DESES: Middleware for Dynamic End-to-end Session Enhancing Services for Java Enabled Mobile Phones

Dimitris Kalofonos
Pervasive Computing Group
Nokia Research Center Cambridge
Cambridge, MA 02142, U.S.A.
dimitris.kalofonos@nokia.com

Parijat Shah
College of Engineering
Northeastern University
Boston, MA 02115, U.S.A.
parijat@coe.neu.edu

Elias Manolakos
ECE Department
Northeastern University
Boston, MA 02115, U.S.A.
elias@ece.neu.edu

6th International Workshop on Applications and Services in Wireless Networks (ASWN'06)
Berlin, Germany, May 30 2006

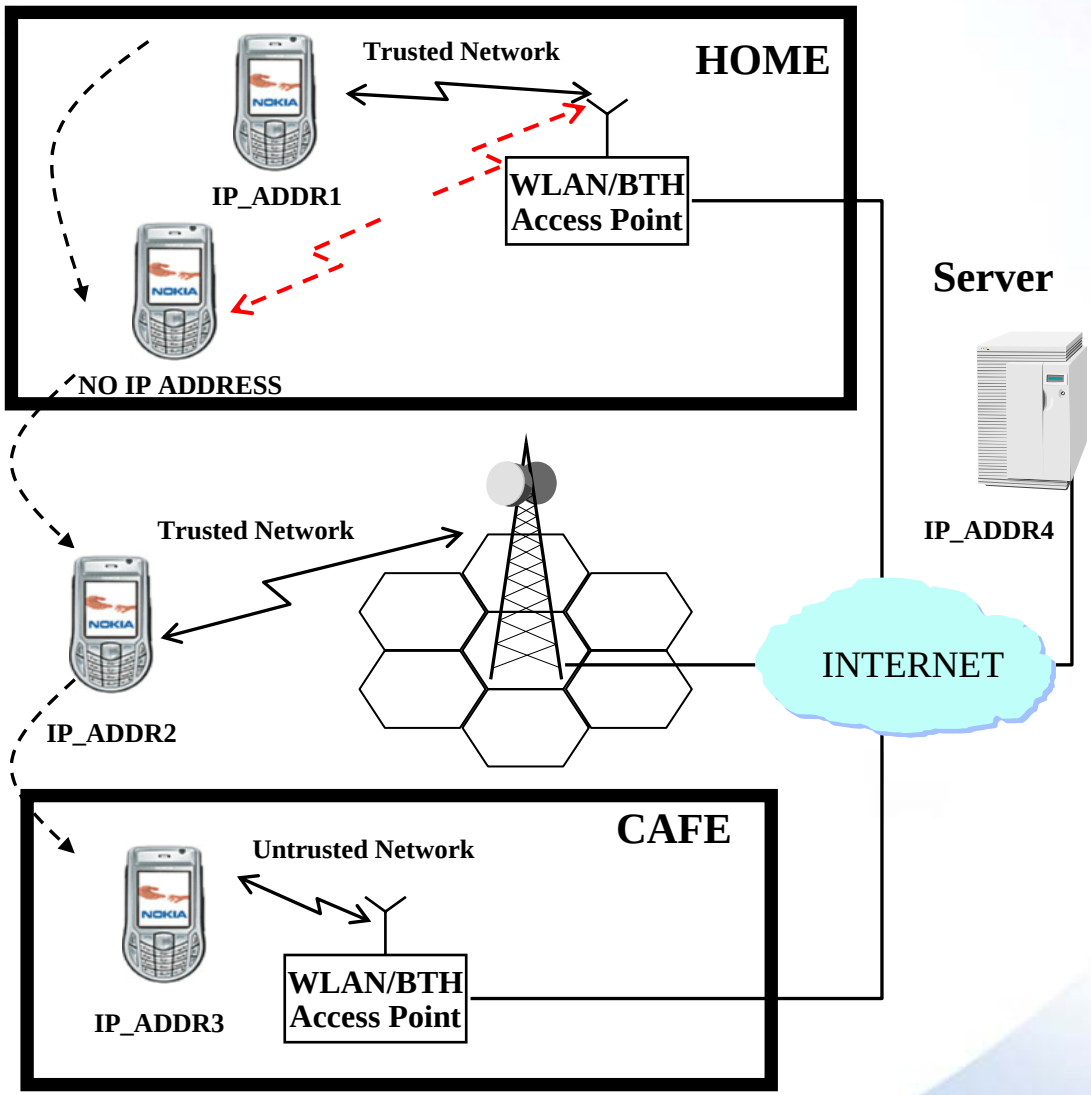
Outline

- Introduction
  - Motivation
  - Related work
- System Architecture and Design
- Implementation of DESES Library API
- Proof-of-concept prototype
- Conclusions

Project Motivation

- In infrastructure networks (e.g. cellular), dedicated servers and proxies can be used to provide session enhancing services, such as mobility support (e.g. mobile IP, SIP), compression, encryption. However,
  - this is an unlikely assumption in pervasive computing networks (e.g. heterogeneous, ad-hoc, P2P, unmanaged home networks)
  - deployment of any new infrastructure-based end-to-end services may require expensive roll out hardware and software upgrades
- The burden of e2e session enhancements cannot lie on application developers. It is desirable to offer support without requiring modifications to applications, platform and OS
- **Project Goal:** design and implement a session-layer middleware to enable development and deployment of end-to-end session enhancing services on mobile phones without requiring infrastructure support
  - Examples of include support against mobility and intermittent connectivity, end-to-end encryption, and end-to-end compression

An Example Use Case



Middleware Design Requirements

- It should provide end-to-end session enhancing services without requiring infrastructure support
- it should offer mobile platform independence as much as possible: use the pervasive Java Mobile Edition (J2ME)
- it should provide enhancements to legacy applications as transparently as possible
- it should provide an easy to use API for developers to develop new distributed applications
- it should be extensible to accommodate new session-enhancing services
- it should be able to negotiate which, if any, e2e session-enhancing services will be used at session establishment
- it should provide a mechanism to dynamically add and remove session-enhancing services at any time during an active session.

