Chord

Programming Guide

Version 2.0

Revision History

Version	Date	Description
2.0		

Table of Contents

1. Overview

Chord allows you to easily develop local information-sharing applications. Devices running Chord-based applications locate each other using UDP-broadcast-based discovery, and then use a TCP/IP-based protocol stack to create a reliable, local, peer-to-peer communications network. Chord-based applications use this network to share data, including text messages, binary messages and files, with selected members of the network. Chord version 2.0 and above also allows you to use UDP-based protocol stack to transfer time-sensitive data, including video, audio or game.

You can use Chord to:

- Discover devices (nodes).
- Join and leave private channels.
- Share information by sending and receiving data and files.

1.1. Basic Knowledge

A node is a device that is connected to other devices through the Chord protocol. The Chord public channel includes all the nodes in the local Chord network, while individual applications on nodes use private channels to interact with each other. A node is always a member of the public channel, and can also be a member of multiple private channels.

Devices running Chord-based applications join the Chord public channel automatically and become nodes. However, the Chord public channel is not "visible" to users. This means that users must use private channels to share data, conversations and files. A private channel only includes nodes that are running the same application.

The following figure shows five devices connected through a Chord public channel. Nodes 2, 3, and 4 are also connected to each other in a private channel (Channel B); while nodes 4 and 5 are connected through another private channel (Channel A). Note that node 4 is connected to both Channel A and Channel B, and that node 1 is not connected to any private channels.

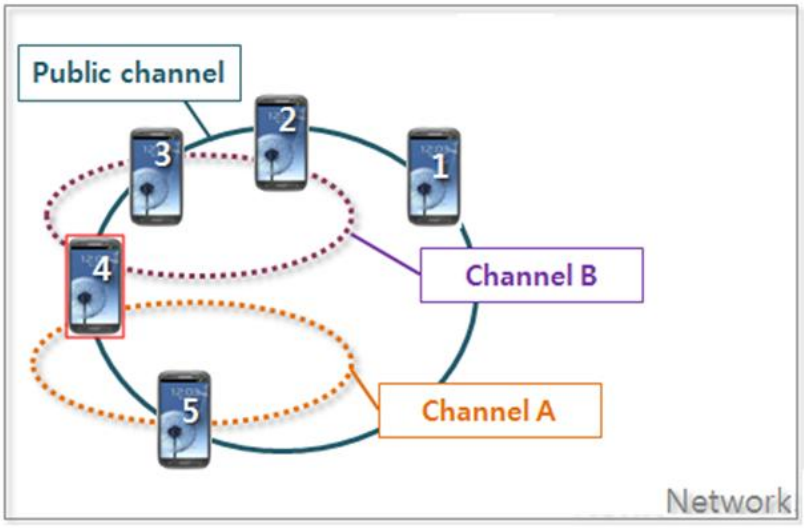

Figure 1: Nodes in channels

1.2. Architecture

The following figure shows the Chord architecture.

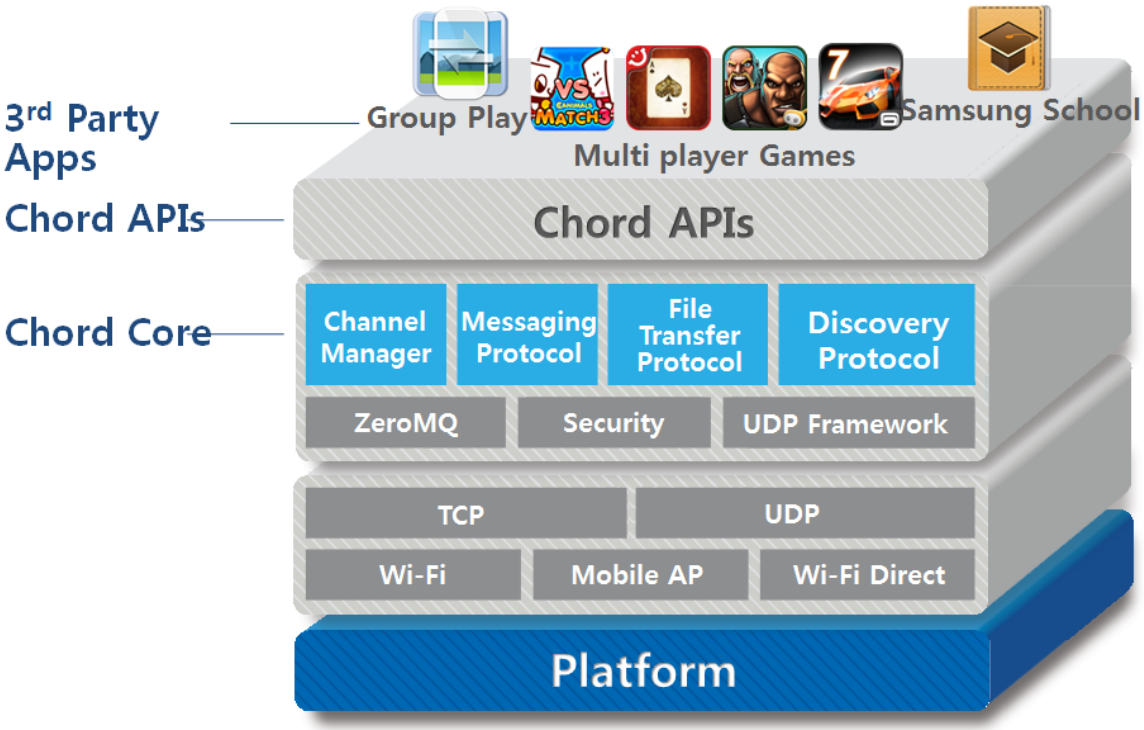

Figure 2: Chord architecture

The architecture consists of:

- **Applications:** One or more applications that use Chord.
- **Channel Manager:** Chord components for managing channels, which are logical groups of nodes.
- **Messaging Protocol:** Chord components for data transfer between nodes.
- **File Transfer Protocol:** Chord components for file transfer between nodes.
- **Discovery Protocol:** Chord components for node discovery.
- **ZeroMQ:** Open source ZeroMQ messaging library, which uses TCP sockets.

1.3. Class Diagram

The following figure shows the Chord classes and interfaces that you can use in your application.

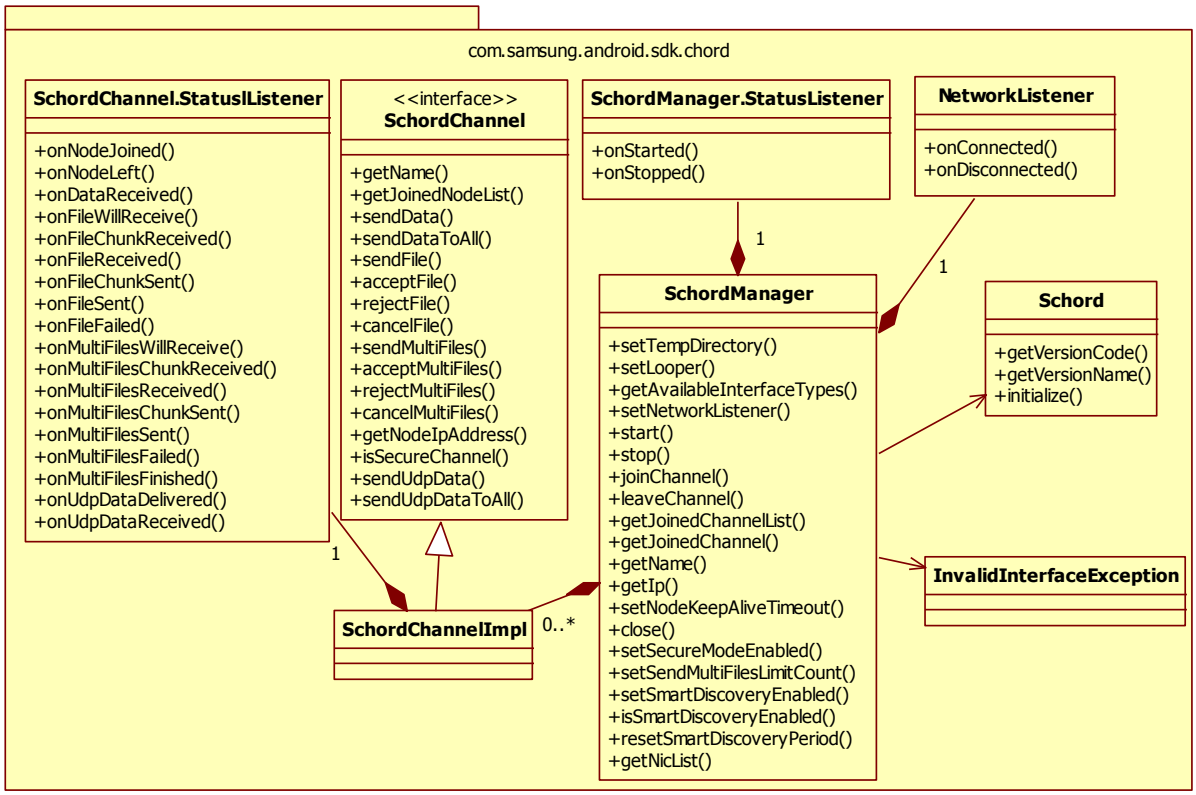

Figure 3: Chord classes and interfaces

The Chord classes and interfaces include:

