Java API

Proposal: 0007
Authors: Operator Foundation
Status: Implementation Complete
Implementation: https://github.com/OperatorFoundation/shapeshifter-dispatcher/tree/v3

Introduction

There are currently two language-specific APIs for Pluggable Transports: Go and Swift. This proposal adds a third language-specific API for Java.

Motivation

Java is the primary language used for writing Android applications, a platform which has many users of Pluggable Transports. Currently, Android developers use the Go API to integrate Pluggable Transports into their applications. However, Android developers have requested a native Java API. Similar to the Swift API, the purpose of having a Java API would be to provide native Java implementations of popular Pluggable Transports for easy integration into applications written in Java.

Proposed solution

As with other language-specific Pluggable Transport APIs, the Java API mimics the existing Java networking API, allowing for socket-like client connections. There are two different socket APIs in Java: classic and NIO, with the NIO being more popular in server-side applications and classic being more common in client-side applications such as Android applications. This document describes a classic socket implementation targeted to Android client applications.

Design

Classic networking in Java is handled by the java.net package. TCP connections are handled by the Socket class. Switching an application from using direct classic networking to transports therefore only requires changing from using Socket to the equivalent class provided by the transport.

1. Java Transport API

The Pluggable Transport Java API provides a way to use Pluggable Transports directly from

Java code. It is an alternative to using the IPC interface. The Java API may be an appropriate choice when the application and the required transport are both written in Java. When either the application or the transport are written in a different language, the IPC interface provides a language-neutral method for configuring and using transports running in a separate process.

This API specification is divided into three parts. The “Modules” section provides documentation of the types and methods provided by the Pluggable Transports Java API. The “Implementing a Transport” and “Using a Transport” sections then provide context on how the API is used.

1.1 Modules

The Pluggable Transports Java API does not define any types. Rather, it reuses the existing classes from the Java standard library’s java.net module. Key parts of these are provided here for reference.

1.1.1 Module java.net

For client connections, Java classic networking uses the Socket class.[1]

```
class Socket
{
    Socket(String host, int port)
    Socket(String host, int port, InetAddress localAddr, int localPort)
    Socket(InetAddress address, int port)
    Socket(InetAddress address, int port, InetAddress localAddr, int
localPort)
    Socket(Proxy proxy)

    void bind(SocketAddress bindpoint)
    void close()
    void connect(SocketAddress endpoint)
    void connect(SocketAddress endpoint, int timeout)
    SocketChannel getChannel()
    InetAddress getInetAddress()
    InputStream getInputStream()
    boolean getKeepAlive()
    InetAddress getLocalAddress()
    int getLocalPort()
    SocketAddress getLocalSocketAddress()
    boolean getOOBInline()
    OutputStream getOutputStream()
    int getPort()
    int getReceiveBufferSize()
    SocketAddress getRemoteSocketAddress()
    boolean getReuseAddress()
    int getSendBufferSize()
    int getSoLinger()
    int setSoTimeout()
    boolean getTcpNoDelay()
    int getTrafficClass()
    boolean isBound()
    boolean isClosed()
    boolean isConnected()
    boolean isInputShutdown()
    boolean isOutputShutdown()
    void sendUrgentData(int data)
    void setKeepAlive(boolean on)
    void setOOBInline(boolean on)
    void setPerformancePreferences(int connectionTime, int latency,
int bandwidth)
    void setReceiveBufferSize(int size)
    void setReuseAddress(boolean on)
    void setSendBufferSize(int size)
    void setSoLinger(boolean on, int linger)
    void setSoTimeout(int timeout)
    void setTcpNoDelay(boolean on)
    void setTrafficClass(int tc)
    void shutdownInput()
    void shutdownOutput()
    String toString()
}
```

Additionally, this class references the InputStream and OutputStream abstract classes, which will also need to be implemented by the transport.

```
public abstract class InputStream extends Object implements Closeable
{
    int available()
    void close()
    void mark(int readlimit)
    boolean markSupported()
    abstract int read()
    int read(byte[] b)
    int read(byte[] b, int off, int len)
    void reset()
    long skip(long n)
}

public abstract class OutputStream implements Closeable, Flushable
{
    void close()
    void flush()
    void write(byte[] b)
    void write(byte[] b, int off, int len)
    abstract void write(int b)
}
```

It is important to note that Socket, InputStream, and OutputStream are classes, not interfaces. Therefore, transports classes need to inherit from these system classes in order to be used as replacements for traditional sockets in the application code.

1.2 Implementing a Transport

In order to provide a socket-like interface, the transport must provide a class that inherits from java.net.Socket, as well as classes that inherit from java.io.InputStream and java.io.OutputStream. Therefore, a transport implementation in Java is split across at least three classes. Not all methods need to be implemented as they will be inherited from the superclasses. However, the superclass behavior may not always be appropriate. The specific implementation strategy is left to the discretion of the transport implementer.

The crucial components of Socket are the initializers and the connect(), close(), getInputStream(), and getOutputStream() functions. The most important method on InputStream is read() and the most important method on OutputStream is write(). It is important to note that InputStream and OutputStream have multiple read() and write() methods and they should all be implemented. Implementing just "`int read()`" and "`void write(int b)`" will incur an unacceptable performance penalty.

1.4 Using a Transport

Switching an application from using direct classic networking to transports therefore only requires changing from using Socket to the equivalent classes provided by the transport:

```
Socket socket = TransportSocket("127.0.0.1", 1234);
socket.write(123);
int result = socket.read();
```

3. References

1. https://docs.oracle.com/javase/10/docs/api/java/net/Socket.html
2. https://docs.oracle.com/javase/10/docs/api/java/io/InputStream.html

3. https://docs.oracle.com/javase/10/docs/api/java/io/OutputStream.html

Effect on API Compatibility

This is a new API and has no effect on existing APIs.

Effect on IPC Compatibility

This change only adds a new API. There is no effect on IPC compatibility.

Alternatives considered

An alternative to developing native Java implementations of existing transports would be to add Java bindings around the existing implementation of the Go API. However, in order for this to be modular, it would still require defining a Java API. While the motivation of this proposal is to provide native Java implementations, the API would be the same regardless of the method of implementation. There are already users using the Go implementation in their Java applications. If they were to standardize their custom wrapper implementations to conform to the Java API in this document, they could swap between Go transports and native Java transports interchangeably, which would only be of benefit to application developers.