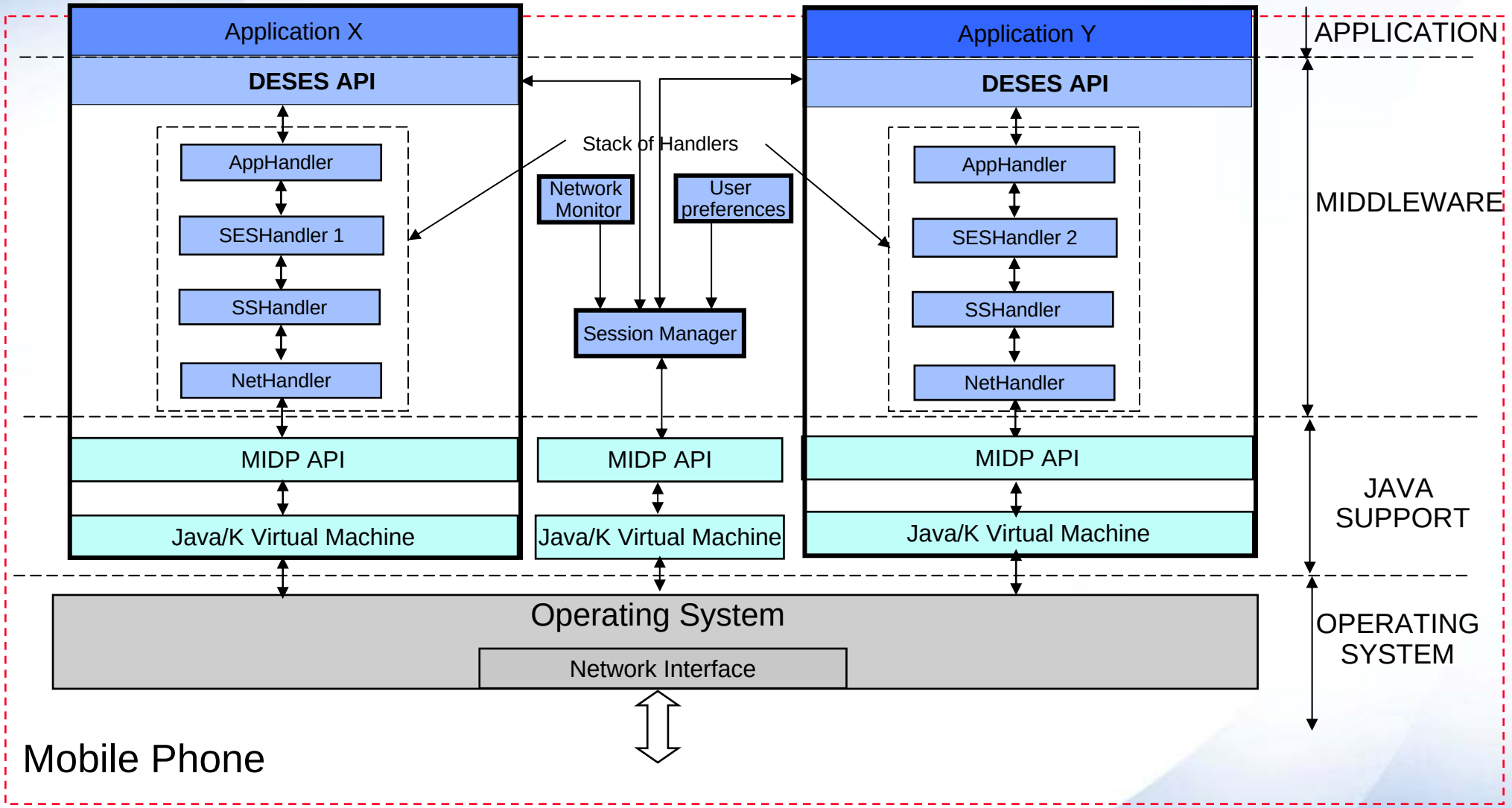
Related Work

- Tadashi Okoshi et al., "MobileSocket: Toward Continuous Operation for Java Application". Proc. of Int. Conf. on Comp. Comm. and Networks, 1999
  - provides end-to-end mobility support (intermittent connectivity, change of network attachment)
  - support for Java-based applications by importing a new MobileSocket API. Supports J2SE applications, no support for J2ME
  - not extensible to support other session services
- Victor C. Zandy et al., "Reliable Network Connections". Proc. Of Int. Conf. on Mobile Computing and Networking, 2002.
  - Linux/Unix based solution for native applications that provides transparent network mobility
  - no provision for using any other session service
- Jon Salz, "TESLA: A Transparent, Extensible Session-Layer Framework for End-to-End Network Services", Master's thesis at MIT, May 2002
  - library interposition between application and operating system (native applications on Linux/Unix)
  - a framework that enables different session services to be used
  - transparent, requires no modification to either application or kernel as library interposition is just a matter of changing environment variable
- Alex C. Snoeren, "MIGRATE: A Session-Based Architecture for Internet Mobility", Ph.D. thesis, MIT, Dec'02
  - uses Tesla's framework to provide end-to-end mobility support (intermittent disconnection, change of IP-address)
  - exports API that Migrate-aware applications can use to set user preferences and policies

Outline

- Introduction
  - Motivation
  - Related work
- System Architecture and Design
- Implementation of DESES Library API
- Proof-of-concept prototype
- Conclusions