- **SchordManager:** Creates a node and manages the node's connection to channels.
- **SchordManager.StatusListener:** Listens for the connection status of the node.
- **NetworkListener:** Listens for the status of the network, even if Chord has not started.
- **SchordChannel:** Manages node names and IP addresses, and data and file transfers.

- **SchordChannelImpl:** Provides the implementation of SchordChannel. You can access the channel instance by using SchordChannel.
- **SchordChannel.StatusListener:** Listens for nodes joining and leaving channels, and data and file transfers.
- **InvalidInterfaceException:** Provides the exception thrown when `start()` is called with an invalid interface type.

1.4. Supported Platforms

Android 4.0 (Ice Cream Sandwich API 14) or above support Chord.

1.5. Supported Features

Chord supports the following features:

- Basic methods for communications between devices
- Group management
- Sending data and files
- Sending encrypted data (file data or channel data) over a channel

1.6. Components

- Components
    - chord-v2.0.0.jar
    - libchord-v2.0.so
- Imported packages:
    - com.samsung.android.sdk.chord

1.7. Installing the Package for Eclipse

To install Chord for Eclipse:

1. Add the libchord-v2.0.so and chord-v2.0.0.jar files to the libs folder in Eclipse.

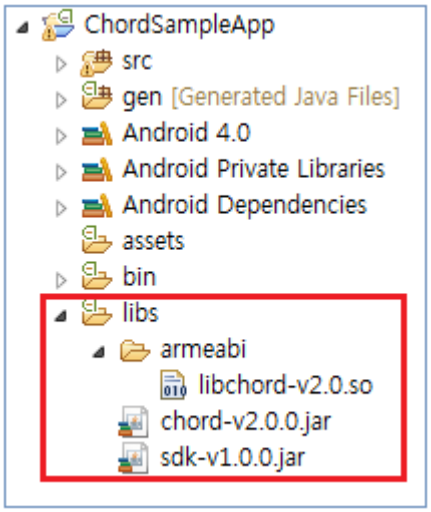

Figure 4: libs folder in Eclipse

2. Add the following permissions to your Android manifest file:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

3. Optionally, add the following permission to your Android manifest file:

```
<uses-permission android:name="android.permission.WAKE_LOCK" />
```

For more details on when to add this permission, see Discovering Chord Peers.

2. Hello Chord

Hello Chord is a simple program that:

1. Joins a channel.
2. Sends the string “Hello Chord!” to another node in the channel.

```
public class HelloChord extends Activity {

    SchordManager mChordManager;

    private static final String CHORD_SAMPLE_MESSAGE_TYPE =
                                "com.samsung.android.sdk.chord.example.MESSAGE_TYPE";
    private static final String CHORD_HELLO_TEST_CHANNEL =
                            "com.samsung.android.sdk.chord.example.HELLOTESTCHANNEL";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.chord_sample_app_activity);

        // Create an instance of Schord.
        Schord chord = new Schord();
        try {
            // Initialize an instance of Schord.
            chord.initialize(this);
        } catch (SsdkUnsupportedException e) {
            // Error handling
        }
        mChordManager = new SchordManager(this);

        List<Integer> interfaceList = mChordManager.getAvailableInterfaceTypes();
        if (interfaceList.isEmpty()) {
            // There is no connection.
            return;
        }

        try {
             mChordManager.start(interfaceList.get(0).intValue(),
                    mManagerListener);
        } catch (InvalidInterfaceException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }


       // Listener for Chord manager events.
       private SchordManager.StatusListener mManagerListener =
                new SchordManager.StatusListener () {

                    @Override
                    public void onStarted(String name, int reason) {
                        if (STARTED_BY_USER == reason) {
                            // Called when Chord is started
```

```
                            // successfully.

                            joinChannel();
                        }
                    }

                    @Override
                    public void onStopped(int reason) {
                        if (STOPPED_BY_USER == reason) {
                            // Called when Chord is stopped.
                        }
                    }
                };
    }

    // Join a desired channel with a given listener.
    private void joinChannel() {

        SchordChannel channel = null;

        try {
            channel = mChordManager.joinChannel(CHORD_HELLO_TEST_CHANNEL,
                    mChannelListener);
        } catch (IllegalArgumentException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // Listener for Chord channel events.
    private SchordChannel.StatusListener mChannelListener = new
SchordChannel.StatusListener () {

        @Override
        public void onNodeJoined(String fromNode, String fromChannel) {

            byte[][] payload = new byte[1][];
            payload[0] = "Hello chord!".getBytes();

            SchordChannel channel = mChordManager.getJoinedChannel(fromChannel);

            // Send simple data.
            channel.sendData(fromNode, CHORD_SAMPLE_MESSAGE_TYPE, payload);
        }

        .....

    };
}
```

3. Using the Schord Class

The Schord class provides the following methods:

- `initialize()` initializes Chord. You need to initialize the Chord package before you can use it. If the device does not support Chord, SsdkUnsupportedException is thrown.
- `getVersionCode()` gets the Chord version number as an integer.
- `getVersionName()` gets the Chord version name as a string.
- `isFeatureEnabled()` checks if a Chord package feature is available on the device.

```
Schord chord = new Schord();
try {
     // Initialize an instance of Schord.
     chord.initialize(this);
} catch (SsdkUnsupportedException e) {
     if(e.getType()==SsdkUnsupportedException.VENDOR_NOT_SUPPORTED) {
         // Vendor is not Samsung
     }
}

int versionCode = chord.getVersionCode();

String versionName = chord.getVersionName();
```

For more information, see `initChord()` in HelloChordFragment.java in ChordSampleApp.

3.1. Using the initialize() method

The `Schord.initialize()` method:

- Initializes the Chord package.
- Checks if the device is a Samsung device.
- Checks if the Samsung device supports the Chord package.
- Checks if the Chord package libraries are installed on the device.

```
void initialize(Context context) throws SsdkUnsupportedException
```

If the Chord package fails to initialize, the `initialize()` method throws an SsdkUnsupportedException exception. To find out the reason for the exception, check the exception message.

3.2. Handling SsdkUnsupportedException

If an SsdkUnsupportedException exception is thrown, check the exception message type using `SsdkUnsupportedException.getType()`.

The following types of exception messages are defined in the Schord class:

- **VENDOR_NOT_SUPPORTED:** The device is not a Samsung device.

3.3. Checking the Availability of Chord Package Features

You can check if a Chord package feature is supported on the device with the `isFeatureEnabled()` method. The feature types are defined in the Schord class. Pass the feature type as a parameter when calling the `isFeatureEnabled()` method. The method returns a Boolean value that indicates the support for the feature on the device.

```
boolean isFeatureEnabled(int type)
```

4. Using the Chord Package

This section describes how to use the Chord package in your application.

4.1. Starting Chord

To start Chord:

1. Create an instance of SchordManager.

   Chord version 2.0 and above allows you to create instances of SchordManager for individual network interfaces. You can use multiple network interfaces at the same time.

2. Call the following methods:
   - `setTempDirectory()` sets a temporary directory for Chord functions.
   - `setLooper()` sets a looper object associated with the thread for processing callbacks.
   - `getAvailableInterfaceTypes()` gets the list of available network interface types.
   - `start()` starts Chord.
3. Once Chord starts, SchordManager calls the following callback method on the application:
   - `onStarted(STARTED_BY_USER)` indicates that Chord has started.

