Java API

Proposal: 0007
Authors: Operator Foundation
Status: Initial proposal

Implementation: TBD

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 PT APIs, the Java API will mimic the existing Java networking API, allowing for socket-like client connections, as well as socket-like listeners and server connections. There are two different socket APIs in Java: classic and NIO. Therefore, there will be two transport APIs for Java, one for each of the Java socket APIs.

Design

Classic networking in Java is handled by the java.net package. TCP connections are handled by the Socket and ServerSocket classes. UDP networking is handled by the DatagramSocket class. However, these classes delegate to a SocketImpl class, which is provided by a configurable SocketImpFactory. Switching an application from using direct classic networking to transports therefore only requires changing the SocketImplFactory.

NIO networking in Java is handled by the java.nio.channels.SocketChannel and java.nio.channels.ServerSocketChannel classes. However, these classes delegate to a

SelectorProvider class which acts as a factor to create new socket objects. Switching an application from using direct NIO networking to transports therefore only requires changing the SelectorProvider.

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 class type. Rather, it reuses the existing classes from the Java standard library’s java.net and java.nio.channels modules. Key parts of these are provided here for reference.

1.1.1 Module java.net

This is the classic networking factory for both client and server TCP connections:

```
interface SocketImplFactory
{
     public SocketImpl createSocketImpl()
}
```

This is the implementation for classic TCP sockets, both client and server connections. This is an abstract class that must be subclassed by the transport implementation. Only the abstract methods that the transport must implement are shown here and documentation on these methods is omitted for space. For comprehensive documentation of this class, see the official Java documentation [FIXME - reference]

```
public abstract class SocketImpl implements SocketOptions
{
    protected abstract void create(boolean stream) throws
IOException;
    protected abstract void connect(String host, int port) throws
IOException;
```

```
    protected abstract void connect(InetAddress address, int port)
throws IOException;
    protected abstract void connect(SocketAddress address, int
timeout) throws IOException;
    protected abstract void bind(InetAddress host, int port) throws
IOException;
    protected abstract void listen(int backlog) throws IOException;
    protected abstract void accept(SocketImpl s) throws IOException;
    protected abstract InputStream getInputStream() throws
IOException;
    protected abstract OutputStream getOutputStream() throws
IOException;
    protected abstract int available() throws IOException;
    protected abstract void close() throws IOException;
    protected abstract void sendUrgentData (int data) throws
IOException;
}
```

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
{
    public InputStream();
    public int available() throws IOException;
    public void close() throws IOException;
    public abstract int read() throws IOException;
    public int read(byte[] buffer) throws IOException;
    public int read(byte[] buffer, int byteOffset, int byteCount)
throws IOException;
}

public abstract class OutputStream implements Closeable, Flushable
{
    public OutputStream();
    public void close() throws IOException;
    public void flush() throws IOException;
    public void write(byte[] buffer) throws IOException;
    public void write(byte[] buffer, int offset, int count) throws
IOException;
    public abstract void write(int oneByte) throws IOException;
}
```

This is the factory for both client and server UDP:

```
public interface DatagramSocketImplFactory
{
     DatagramSocketImpl createDatagramSocketImpl();
}
```

This is the implementation for UDP sockets, both client and server. This is an abstract class that must be subclassed by the transport implementation. Only the abstract methods that the transport must implement are shown here and documentation on these methods is omitted for space. For comprehensive documentation of this class, see the official Java documentation [FIXME - reference]

```
public abstract class DatagramSocketImpl implements SocketOptions {
    protected abstract void create() throws SocketException;
    protected abstract void bind(int lport, InetAddress laddr) throws
SocketException;
    protected abstract void send(DatagramPacket p) throws
IOException;
    protected abstract int peek(InetAddress i) throws IOException;
    protected abstract int peekData(DatagramPacket p) throws
IOException;
    protected abstract void receive(DatagramPacket p) throws
IOException;
    protected abstract void setTimeToLive(int ttl) throws
IOException;
    protected abstract int getTimeToLive() throws IOException;
    protected abstract void join(InetAddress inetaddr) throws
IOException;
    protected abstract void leave(InetAddress inetaddr) throws
IOException;
    protected abstract void joinGroup(SocketAddress mcastaddr,
                                      NetworkInterface netIf)
        throws IOException;
    protected abstract void leaveGroup(SocketAddress mcastaddr,
                                       NetworkInterface netIf)
        throws IOException;
    protected abstract void close();
}
```

1.1.2 Module java.nio.channels

This is the NIO networking factory for both client and server TCP connections and UDP:

```
public abstract class SelectorProvider
{
    public abstract DatagramChannel openDatagramChannel() throws
IOException;
    public abstract DatagramChannel
openDatagramChannel(ProtocolFamily family)
        throws IOException;

    public abstract ServerSocketChannel openServerSocketChannel()
        throws IOException;
    public abstract SocketChannel openSocketChannel()
        throws IOException;
}
```

NIO TCP connections are represented by SocketChannel for client connections and ServerSocketChannel for server connections and listeners.