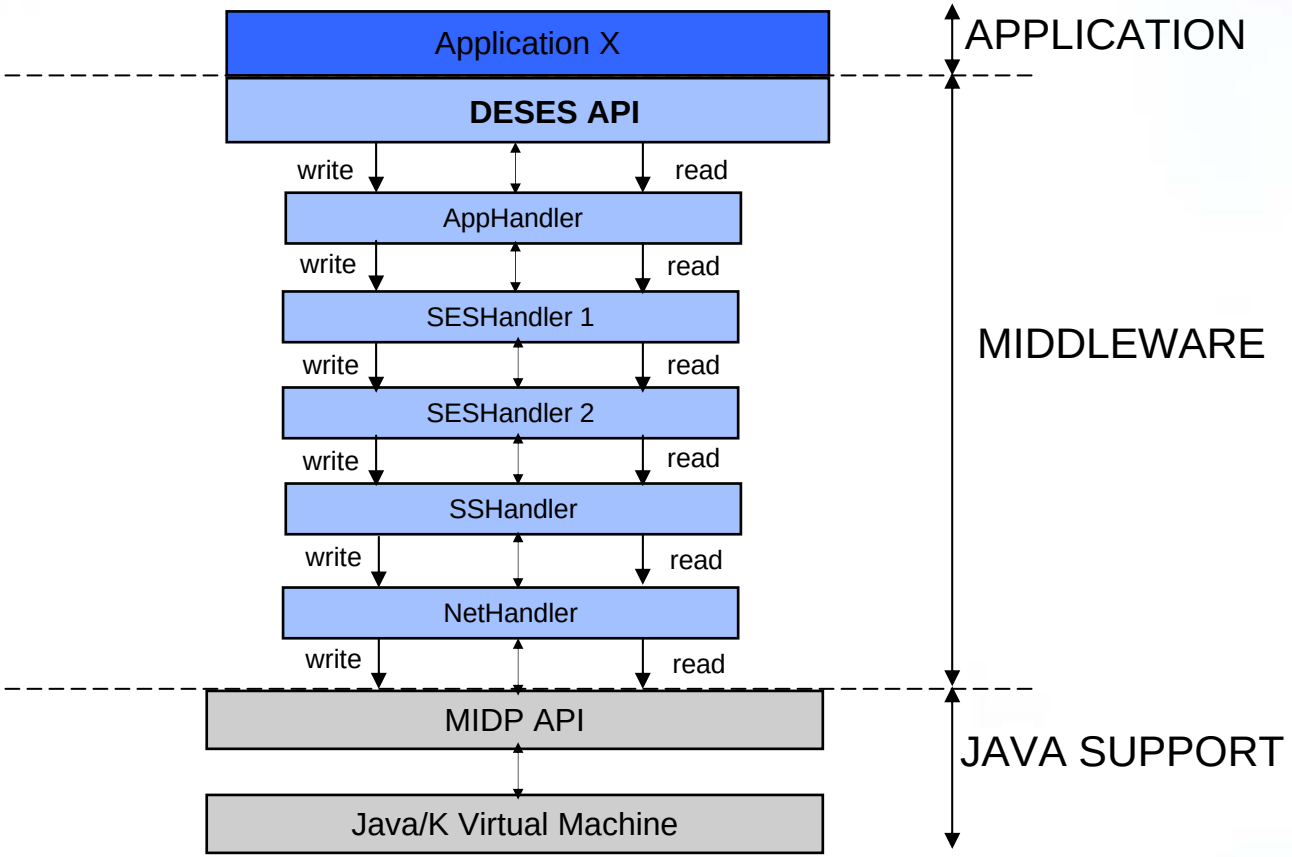
DESES Architecture Overview



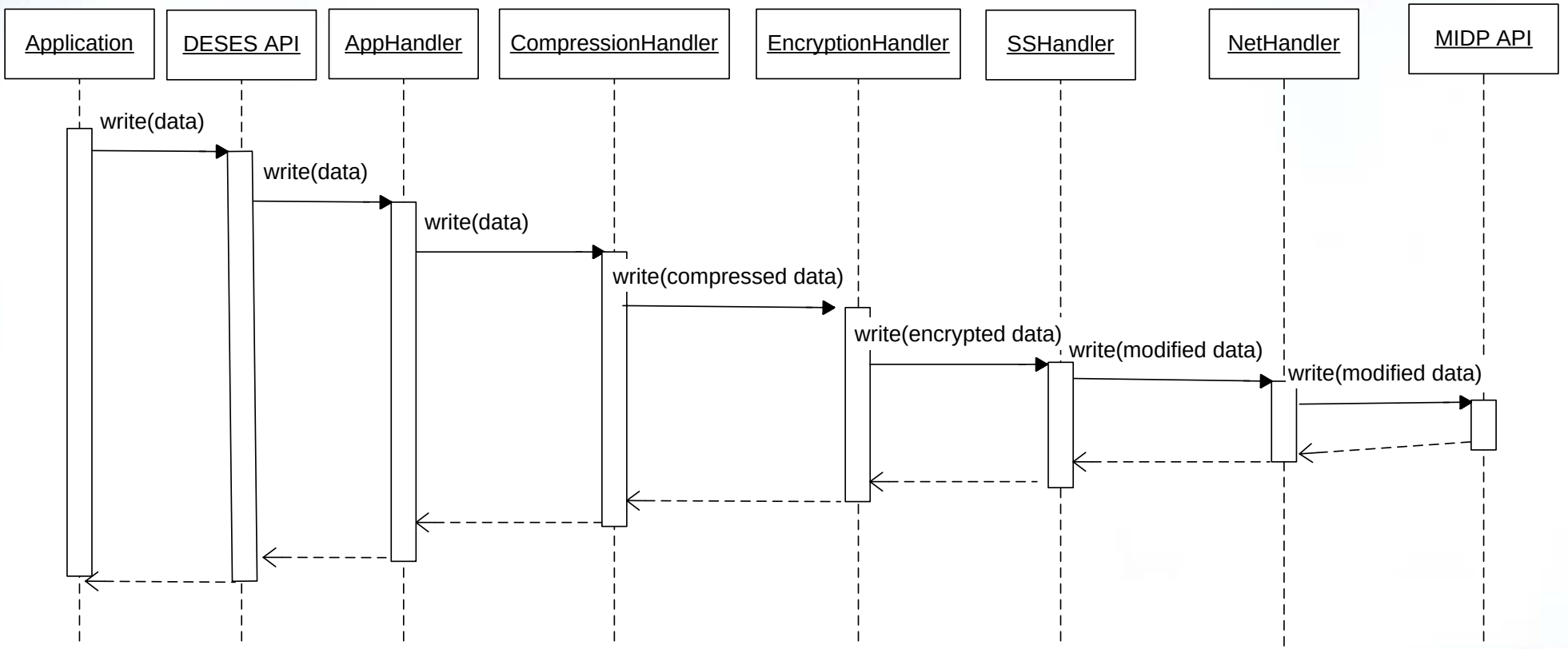
DESES Architecture Overview

- A stack of session service handlers operates on network flow. Each handler may or may not modify network flow
  - MasterHandler is the base class of all session handlers operating on the network flow
  - AppHandler, NetHandler, and SSHandler are derived from MasterHandler and are part of the middleware
- Session Manager
  - Provides end-to-end control between two hosts
  - Negotiate and agree on session services to be used
  - Renegotiation of SES in use
- Network Monitor and User Preferences
  - Network monitor notifies Session Manager about changes in network infrastructure
  - User preferences tell session manager about use/agent policies in certain events and conditions

DESES Architecture : Handler Interaction

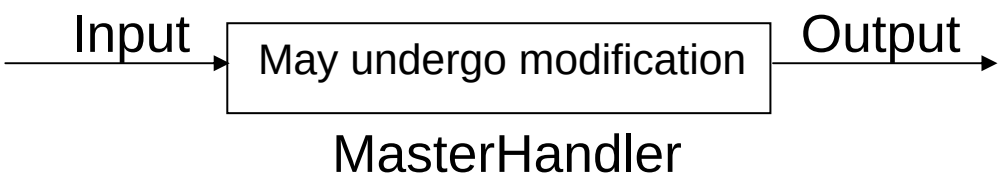


Example: an Application's write()



MasterHandler

- Takes a single network flow as input and produces a single network flow as output
- All session enhancing service handlers are derived from MasterHandler and override its methods to perform the desired functionality
- All session enhancing service handlers get plugged in as nodes in a directed graph