   At this point, Channel Management takes over. When the application is closed, use the following methods in your application:
   - `stop()` stops Chord.
   - `onStopped(STOPPED_BY_USER)` indicates that Chord has stopped.
   - `close()` releases the instance.

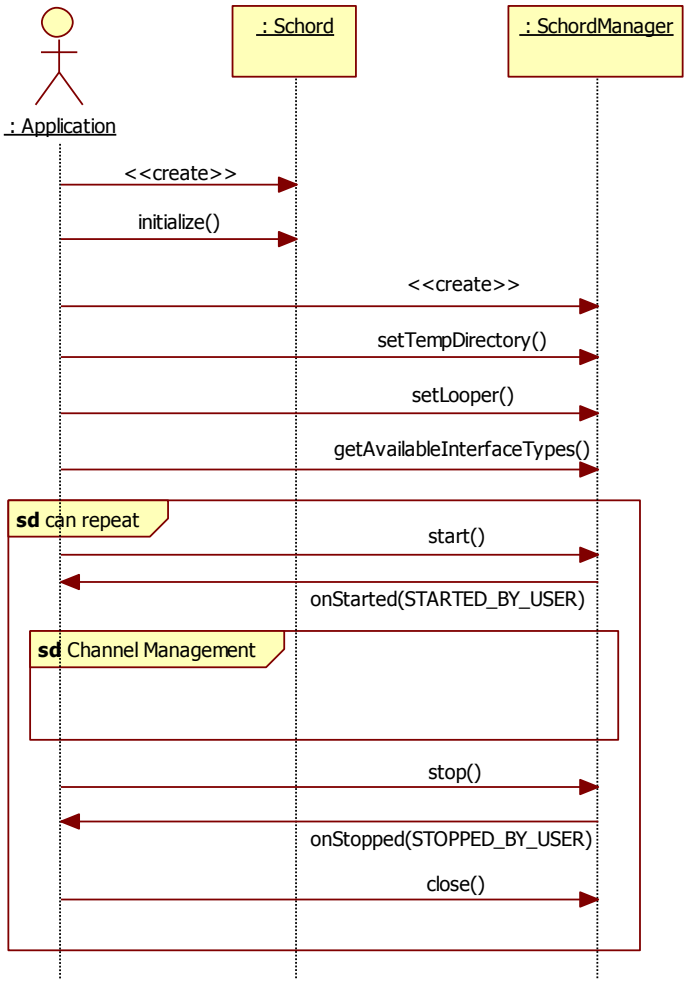

Figure 5: Starting Chord

```
// Initialize Schord
Schord chord = new Schord();
chord.initialize(this);

SchordManager chordManager = new SchordManager(this);
chordManager.setTempDirectory(tempDir);
chordManager.setLooper(getMainLooper());

List<Integer> interfaceList = mChordManager.getAvailableInterfaceTypes();
if (interfaceList.isEmpty()) {
    // There is no connection.
    return;
}

chordManager.start(interfaceList.get(0).intValue(), new
SchordManager.StatusListener() {

    @Override
    public void onStarted(String name, int reason) {
```

```
        if (STARTED_BY_USER == reason) {
            // Started
        }
    }

    @Override
    public void onStopped(int reason) {
        if (STOPPED_BY_USER == reason) {
            // Stopped
        }
    }
});

chordManager.stop();
chordManager.close();
```

For more information, see `initChord()`, `startChord()` and `stopChord()` in HelloChordFragment.java in ChordSampleApp.

4.1.1. Discovering Chord Peers

Each Chord node transmits a UDP broadcast periodically, and parses broadcast messages from other nodes to discover all the nodes on the same subnet.

Devices running Chord-based applications join the public channel automatically.

A node cannot receive a UDP broadcast if it is in LCD-off status. Set the node status to LCD-on to enable the node to discover other nodes while the application is running. To do this, use the normal methods enabled by `android.os.PowerManager.WakeLock`.

4.1.2. Joining and Leaving Private Channels

Once SchordManager is running and the node has been created, call `joinChannel()` in your application, and use the returned SchordChannel to send and receive data on that channel.

To leave the channel, call `leaveChannel()`.

If a Chord node does not receive a UDP signal within a specified amount of time, it considers the missing node to be no longer a part of the network.

The sequence for joining and leaving a private channel is:

1. The application calls `joinChannel()`.
2. The SchordManager instance returns SchordChannel to the application.
3. While the application is active, SchordChannel calls `onNodeJoined()` on the application when a node joins the channel.
4. SchordChannel calls `onNodeLeft()` on the application when a node leaves the channel, or when the UDP signal is not received in the specified amount of time.

5. The application calls `leaveChannel()` to indicate that it is leaving the channel.

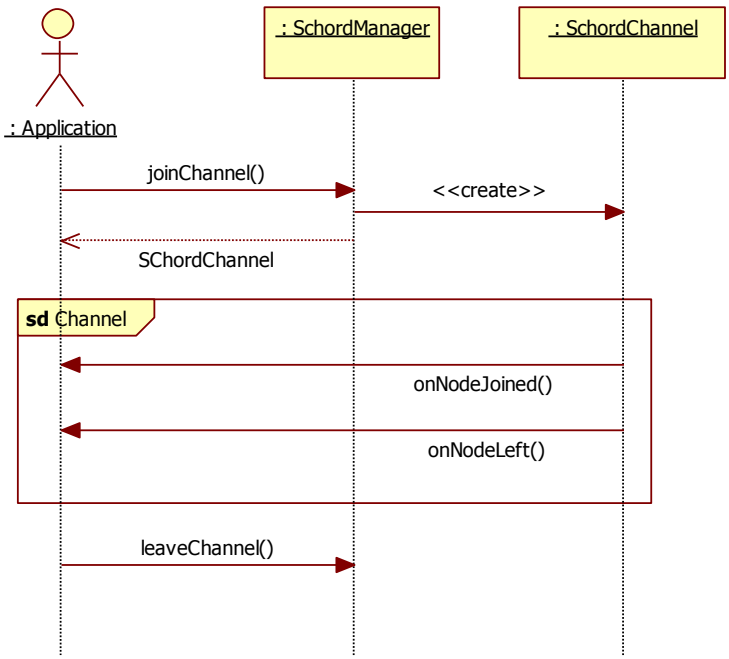

Figure 6: Joining and leaving a private channel

```
mChordManager.joinChannel(CHORD_HELLO_TEST_CHANNEL, new
SchordChannel.StatusListener() {

    @Override
    public void onNodeLeft(String fromNode, String fromChannel) {
        // Called when a node leave event is raised on the
        // channel.
    }

    @Override
    public void onNodeJoined(String fromNode, String fromChannel) {
        // Called when a node join event is raised on the
        // channel.
    }
});
```

For more information, see `joinTestChannel()` in HelloChordFragment.java in ChordSampleApp.

4.2. Sending and Receiving Data and Files

SchordChannel provides the following methods for sending data and files:

- `sendData()` sends data to a specific node on a channel.

- `sendDataToAll()` sends data to all nodes on a channel.
- `sendFile()` sends a file to a specific node on a channel.
- `sendUdpData()` sends data using UDP to a specific node on a channel.
- `sendUdpDataToAll()` sends data using UDP to all nodes on a channel.
- `cancelFile()` cancels a file transfer.
- `acceptFile()` accepts a file transfer.
- `rejectFile()` declines a file transfer.
- `sendMultiFiles()` sends multiple files to a specific node on a channel.
- `cancelMultiFiles()` cancels transfer of multiple files.
- `acceptMultiFiles()` accepts transfer of multiple files.
- `rejectMultiFiles()` declines transfer of multiple files.

4.2.1. Sending and Receiving Data

The sequence for passing data between nodes is as follows:

1. Node A's application calls `sendData()` with a message to indicate the start of data transfer to Node B.
2. Node A's SchordChannel passes the message "Hello B?" to Node B's SchordChannel.
3. Node B's SchordChannel calls `onDataReceived()` and sends the message to Node B's application.
4. Node B's application replies by calling `sendData()` and by sending a message of its own to its SchordChannel.
5. Node B's SchordChannel passes the message "Hello A?" to Node A's SchordChannel.
6. Node A's SchordChannel passes `onDataReceived()` to Node A's application.

