Java Transport API

Version 1.0

Abstract
Pluggable Transports (PTs) are a generic mechanism for the rapid development and deployment of censorship circumvention, based around the idea of modular transports that transform traffic to defeat censors.

The Transport APIs define a set of language-specific APIs to use transports directly from within an application. This document describes the Transport API for the Java programming language, as well as associated implementation details for using and creating pluggable transports that are utilized directly in a Java or Kotlin application by linking to a library of transport implementations. To use this interface, the application makes calls to library functions to configure the transports and sends and receives data using functions that provide a socket-like interface. This interface is most useful if you are seeking to build a transport directly into your application which is also written in the Java or Kotlin programming languages. It can be the simplest mode of PT integration if the client should be a single binary and should run in a single process. It is primarily targeted for use in Android applications and not other places where Java can be deployed such as on servers or in embedded devices. The Java API is compatible with JVM version of Kotlin for Android devices and not any other variants such as Kotlin Native.

Table of Contents

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 main 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.

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.

1.1 Logging

For each language-specific API, there is a language-specific logging mechanism, with standardized log levels used across all languages. In Java, adding logging support is done by having the factory initializers take a logger argument that can be used for logging inside of the transport. This logger is provided by the application that is using the transport. The built-in standard library logging system java.util.Logging will be used. The full documentation for java.util.Logging can be found here: https://docs.oracle.com/javase/10/core/java-logging-overview.htm#JSCOR-GUID-B83B652C-17EA-48D9-93D2-563AE1FF8EDA

1.2 Modules

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.

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.

1.2.1 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.

1.3. 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.

1.3.1 Logging and Constructors

Socket implementations are required to have certain constructors. In order to implement the logging specification, additional constructors can be added that take a logger parameter:

```
import java.util.Logging

public class TransportSocket extends Socket
{
      private Logger logger;

      Socket(String host, int port, logger Logger)
      {
            super(host, port);
            self.logger = logger;
      }
}
```

1.3.2 Using Java Logging from Inside a Transport

Import the logging library

```
import java.util.logging.*;
```

The logger has functions for each loglevel. The permitted logging functions are severe, info, warning, and fine. The names for the loglevels in Java differ from the other language-specific

logging APIs, but correspond to the same ERROR, INFO, WARNING, and DEBUG loglevels used in the Transport APIs for other languages.

Examples of logging:

```
logger.fine("debug”); // This corresponds to the DEBUG log level
logger.info("info");
logger.warning("warning");
logger.severe("error"); // This corresponds to the ERROR log level
```

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();
```

1.4.1 Setting up a Log in a Java Application

Applications must provide the logger to the transport. The logger can be configured with application-specific options.

Import the logging library

```
import java.util.logging.*;
```

Create a logger

```
Logger logger = Logger.getLogger("com.example.App");
```

Optionally create a logging backend, otherwise the default backend will be used.

```
Handler handler = MemoryHandler();
```

This backend stores logs in memory.

Optionally, set the loglevel for the backend, otherwise the default loglevel will be used.

```
handler.setLevel(Level.INFO);
```

This sets the backend’s loglevel to INFO.

Finally, set the backend for the log.

```
logger.addHandler(handler);
```

Now that the logger has been initialized, a logging-capable constructor can be called for the transport:

```
Socket transport = TransportSocket(“127.0.0.1”, 1234, logger);
```

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

Appendix A. Changelog

PT 3.0

- Initial implementation of proposal 0007 - Java API v1.0