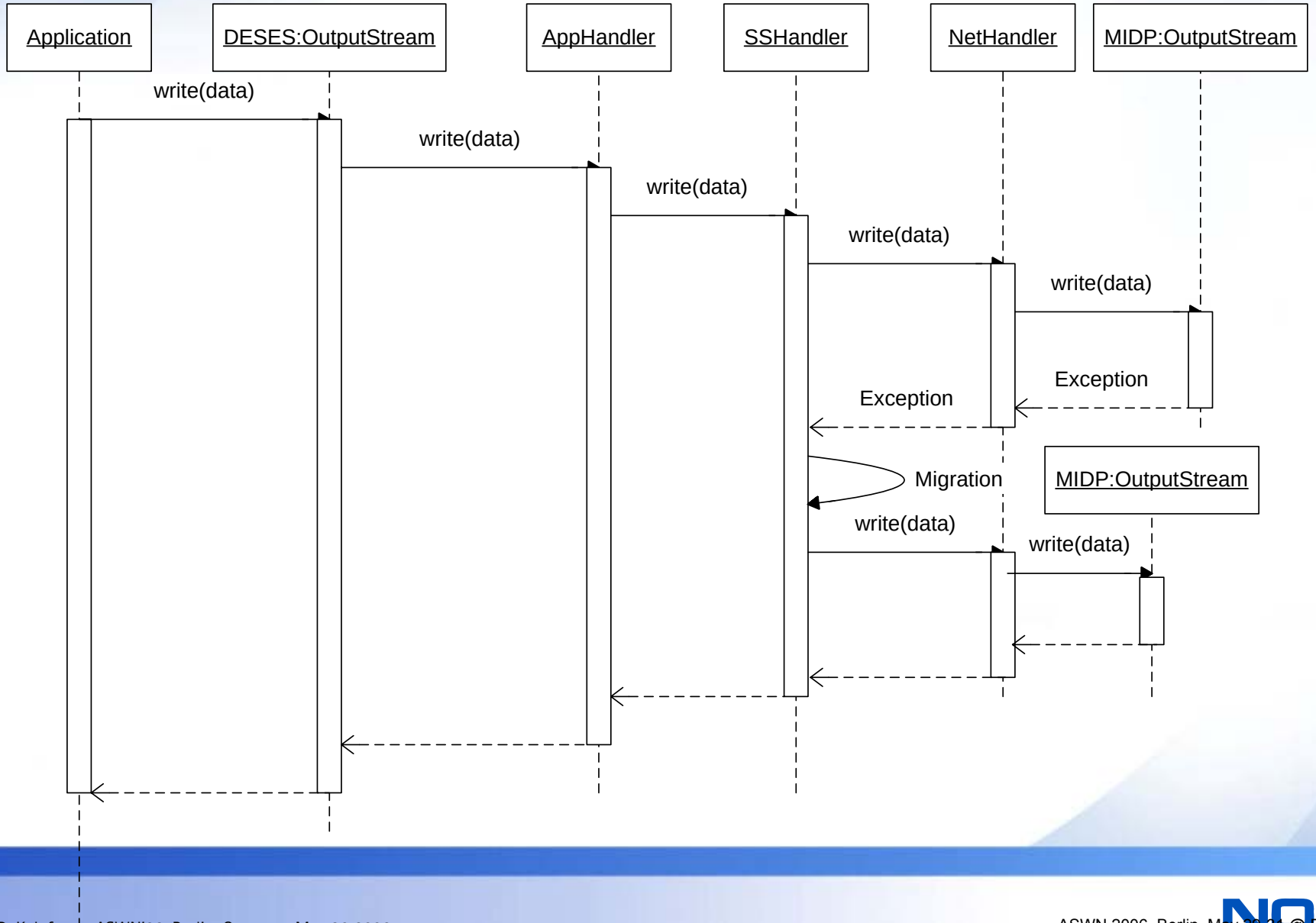



```
pubic class MasterHandler {
            pubilc  MasterHandler() {
            }
public InputStream openInputStream();
public OutputStream openOutputStream();
public DataInputStream openOutputStream();
public DataOutputStream openDataOutputStream();
public int read();
public long skip(long n);
public in available();
public synchronized void mark(int(int readlimit);
public boolean markSupported();
public int read(byte[],int off, int len);
public void iclose(); // similar to the close() of InputStream

public void write(int b);
public void write(byte b[], int off, int len);
public void flush();
public void oclose(); //similar to the close() of OutputStream

public String getLocalName();
public int getLocalPort();
public int getRemotePort();
public String getRemoteName();
public int getSocketOption(byte option);
public void nclose(); //similar to the close of SocketConnection
}
```

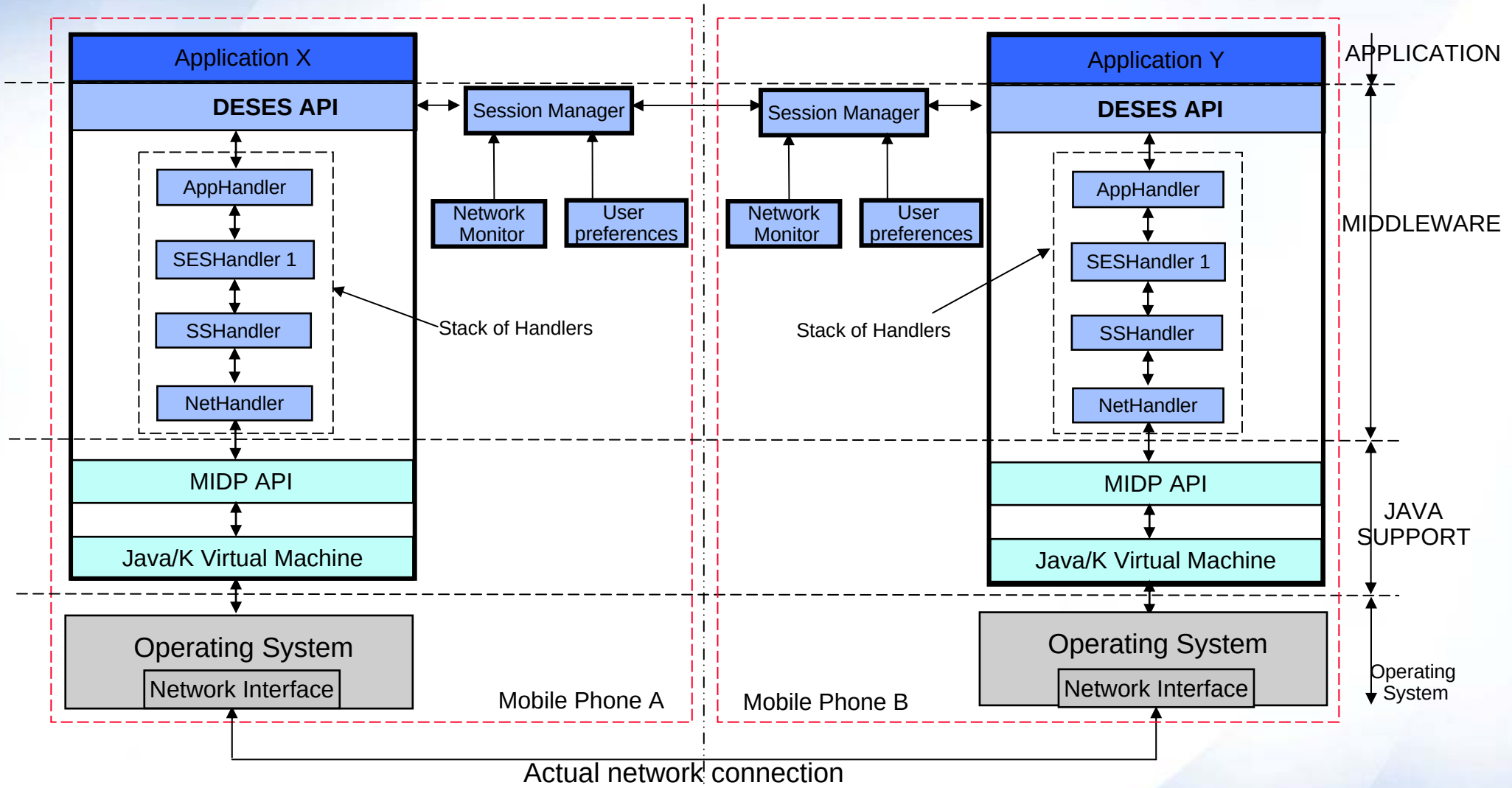
AppHandler, NetHandler and SSHandler

- AppHandler, NetHandler and SSHandler are required handlers of the DESES middleware and are present all the time
- AppHandler
  - interacts with the application
- SSHandler
  - facilitates the session renegotiation, where session enhancing services are added or removed while a session is active
  - provides support in the event of disconnection, and change of IP address
- NetHandler
  - interacts with the actual connected socket