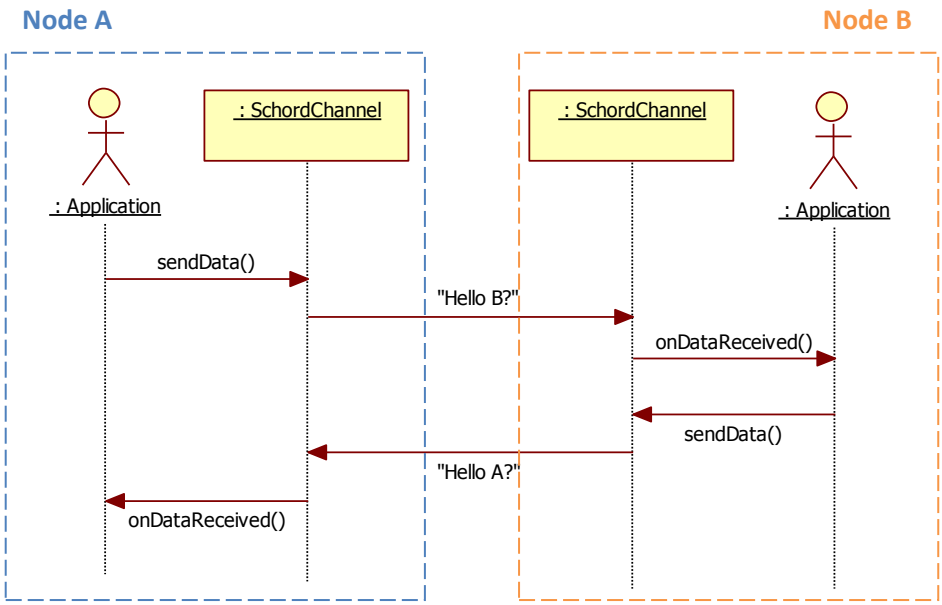

Figure 7: Passing data between nodes

```
mChordManager.joinChannel(CHORD_HELLO_TEST_CHANNEL, new
SchordChannel.StatusListener() {

      @Override
      public void onDataReceived(String fromNode, String fromChannel,
                                       String payloadType, byte[][] payload) {

            // Receive “Hello B?”
            String receivedData = new String(payload[0]);

            // Send “Hello A?”
            byte[][] data = new byte[1][];
            data[0] = "Hello A?".getBytes();

            SchordChannel channel = mChordManager.getJoinedChannel(fromChannel);
            channel.sendData(fromNode, CHORD_SAMPLE_MESSAGE_TYPE, data);
      }
});
```

For more information, see `onNodeJoined()` in HelloChordFragment.java in ChordSampleApp.

4.2.2. Sending and Receiving Data Using UDP

Chord provides the following UDP data transfer features.

- Transfer Mode: Chord supports the following data types.

    - Reliable Data: The data for which reliability has a higher priority than latency. For example, data such as non real time file transfer and signaling data. Chord allows reliable data transfer if the `reliableTime` parameter is set to -1.
    - Non Reliable Data: The data for which latency has a higher priority than reliability. For example, real time data such as audio and video. Chord allows non reliable data transfer if the `reliableTime` parameter is set to 0.
    - Semi Reliable Data: The data for which latency as well as reliability are of equal priority. For example, data such as gaming events. Chord allows semi reliable data transfer if the `reliableTime` parameter is set to any user defined value in ms (Data delivery is guaranteed till the user defined value is reached).
- Response Flag: Chord provides this parameter for receiving notification about the sent data.
    - `respFlag` enabled: Your application receives the `onUdpDataDelivered()` notification when data transfer is successful. You can track the data using the request ID generated by `sendUdpData()`.
    - `respFlag` disabled: Your application does not receive any notification.
- Session Name: Chord allows you to define a session name and use it to send data on multiple sessions.

The sequence for passing data between nodes using UDP (Reliable Mode**,** CHORD_SESSION_CHAT**)** is as follows:

1. Node A's application calls `sendUdpData()` with a message to indicate the start of data transfer to Node B.
2. Node A’s SchordChannel passes the message “Hello B?” to Node B’s SchordChannel.
3. Node A's SchordChannel calls `onUdpDataDelivered()` and sends the notification to Node A's application if `respFlag` is true.
4. Node B’s SchordChannel calls `onUdpDataReceived()` and sends the message to Node B’s application.
5. Node B’s application replies by calling `sendUdpData()` and by sending a message of its own to its SchordChannel.
6. Node B’s SchordChannel passes the message “Hello A?” to Node A’s SchordChannel.
7. Node B's SchordChannel calls `onUdpDataDelivered()` and sends the notification to Node B's application if `respFlag` is true.
8. Node A’s SchordChannel passes `onUdpDataReceived()` to Node A’s application.

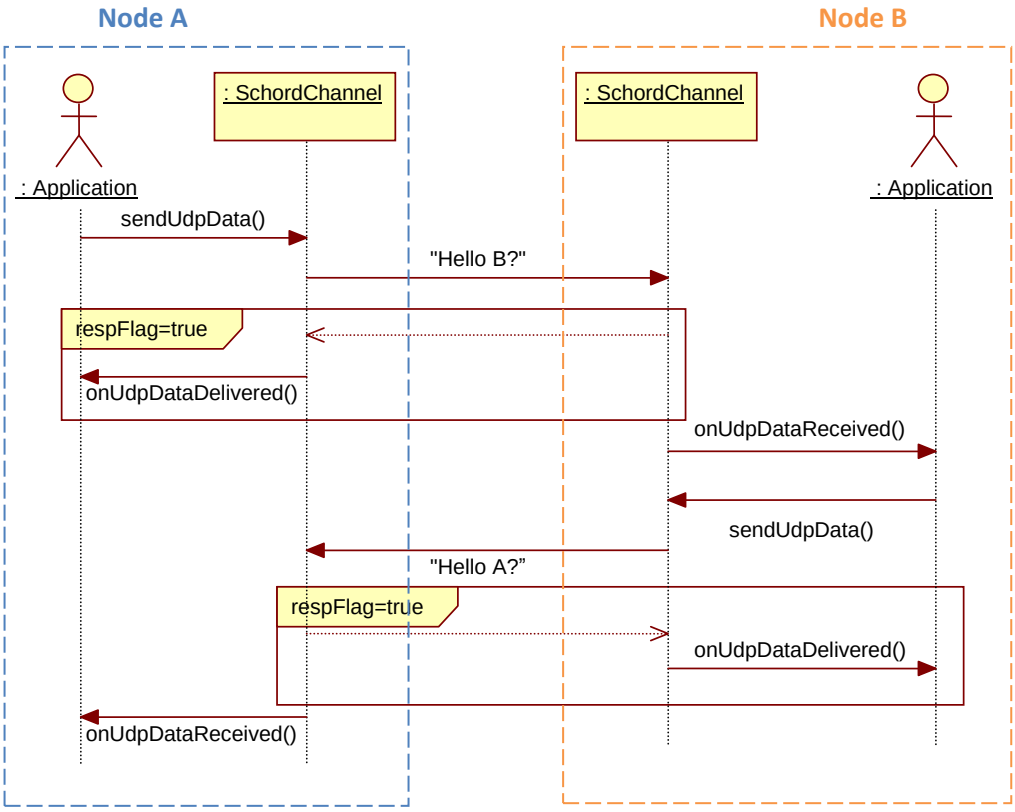

Figure 8: Passing data using UDP between nodes

```
mChordManager.joinChannel(CHORD_HELLO_TEST_CHANNEL, new
SchordChannel.StatusListener() {

      @Override
      public void onUdpDataReceived(String fromNode, String fromChannel,
                                    String payloadType, byte[][] payload,
                                    String sessionName) {

            // Receive “Hello B?”
            String receivedData = new String(payload[0]);

            // Send “Hello A?”
            byte[][] data = new byte[1][];
            data[0] = "Hello A?".getBytes();

            SchordChannel channel = mChordManager.getJoinedChannel(fromChannel);
            channel.sendUdpData(fromNode, -1, false, CHORD_SAMPLE_MESSAGE_TYPE,
                                data, CHORD_SESSION_CHAT);
      }
    @Override
      public void onUdpDataDelivered(String toNode, String toChannel,
                                    String reqId) {

            // Delivered message “Hello A?”
            //Application can delete the reqId data from Queue, if stored
```

```
    }
});
```

For more information, see `UdpFrameworkFragment.java` in ChordSampleApp.