```
public abstract class SocketChannel
{
    public static SocketChannel open() throws IOException;
    public static SocketChannel open(SocketAddress remote)
        throws IOException;
    public abstract SocketChannel bind(SocketAddress local)
        throws IOException;
    public abstract <T> SocketChannel setOption(SocketOption<T> name,
T value)
        throws IOException;
    public abstract SocketChannel shutdownInput() throws IOException;
    public abstract SocketChannel shutdownOutput() throws
IOException;
    public abstract Socket socket();
    public abstract boolean isConnected();
    public abstract boolean isConnectionPending();
    public abstract boolean connect(SocketAddress remote) throws
IOException;
    public abstract boolean finishConnect() throws IOException;
    public abstract SocketAddress getRemoteAddress() throws
IOException;
    public abstract int read(ByteBuffer dst) throws IOException;
    public abstract long read(ByteBuffer[] dsts, int offset, int
length)
        throws IOException;
    public abstract int write(ByteBuffer src) throws IOException;
    public abstract long write(ByteBuffer[] srcs, int offset, int
length)
        throws IOException;
}

public abstract class ServerSocketChannel
{
    public abstract ServerSocketChannel bind(SocketAddress local, int
backlog) throws IOException;
    public abstract <T> ServerSocketChannel setOption(SocketOption<T>
name, T value) throws IOException;
    public abstract ServerSocket socket();
    public abstract SocketChannel accept() throws IOException;
}
```

NIO UDP is represented by DatagramChannel for both client and server.

```
public abstract class DatagramChannel
{
    public static DatagramChannel open() throws IOException;
    public static DatagramChannel open(ProtocolFamily family) throws
IOException;

    public abstract DatagramChannel bind(SocketAddress local)
        throws IOException;
    public abstract <T> DatagramChannel setOption(SocketOption<T>
name, T value)
        throws IOException;
    public abstract DatagramSocket socket();
    public abstract boolean isConnected();
    public abstract DatagramChannel connect(SocketAddress remote)
        throws IOException;
    public abstract DatagramChannel disconnect() throws IOException;
    public abstract SocketAddress getRemoteAddress() throws
IOException;
    public abstract SocketAddress receive(ByteBuffer dst) throws
IOException;
    public abstract int send(ByteBuffer src, SocketAddress target)
        throws IOException;
    public abstract int read(ByteBuffer dst) throws IOException;
    public abstract long read(ByteBuffer[] dsts, int offset, int
length)
        throws IOException;
    public abstract int write(ByteBuffer src) throws IOException;
    public abstract long write(ByteBuffer[] srcs, int offset, int
length)
        throws IOException;
}
```

1.1.3 Transport Modules for Classic Java Networking

Transport modules must provide constructor functions. A module must provide one function for both client and server connections. These functions take transport-specific parameters, so there is no fixed interface definition. The functions below are just examples for what these function signatures might look like for a hypothetical transport:

```
// Factory for TCP connections, both client and server
public class TransportSocketImplFactory extends SocketImplFactory
{
     private String config;

     public TransportSocketImplFactory(String config)
     {
          self.config = config;
     }

     public SocketImpl createSocketImpl()
     {
          return TransportSocketImpl(self.config);
     }
}

//Factory for UDP, both client and server
public class TransportDatagramSocketImplFactory extends
DatagramSocketImplFactory
{
     private String config;

     public TransportDatagramSocketImplFactory(String config)
     {
          self.config = config;
     }

     public DatagramSocketImpl createDatagramSocketImpl()
     {
            return TransportDatagramSocketImpl(self.config);
      }
}
```

The transport will also need to provide implementations of SocketImpl and DatagramSocketImpl, as well as InputStream and OutputStream. More details on this are covered in the next section on implementing a transport.

1.1.4 Transport Modules for NIO Networking

Transport modules must provide constructor functions. In the case of NIO, these constructor functions are attached to a SelectionProvider subclass. The initializer for this class takes transport-specific parameters, so there is no fixed interface definition. The subclass below is just an example for what this subclass might look like for a hypothetical transport:

```
// Factory for TCP connections, both client and server
public class TransportSelectionProvider extends SelectionProvider
{
      private String config;

      public TransportSelectionProvider(String config)
      {
            self.config = config;
      }

      public DatagramChannel openDatagramChannel() throws IOException
      {
            return TransportDatagramChannel(self.config);
      }

      public DatagramChannel openDatagramChannel(ProtocolFamily
family)throws IOException
      {
            return TransportDatagramChannel(self.config, family);
      }

      public ServerSocketChannel openServerSocketChannel() throws
IOException
      {
            return TransportServerSocketChannel(self.config);
      }

      public SocketChannel openSocketChannel() throws IOException
```

```
        {
                return TransportSocketChannel(self.config);
        }
}
```

The transport will also need to provide implementations of DatagramChannel, ServerSocketChannel, and SocketChannel. More details on this are covered in the next section on implementing a transport.

1.2. Implementing a Transport

1.2.1 Implementing Class Java Networking

The constructor function gathers the transport configuration information and then passes it to the transport constructor when the createSocketImpl() or createDatagramSocketImpl() methods are called. The rest of the implementation of a transport lies in the creation of a SocketImpl subclasses and a DatagramSocketImpl subclass, as well as subclasses of InputStream and OutputStream.

For implementing SocketImpl, several methods must be implemented: create, connect, bind, listen, accept, getInputStream, getOutputStream, available, close, and sendUrgentData.

The SocketImpl implementation also needs custom implementations of InputStream and OutputStream. For implementing InputStream, the following methods must be implemented: available, close, and read. For implementing OutputStream, the following methods must be implemented: close, flush, and write.

For implementing SocketDatagramImpl, several methods must be implemented: create, bind, send, peek, peekData, receive, setTimeToLive, getTimeToLive, join, leave, joinGroup, leaveGroup, and close.

1.2.2 Implementing NIO Networking

The SelectionProvider constructor function gathers the transport configuration information and then passes it to the transport constructor when the openDatagramChannel, openServerSocketChannel, or openSocketChannel methods are called. The rest of the implementation of a transport lies in the creation of subclasses of DatagramChannel, ServerSocketChannel, and SocketChannel.

For implementing DatagramChannel, several methods must be implemented: open, bind, setOption, socket, isConnected, connect, disconnect, getRemoteAddress, receive, send, read, and write.

For implementing SocketChannel, several methods must be implemented: open, bind, setOption, shutdownInput, shutdownOutput, socket, isConnected, isConnectionPending, connect, finishConnect, getRemoteAddress, read, and write.

For implementing ServerSocketChannel, several methods must be implemented: bind, setOption, socket, and accept.

1.3 Using a Transport

1.3.1 Using a Transport with Classic Java Networking

Using a transport is similar to using the normal networking API in Java. The only difference is that the SocketImplFactory or SocketDatagramImplFactory must be set first.

Here is an example of opening a client TCP connection using the hypothetical example transport defined above:

```
TransportSocketImplFactory transportFactory =
TransportSocketImplFactory("config goes here");

Socket.setSocketImplFactory(transportFactory);

// Now make a normal Java TCP client connection
Socket sock = Socket("127.0.0.1", 8888);
```

The same setup applies for server-side TCP connections.

Here is an example of setting up a UDP client.

```
TransportDatagramSocketImplFactory transportFactory =
TransportDatagramSocketImplFactory("config goes here");

DatagramSocket.setDatagramSocketImplFactory(transportFactory);

// Now make a normal Java UDP socket object
DatagramSocket sock = DatagramSocket();
```

The same setup applies for a UDP server.

1.3.2 Using a Transport with NIO Networking

Using a transport is similar to using the normal NIO networking API in Java. The only difference is that the SelectionProvider must be set to the transport’s subclass of SelectionProvider first.

Java finds the SelectionProvider by first looking at the system property “java.nio.channels.spi.SelectorProvider” for the fully-qualified name of the provider class. Alternatively, if a provider class has been installed in a jar file that is visible to the system class loader, and that jar file contains a provider-configuration file named “java.nio.channels.spi.SelectorProvider” in the resource directory “META-INF/services”, then the first class name specified in that file is used.

No changes to the application source code are necessary to switch an application that uses NIO from using direct networking to using transports. Once this property is set, the transition happens automatically. This applies to TCP client, TCP server, and UDP.

2. References

1. https://docs.oracle.com/javase/10/docs/api/java/net/DatagramSocket.html
2. https://docs.oracle.com/javase/10/docs/api/java/net/Socket.html
3. https://docs.oracle.com/javase/10/docs/api/java/net/SocketImpl.html
4. https://docs.oracle.com/javase/10/docs/api/java/net/DatagramSocketImpl.html
5. https://docs.oracle.com/javase/10/docs/api/java/net/DatagramSocketImplFactory.html
6. https://docs.oracle.com/javase/10/docs/api/java/net/SocketImplFactory.html
7. https://docs.oracle.com/javase/10/docs/api/java/io/InputStream.html
8. 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.