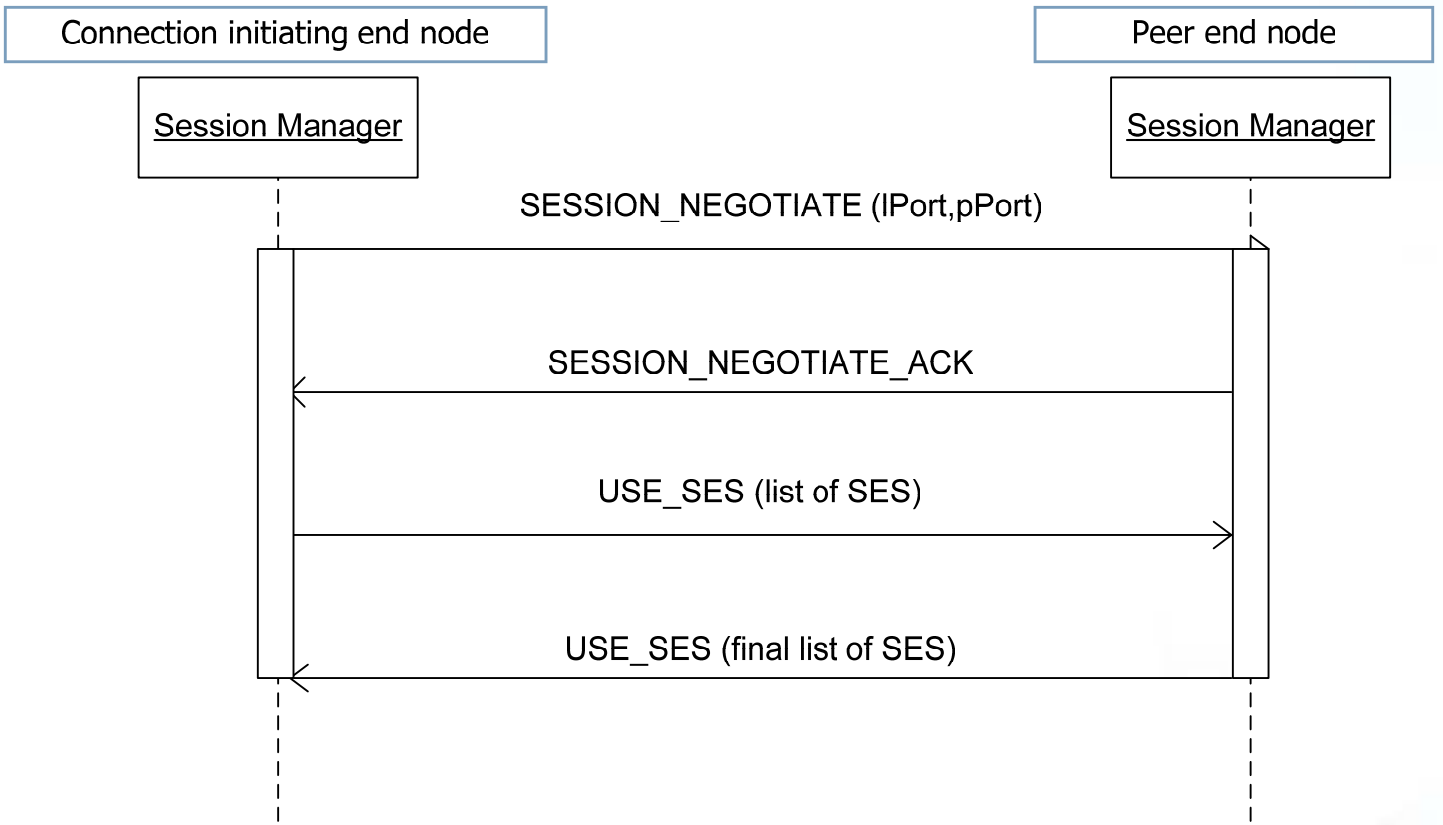
The Role of SSHandler



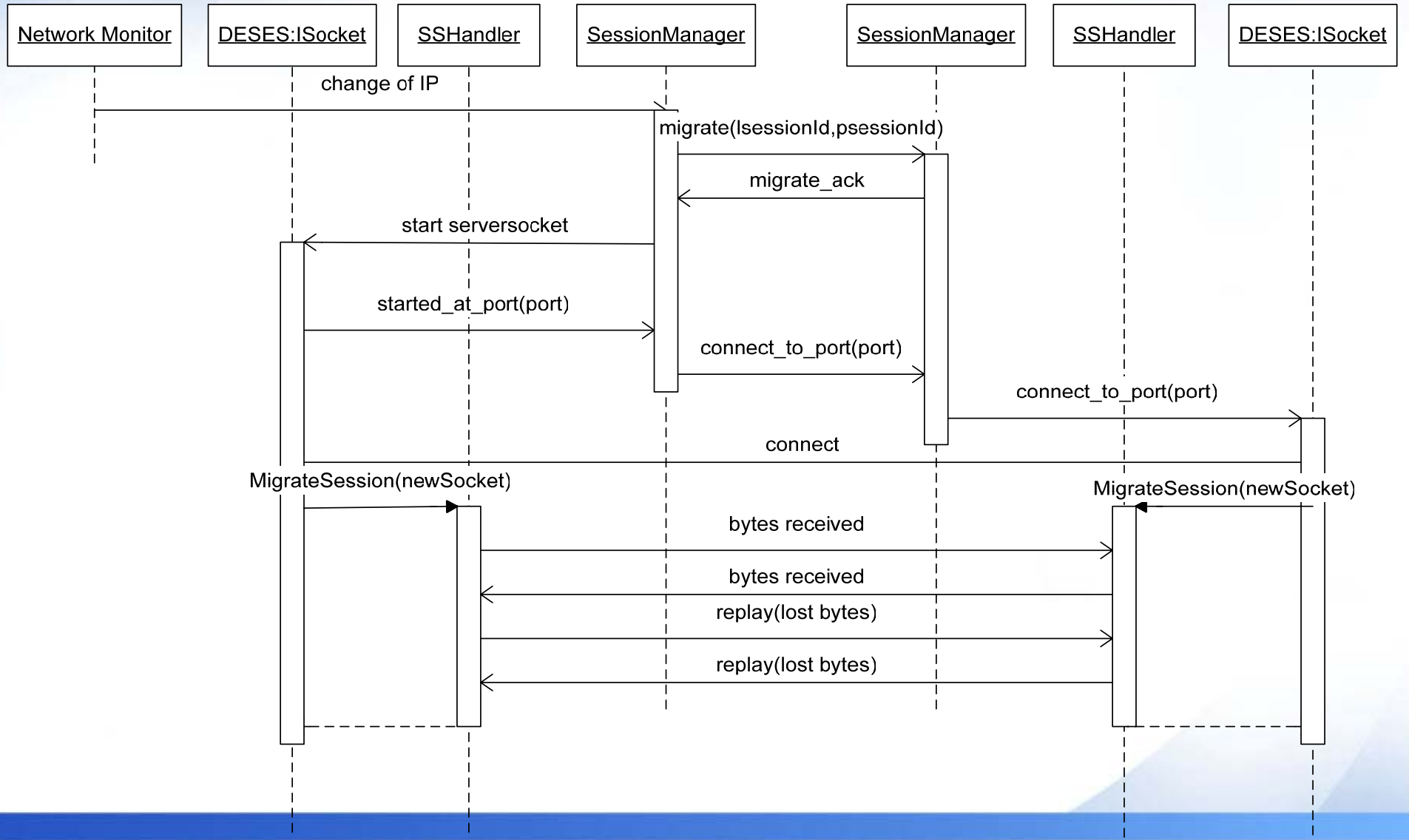
End-to-End Communication



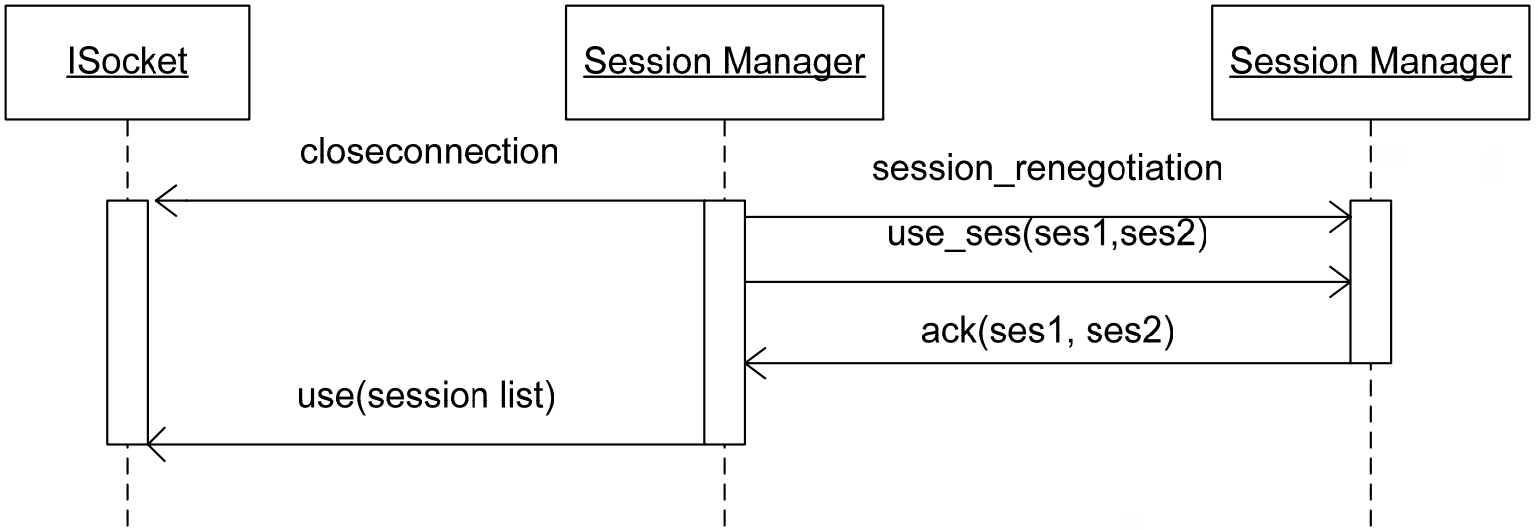
Session Negotiation



Session Resumption



Session Renegotiation



Outline

- Introduction
  - Motivation
  - Related work
- System Architecture and Design
- Implementation of DESES Library API
- Proof-of-concept prototype
- Conclusions

DESES Connection API

- DESES connection API provides interface hierarchy similar to the one provided by MIDP v2.0 generic connection framework