Note:

- Chord TCP and UDP data transfer operations are independent of each other. If the data is sent using both `sendData()` and `sendUdpData()` in some order, the receiver does not necessarily receive the data in the same order as it was sent.
- When communicating with AVD using UDP, no data with the size over 8 KB can be sent due to the relay server limitations.

4.2.3. Sending and Receiving Files

When a node sends files to another node, there is an acknowledgement to verify the successful receipt. Chord breaks down the file into chunks. The sender acknowledges the successful sending of each chunk and the receiver acknowledges the successful receipt of each chunk. On completion, the sender and receiver receive a message indicating the completion of the file transfer.

Node A begins with `sendFile()`. Node B receives `onFileWillReceive()`. The user either accepts (`acceptFile()`) or declines (`rejectFile()`). If Node B accepts the file, the Node B application calls `acceptFile()` and Node A begins sending chunks of the file to Node B.

Accepting a File

As each chunk is sent and verified, Node A receives `onFileChunkSent()` and Node B receives `onFileChunkReceived()`. When the file transfer is complete, Node A receives `onFileSent()` and Node B receives `onFileReceived()`.

The sequence for sending and accepting a file is as follows:

1. Node A's application begins by calling `sendFile()`. This signals the start of file transfer to Node B.
2. SchordChannel calls `onFileWillReceive()` on Node B's application. This is to get user confirmation for receipt of the file.
3. If the user accepts, then Node B's application calls `acceptFile()`.
4. The file transfer begins, and Node A starts sending chunks of data to Node B. When Node A sends a chunk successfully, Node A's application receives `onFileChunkSent()`.
5. Node B's application receives `onFileChunkReceived()` for each chunk of data.
6. When Node B successfully receives the last chunk, SchordChannel calls `onFileSent()` for Node A, and `onFileReceived()` for Node B.

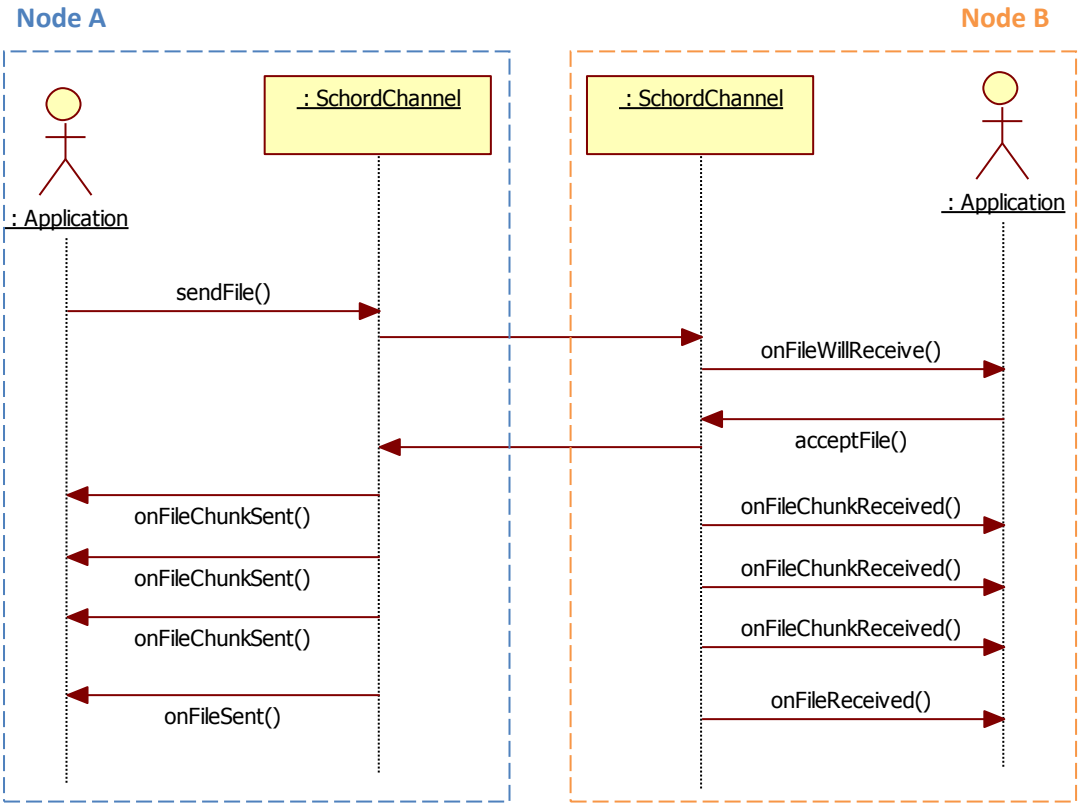

Figure 9: Sending and accepting a file

```
mChordManager.joinChannel(CHORD_HELLO_TEST_CHANNEL, new
SchordChannel.StatusListener() {

        @Override
        public void onFileWillReceive(String fromNode,
                String fromChannel, String fileName, String hash,
                String fileType, String exchangeId, long fileSize) {
            // Accept file transfer
            SchordChannel channel = mChordManager
                    .getJoinedChannel(CHORD_SEND_TEST_CHANNEL);
            channel.acceptFile(trId, 30 * 1000, 2, 300 * 1024);
        }

        @Override
        public void onFileChunkReceived(String fromNode, String fromChannel,
               String fileName, String hash, String fileType, String exchangeId,
               long fileSize, long offset) {
            // Called when an individual chunk of the file is received
        }

        @Override
        public void onFileReceived(String fromNode, String fromChannel, String
               fileName, String hash, String fileType, String exchangeId, long
               fileSize, String tmpFilePath) {
            // Called when the file transfer is completed
```

```
        }
});
```

For more information, see `onActivityResult()` and `SchordManager.StatusListener()` in SendFilesFragment.java in ChordSampleApp.

Rejecting a File

If Node B's user declines the file from Node A, then Node B's application calls `rejectFile()`. Node A's application receives `onFileFailed(ERROR_FILE_REJECTED)` and the file is not sent.

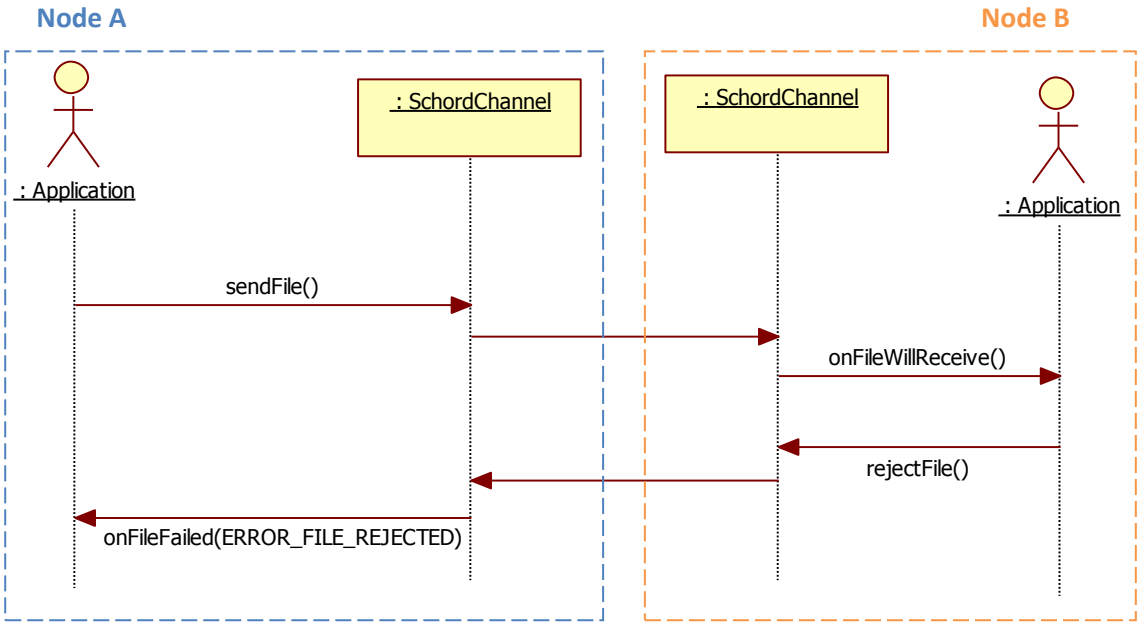

Figure 10: Rejecting a file before the transfer begins

```
mChordManager.joinChannel(CHORD_HELLO_TEST_CHANNEL, new
SchordChannel.StatusListener() {

        @Override
        public void onFileWillReceive(String fromNode, String fromChannel,
                String fileName, String hash, String fileType, String exchangeId,
                long fileSize) {

            // Reject file transfer
            SchordChannel channel = mChordManager
                    .getJoinedChannel(CHORD_SEND_TEST_CHANNEL);
            channel.rejectFile(exchangeId);
        }
});
```

For more information, see `displayFileNotify()` in SendFilesFragment.java in ChordSampleApp.

