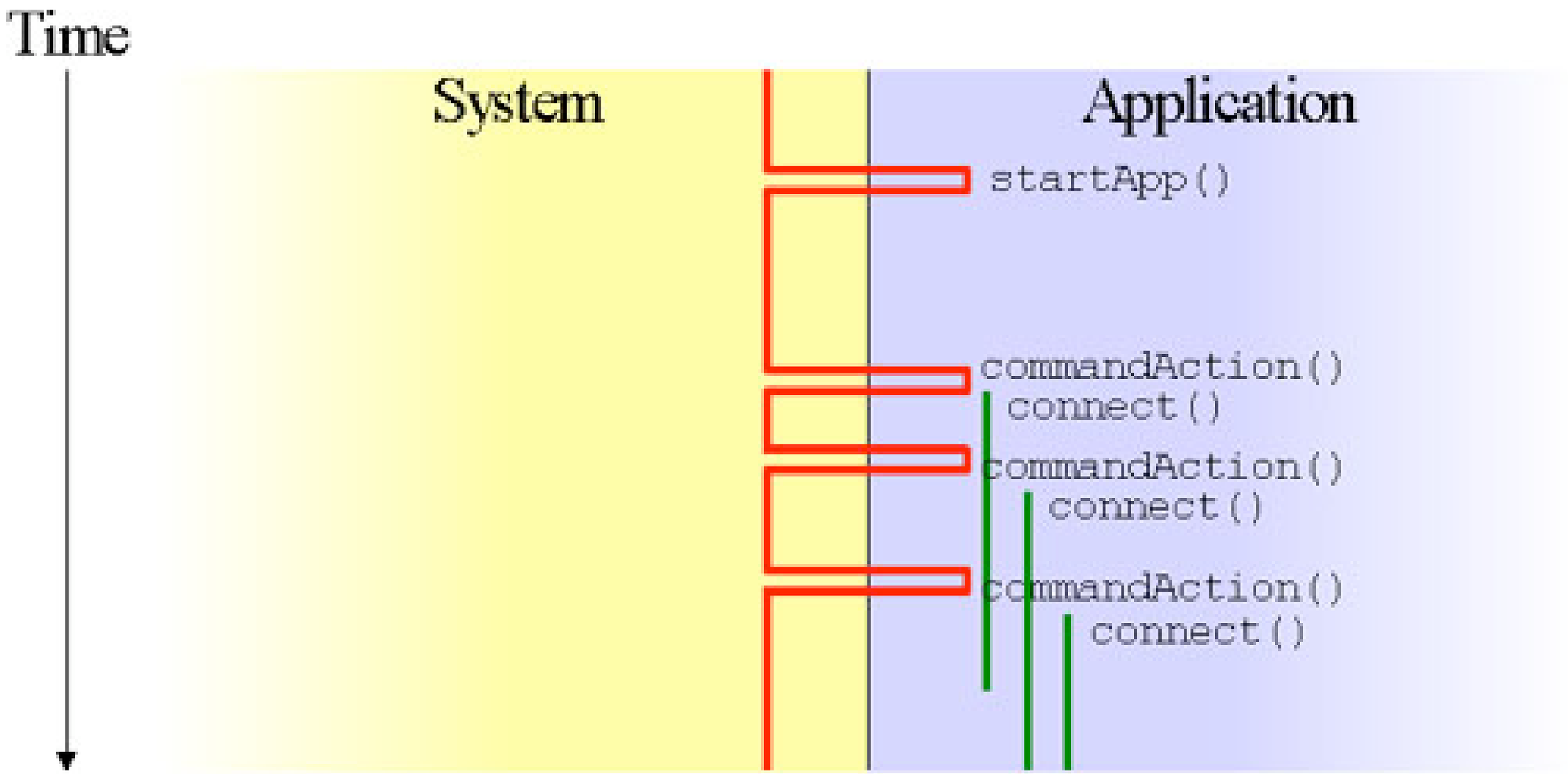
Primitive Threading



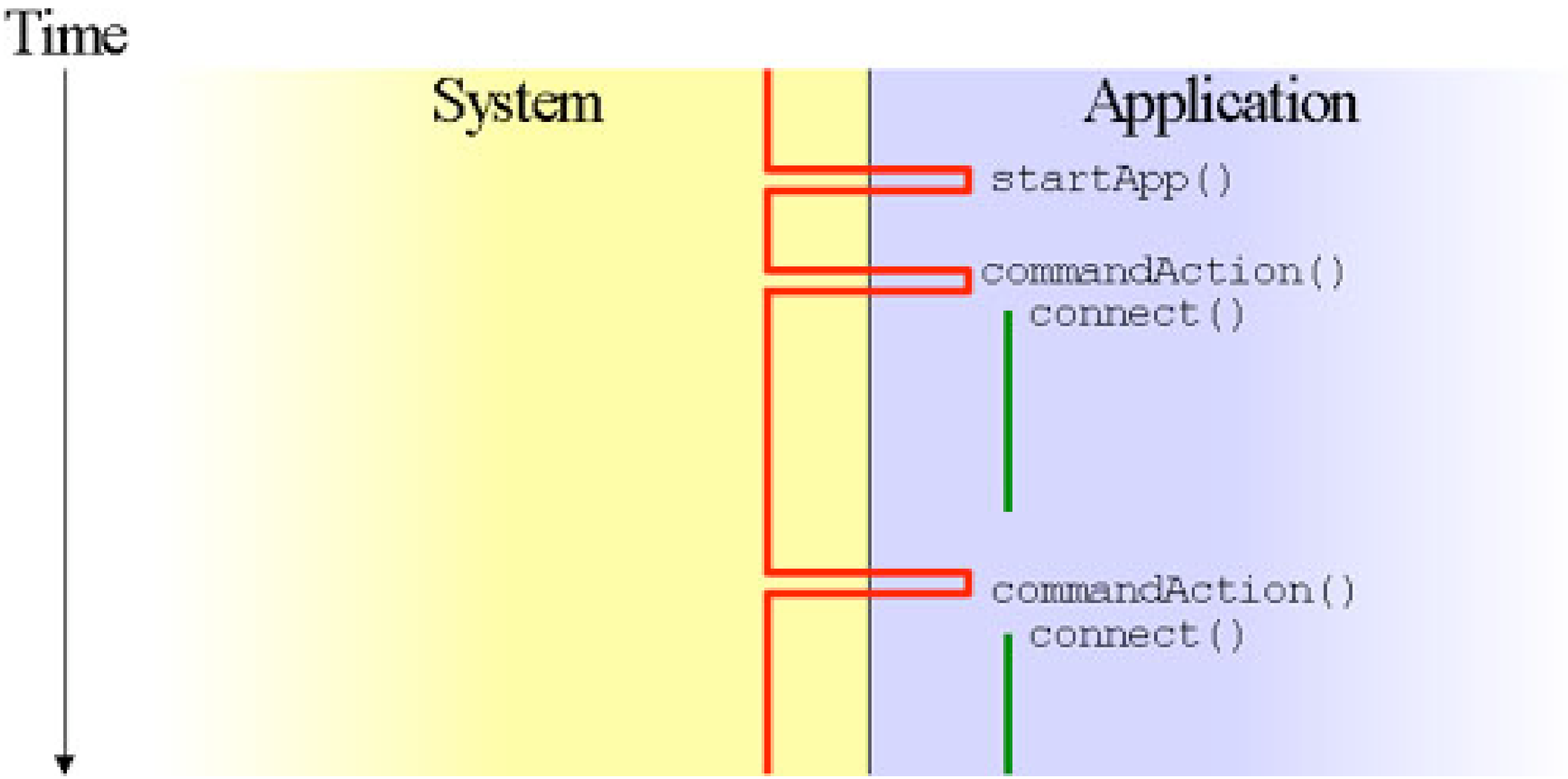
Single Threaded



Too many Threads!



Clean Threading



This Lecture

- Breymann, U., Mosemann, H.: Java ME – Anwendungsentwicklung für Handys, PDA und Co., Hanser, 2006, www.java-me.de
  - Chapter 10
- HTTP networking and threading in MIDP
  - http://developers.sun.com/techtopics/mobility/midp/articles/threading/
- Generic Connection Framework:
  - http://developers.sun.com/techtopics/mobility/midp/articles/genericframework/
  - http://developers.sun.com/techtopics/mobility/midp/articles/midp2network/

Thank you!

Dr. Thilo Horstmann

e-mail: thilo.horstmann@gmail.com
blog: http://www.das-zentralorgan.de