- Application Developers can directly use DESES connection API as follows

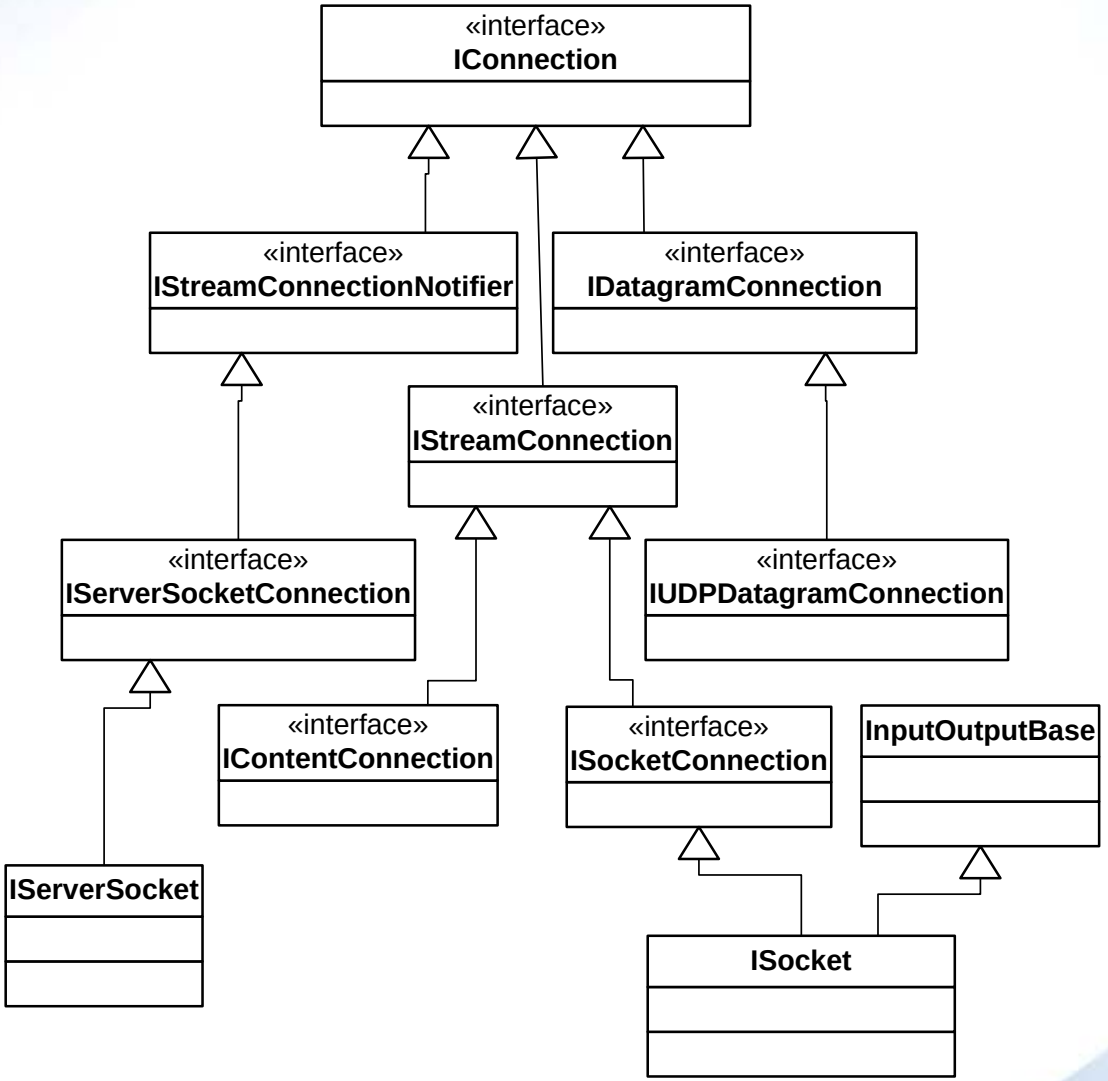
HTTP Connection: IConnector.open(http://abc.com);

SocketConnection: IConnector.open("socket//:ftp.sun.com:21);

ServerSocketConnection: IConnector.open("socket://1333");

- When connection is established DESES class implementing one of the Interfaces is returned
- ISocket class implements ISocketConnection interface
- IServerSocket class implements IServerSocketConnection interface