Receiver Cancelling a File

If the receiver cancels the file transfer, Node B's application calls `cancelFile()`. Both applications then receive an `onFileFailed(ERROR_FILE_CANCELLED)` message, and the file transfer stops.

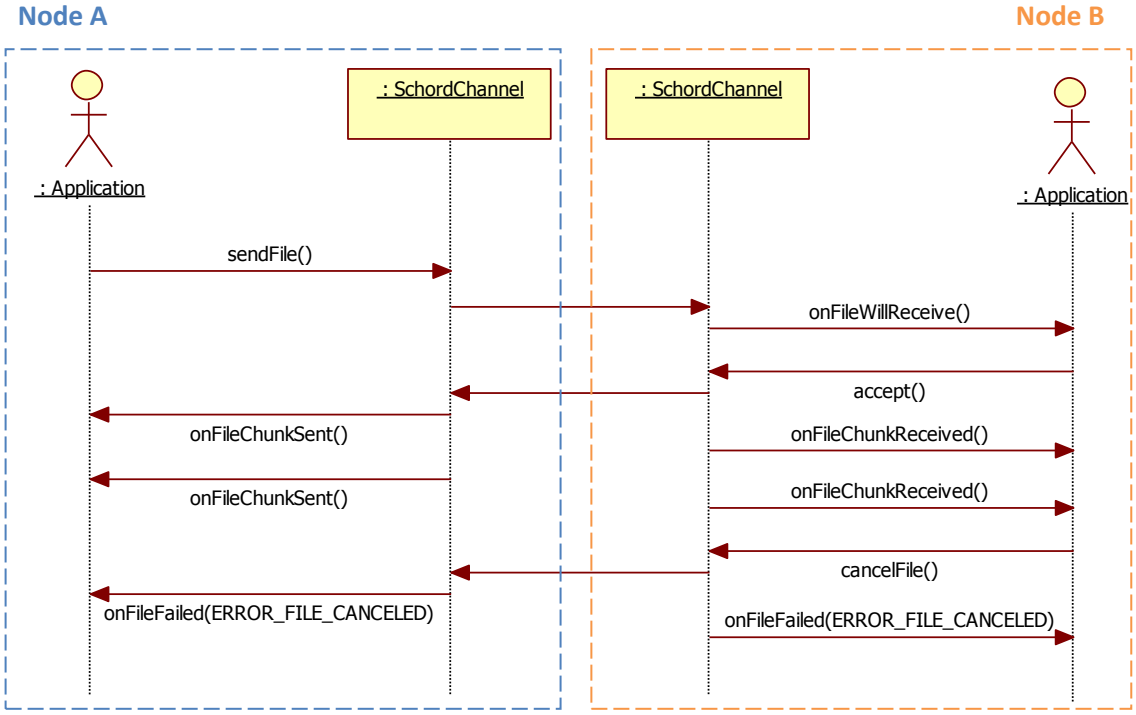

Figure 11: Receiver cancels a file transfer that is in progress

For more information, see `onFileCanceled()` and `onFileFailed()` in SendFilesFragment.java in ChordSampleApp.

Sender Cancelling a File

If the sender cancels the file transfer, Node A's application calls `cancelFile()`, and both applications receive an `onFileFailed(ERROR_FILE_CANCELLED)` message.

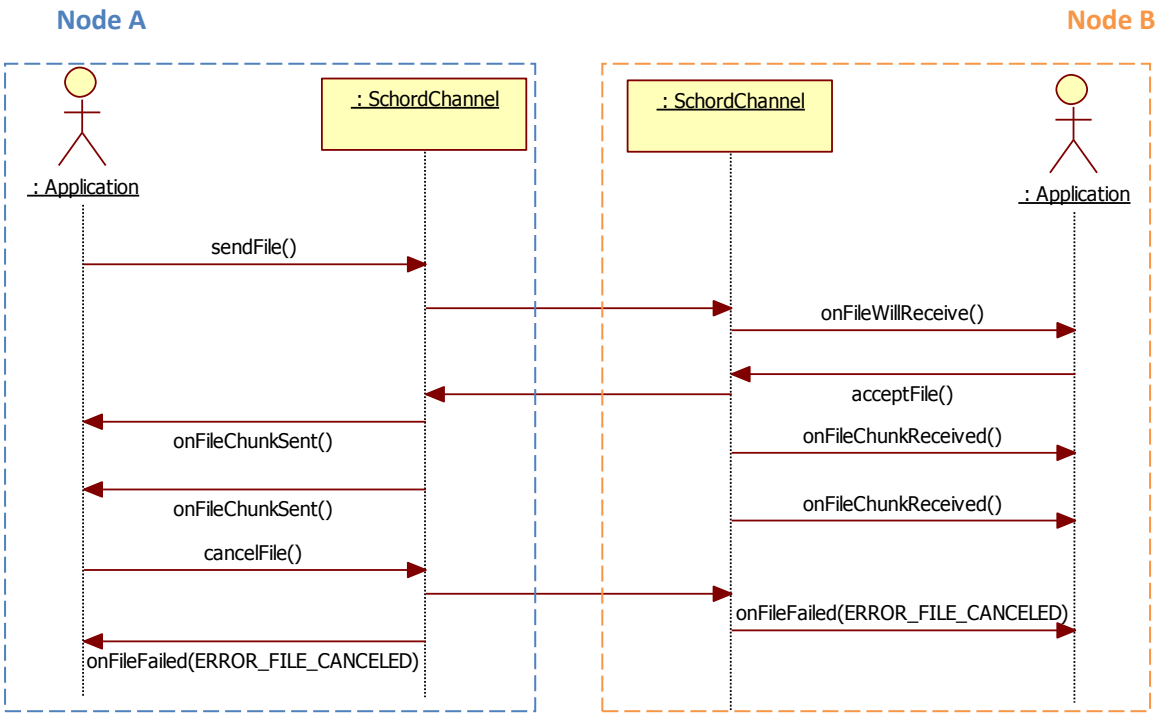

Figure 12: Sender cancels a file transfer that is in progress

For more information, see `onFileCanceled()` and `onFileFailed()` in SendFilesFragment.java in ChordSampleApp.

Sending Multiple Files

Chord provides additional methods for supporting batch transfer of multiple files. The method for multiple file transfer is similar in operation to `sendFile()`. This means that Chord allows the initiation of a batch file transfer followed by acceptance/rejection of the entire batch by the receiver. Similarly, file transfer for the entire batch is stopped upon failure of any one file transmission.

Accepting Files

The following figure shows the sequence for accepting multiple file transmissions.

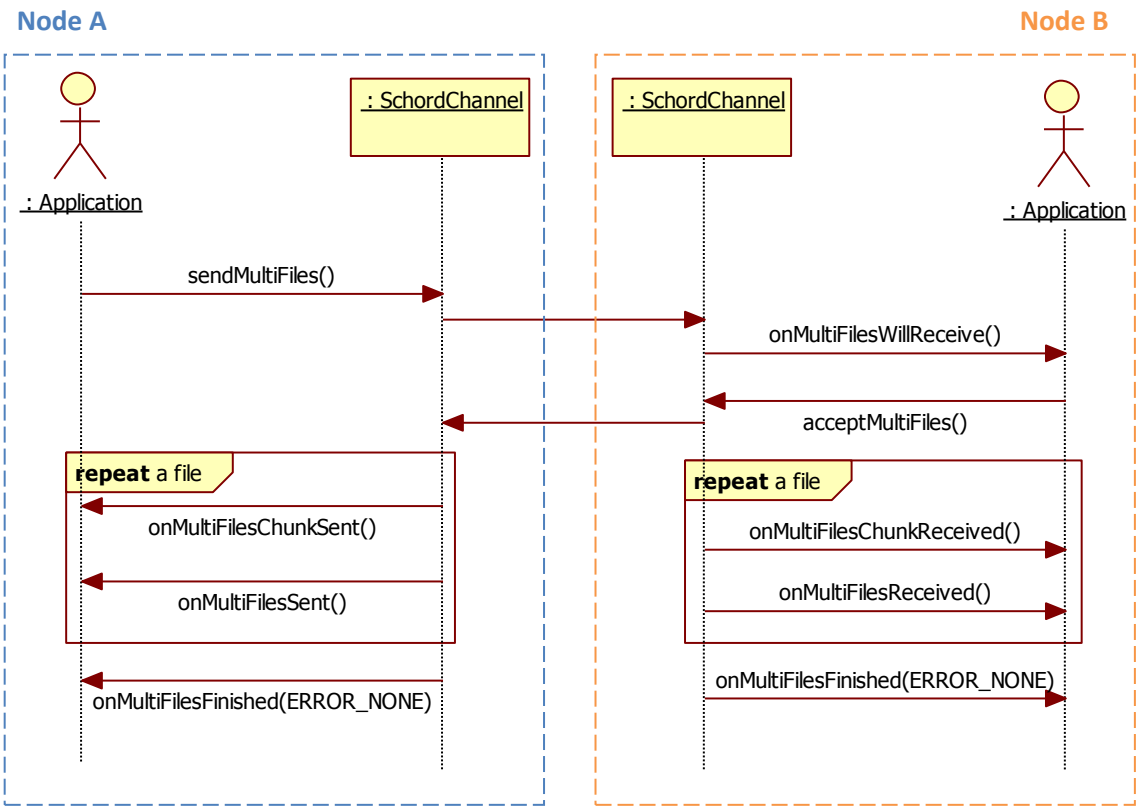

Figure 13: Sending and accepting multiple files

SchordChannel calls `onMultiFilesSent()` / `onMultiFilesChunkSent()` and `onMultiFilesReceived()` / `onMultiFilesChunkReceived()` repeatedly during the file transmission.

```
mChordManager.joinChannel(CHORD_HELLO_TEST_CHANNEL, new
SchordChannel.StatuslListener() {

        @Override
        public void onMultiFilesWillReceive(String fromNode, String fromChannel,
                String fileName, String taskId, int totalCount, String fileType,
                long fileSize) {

            SchordChannel channel = mChordManager
                    .getJoinedChannel(CHORD_SEND_TEST_CHANNEL);
            // Accept multiple file transfer
            channel.acceptMultiFiles(trId, 30 * 1000, 2, 300 * 1024);
        }

        @Override
        public void onMultiFilesChunkReceived(String fromNode, String fromChannel,
                String fileName, String taskId, int index, String fileType,
                long fileSize, long offset) {
            // Called when an individual chunk of the file is received
        }

        @Override
        public void onMultiFilesReceived(String fromNode, String fromChannel,
                String fileName, String taskId, int index, String fileType,
                long fileSize, String tmpFilePath) {
            // Called when the file transfer is completed
        }
```

```
        @Override
        public void onMultiFilesFinished(String node, String channel, String taskId,
                int reason) {

            if (ERROR_NONE == reason) {
                // Finished transfer of multiple files
            }
        }
});
```

For more information, see `onActivityResult()` and `SchordChannel.StatusListener()` in SendFilesFragment.java in ChordSampleApp.

Additionally, an application can use `setSendMultiFilesLimitCount()` to restrict the number of files to be transmitted concurrently. The default value is 1, which means the files are transmitted sequentially.

Rejecting Files

If Node B's user declines the files from Node A, then Node B's application calls `rejectMultiFiles()`. Node A's application receives `onMultiFilesFinished(ERROR_FILE_REJECTED)` and the files are not sent.

The following figure shows the sequence for rejecting multiple file transmission.

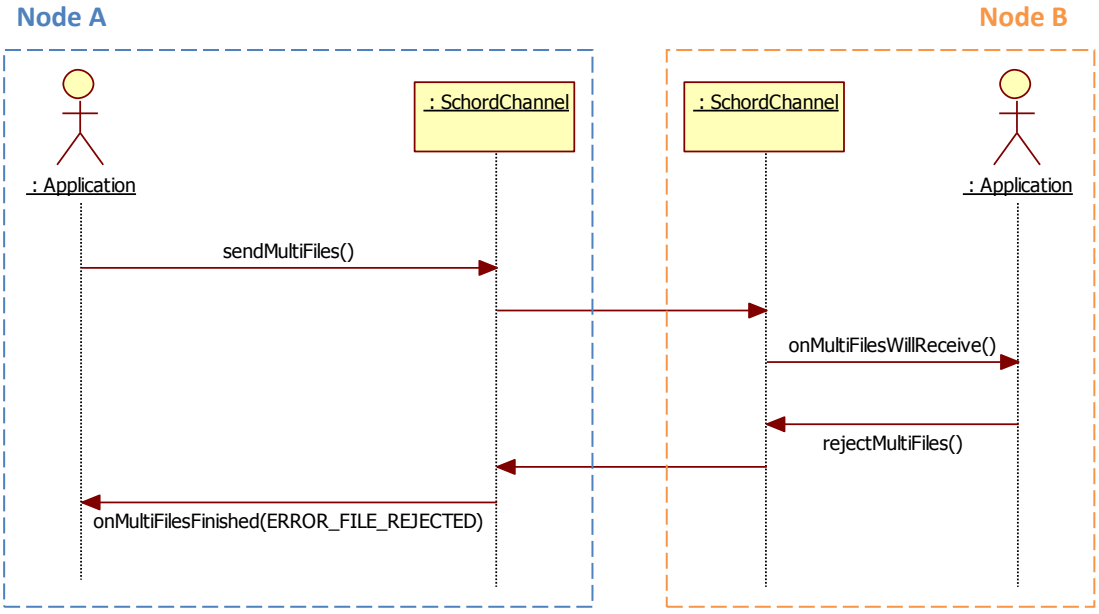

Figure 14: Rejecting multiple files

```
mChordManager.joinChannel(CHORD_HELLO_TEST_CHANNEL, new
SchordChannel.StatusListener() {

        @Override
        public void onMultiFilesWillReceive(String fromNode, String fromChannel,
                String fileName, String taskId, int totalCount, String fileType,
                long fileSize) {

            SchordChannel channel =
```

```
                            mChordManager.getJoinedChannel(CHORD_SEND_TEST_CHANNEL);
            channel.rejectMultiFiles(taskId);
        }
});
```

For more information, see `displayFileNotify()` in SendFilesFragment.java in ChordSampleApp.

Receiver Cancelling Transfer of Multiple Files

If the receiver cancels the transfer of multiple files, Node B's application calls `cancelMultiFiles()`.

If the file transfer is already in progress, the applications receive the `onMultiFilesFailed(ERROR_FILE_CANCELED)` message for every file in the transmission and the `onMultiFilesFinished(ERROR_FILE_CANCELED)` message at the end. You cannot cancel individual file transfers. You can only cancel the entire transmission.

The following figure shows the sequence for cancelling multiple file transmission at the receiver's request.