DESES API Interface and Class Hierarchy



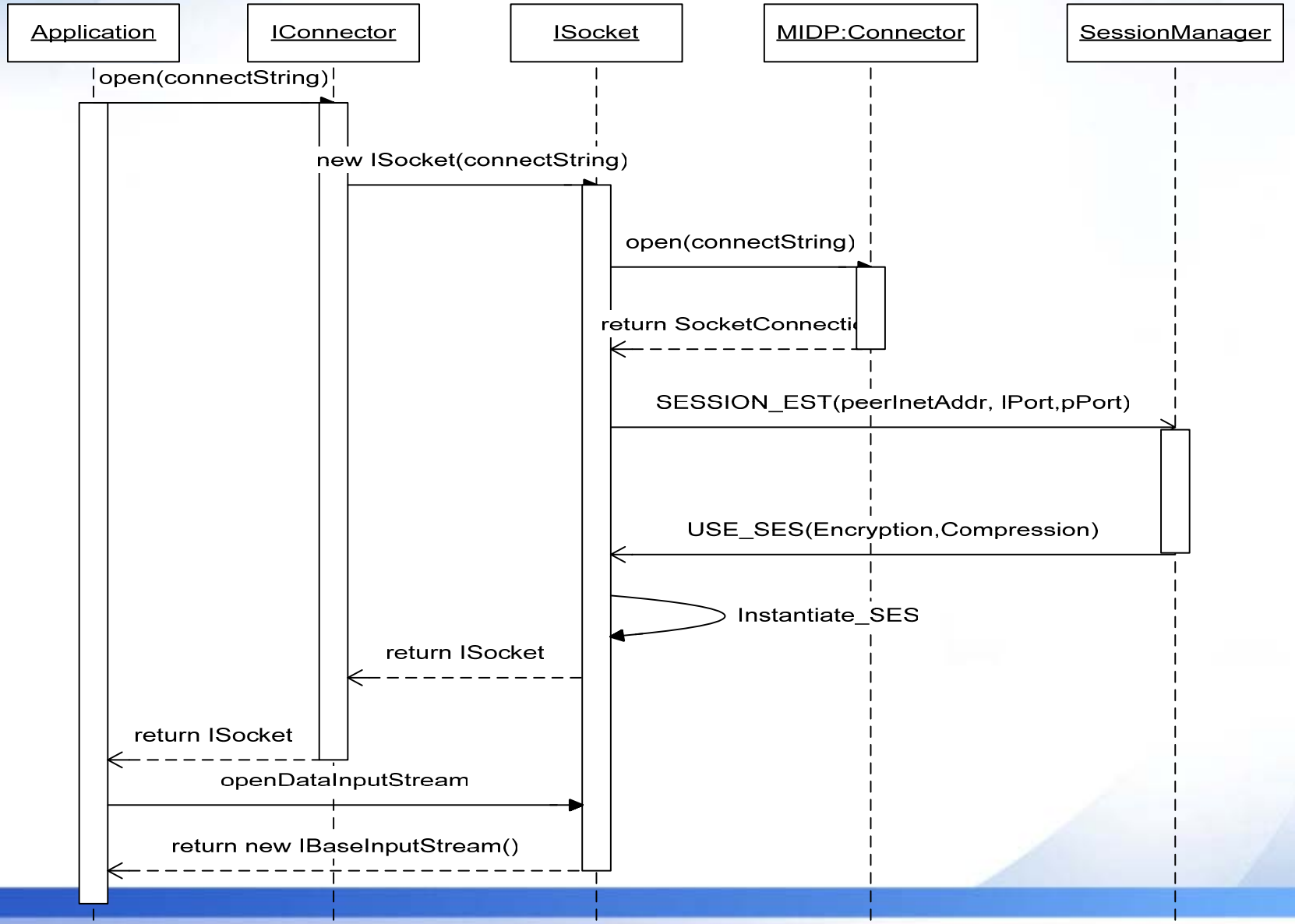
Intercepting Application Communication

- DESES defines class IBaseInputStream and IBaseOutputStream by extending InputStream and OutputStream and implementing abstract methods read and write
- When applications request InputStream and OutputStream DESES API returns the objects of IBaseInputStream and IBaseOutputStream
- Example:

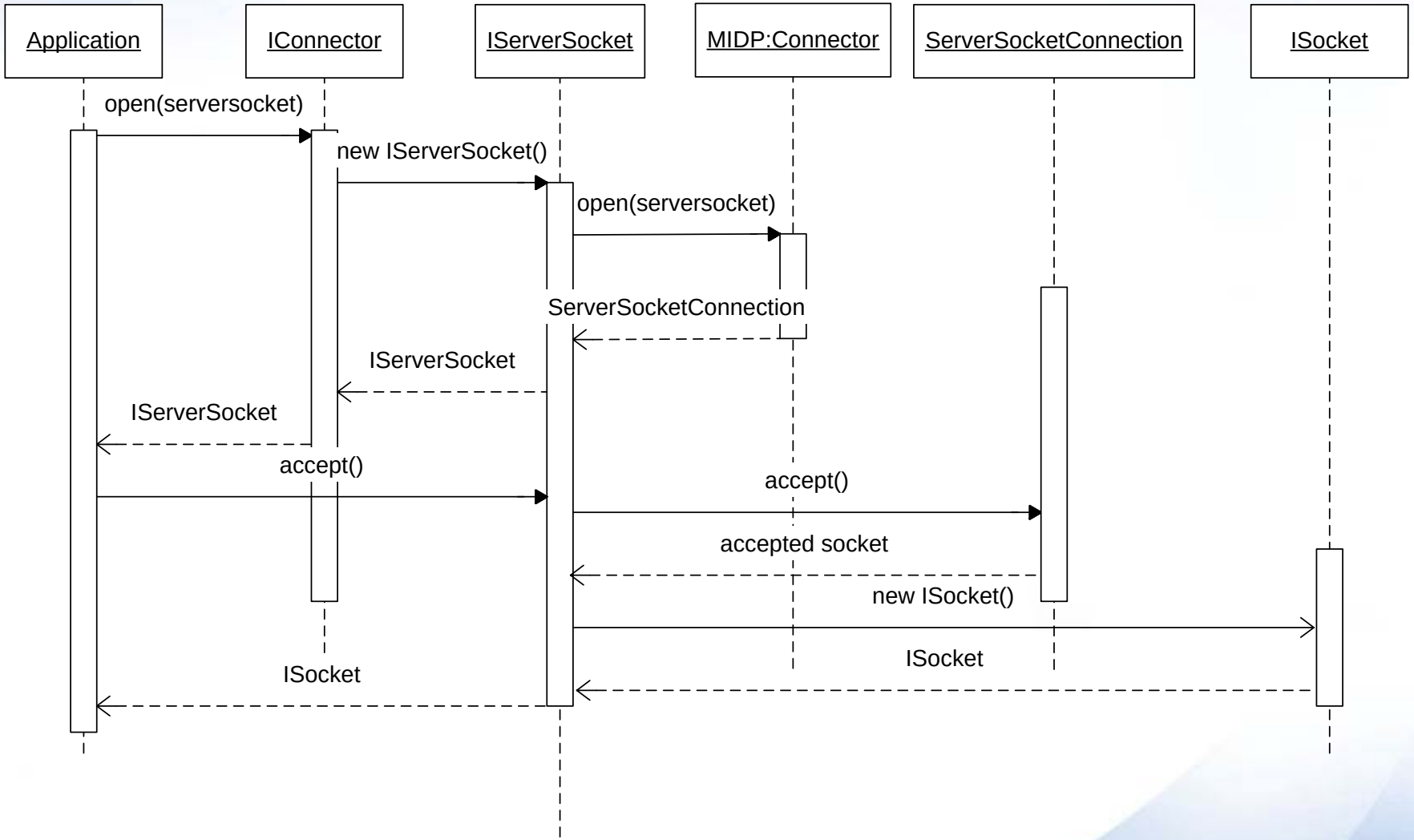
```
public abstract class OutputStream{
        public abstract int write( );
}
```

```
public class IBaseOutputStream extends OutputStream {

        public IBaseOutputStream( );

        public int write( ) {
                return AppHandler.write( );
        }
}
```

ISocket


Application
IConnector
ISocket
MIDP:Connector
SessionManager
open(connectString)
new ISocket(connectString)
open(connectString)
return SocketConnecti
SESSION_EST(peerInetAddr, lPort,pPort)
USE_SES(Encryption,Compression)
Instantiate_SES
return ISocket
return ISocket
openDataInputStream
return new IBaseInputStream()

IServerSocket



Bytecode Engineering

- Bytecode engineering provides the means
  - parse a java class bytecode file and modify string constants in constant_pool
  - replace the bytecode representing J2ME communication API with DESES API (e.g. Connector with IConnector)
- Pre-processing step: J2ME doesn't provide API to read and write files. It is necessary to modify class file offline (e.g. with a script) at install or launch time

```
ClassFile {
        u4 magic
        u2 minor_version
        u2 major_version
        cp_info_constant_pool[constant_pool_count – 1]
        u2 access_flags
        u2 this_class
        u2 super_class
        u2 interface_count
        u2 interface[interface_count]
        u2 fields_count
        field_info_fields[fields_count]
        u2 method_count
        method_info_method[method_count]
        u2 attribute_count
        attribute_info_attribute[attribute_count]
        }
```

Bytecode Engineering (cont)

An example of bytecode engineering

Original Class file

```
import javax.microedition.io.SocketConnection;
public Class Client
{
SocketConnection clientSocket;
clientSocket = (SocketConnection) Connector.open("socket://nimo:80);
}
```

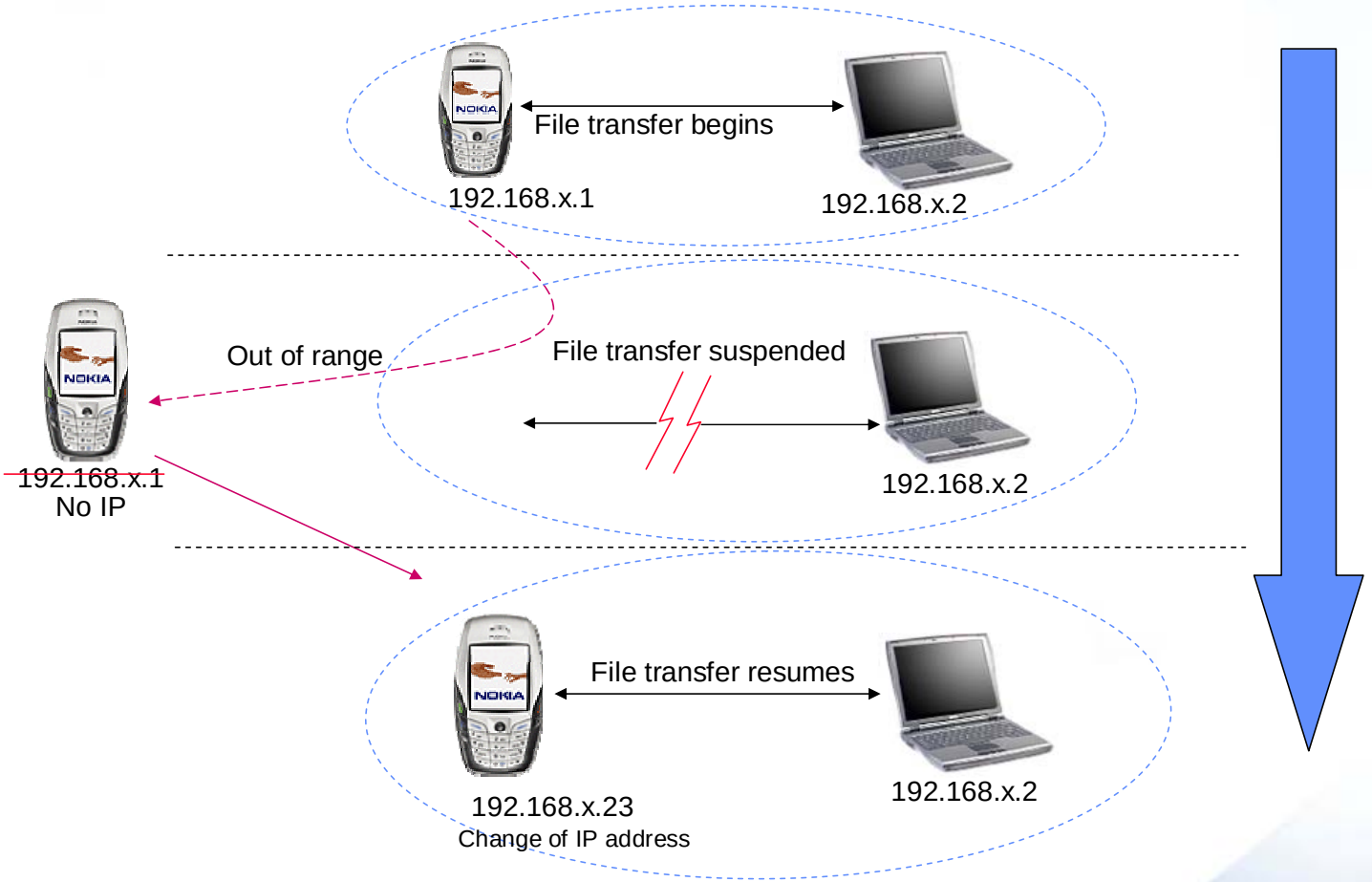
After Bytecode Engineering

```
import deses.connection.ISocketConnection;
public Class Client
{
ISocketConnection clientSocket;
clientSocket = (ISocketConnection) IConnector.open("socket://nimo:80);
}
```

Outline

- Introduction
  - Motivation
  - Related work
- System Architecture and Design
- Implementation of DESES Library API
- Proof-of-concept prototype
- Conclusions

Proof-of-concept Experiment



Implementation on Nokia 6600

- TCP send and receive buffer size
  - We restrict TCP send and receive buffer size to 4Kb each
- Threads and Networking
  - One connection one thread
- Exceptions
  - Any uncaught exceptions causes applications to crash
- OutputStream and Write
  - Data won't be written on to the network until it is flushed using flush method
- Number of Midlet
  - 1 additional MIDlet per device (session manager)
- Middleware size
  - 40Kb

Outline

- Introduction
  - Motivation
  - Related work
- System Architecture and Design
- Implementation of DESES Library API
- Proof-of-concept prototype
- Conclusions

Conclusions

- DESES targets mobile devices: end-to-end session enhancing services for J2ME applications on Java-enabled mobile phones
- Dynamic negotiation of session-support capabilities between two end-nodes at session establishment
- A mechanism to dynamically add and remove session enhancing services while a session is active (session renegotiation)
- Transparent session-support for legacy Java applications that requires no modifications (through use of Byte Code engineering)
- a working prototype implementation in actual mobile phones (Nokia 6600) using J2ME MIDP 2.0 on Symbian

Questions?

- For more information:

Email: dimitris.kalofonos@nokia.com

or

on the web: http://research.nokia.com