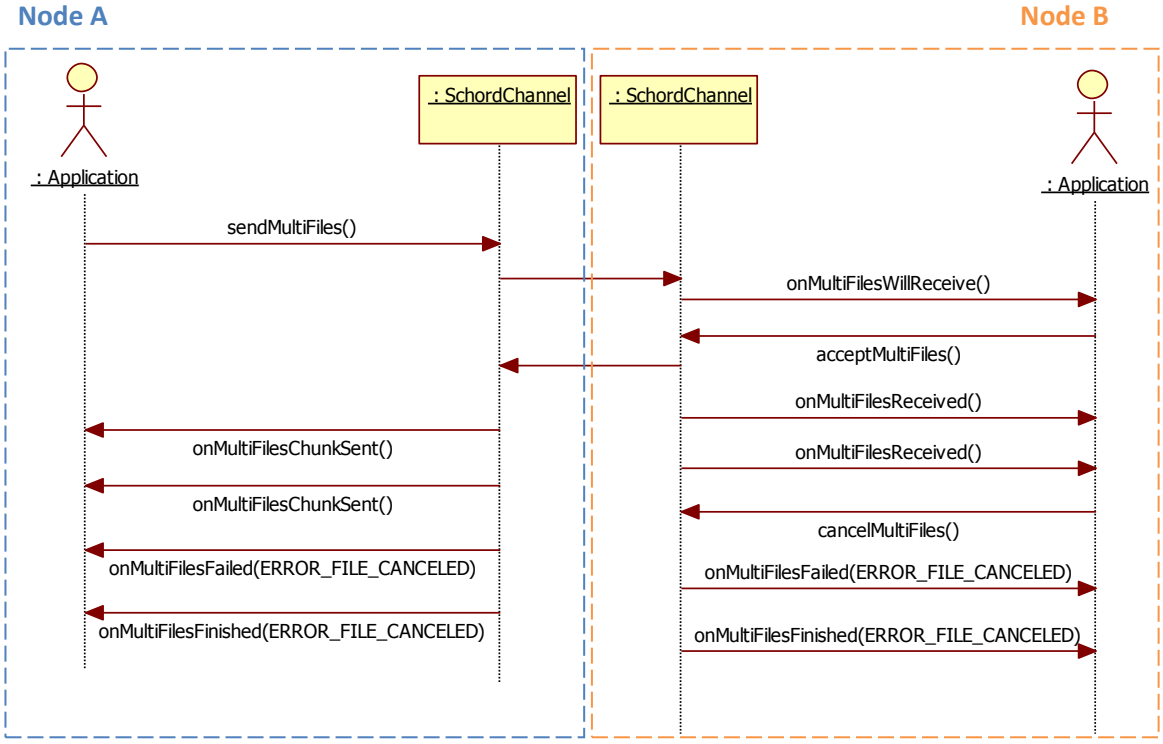

Figure 15: Receiver cancels multiple files transfer that is in progress

For more information, see `onFileCanceled()` and `SchordChannel.StatusListener()` in SendFilesFragment.java in ChordSampleApp.

The sender can also cancel sending the file in the middle of transmissions.

4.2.4. Communicating over a Secure Channel

Secure channels allow applications to send encrypted data, which creates a more secure environment for devices to communicate with one another.

To communicate over a secure channel:

1. Enable the secure mode by calling `setSecureModeEnabled(true)` prior to starting Chord. Secure mode is disabled by default. When you call `setSecureModeEnabled(true)`, an encryption key is exchanged before data exchange.
2. To join a secure channel, generate the channel name according to the specific naming syntax. To apply security on a channel, add `SECURE_PREFIX` as a prefix to the channel name. For example, to apply security to `com.samsung.android.chord.example.CHANNEL_TEST`, call `joinChannel()` with the following channel name:

   `SECURE_PREFIX +com.samsung.android.chord.example.CHANNEL_TEST`

The following figure shows the sequence for joining and leaving a secure channel.

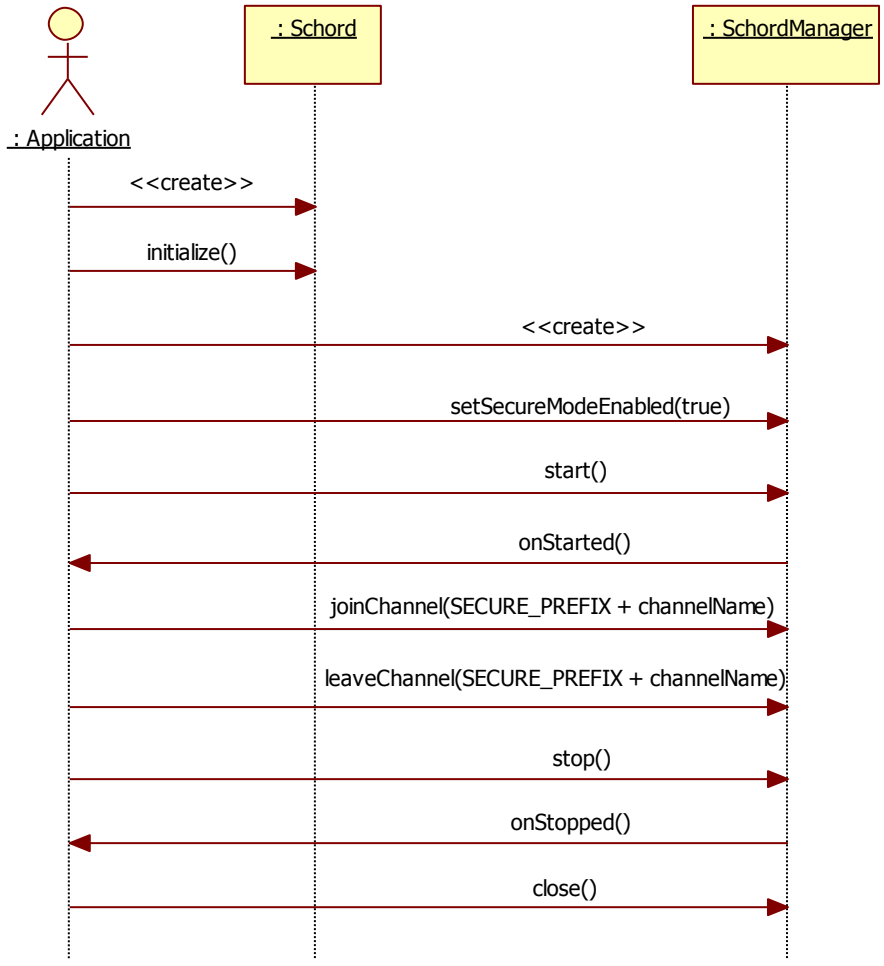

Figure 16: Joining and leaving a secure channel

```
mChordManager.setSecureModeEnabled(true);
mChordManager.start(SchordManager.INTERFACE_TYPE_WIFI, new
SchordManager.StatusListener() {

        @Override
        public void onStarted(String nodeName, int reason) {
```

```
            if (STARTED_BY_USER == reason) {
                // Chord is started successfully
                mChordManager.joinChannel(SchordManager.SECURE_PREFIX
                        + CHORD_SECURE_TEST_CHANNEL, mChannelListener);
            } else if (STARTED_BY_RECONNECTION == reason) {
                // Re-start by network re-connection
            }
        }
});
```

In the security mode, security is ensured simply by encrypting the sent data using AES/RSA. For further security, such as channel authentication, you must add a security algorithm on the application layer.

4.3. Using Smart Discovery

Chord supports the Smart Discovery feature for dynamically adapting to the device environment.

The discovery time period is adapted by Chord. The period is reset to the default value by certain triggers, such as a new node joining the network or when the device LCD is switched on. You can also reinitialize this period explicitly by calling `resetSmartDiscoveryPeriod()`. If you do not want to use this functionality, you can switch it off by calling `setSmartDiscoveryEnabled(false)`.

You can use `setNodeKeepAliveTimeout()` to specify the duration a node waits for communication from another node before marking it as disappeared. This value also impacts the discovery period. When the time increases, the adaptation algorithm works effectively. However, detection of exceptional node terminations can be delayed.

To efficiently use Smart Discovery, it is recommended that you set `setNodeKeepAliveTimeout()` to be larger than 34 seconds, which is the default timeout.

4.4. Managing Network Disconnection and Recovery

Chord creates a very reliable communications network. However, if the network connection is lost, SchordManager invokes `onStopped(NETWORK_DISCONNECTED)`. Depending on the application, Chord can either stop, or it can wait for the network to become available again.

When a node stops receiving UDP signals from another node, it waits for a preset amount of time. If no signal is received within that time period, Chord removes the "missing" node from its private and public channels. The default time that Chord waits is 34 seconds, but you can change it by calling `setNodeKeepAliveTimeout()`.

If Chord is still waiting and the network becomes available again, SchordManager invokes `onStarted(STARTED_BY_RECONNECTION)`, where "STARTED BY RECONNECTION" is the reason code. `SchordChannel.StatusListener` then resumes, and the node is on the same channels as before, as if the disconnection had not happened, and without a new `joinChannel()` call.

While disconnected, `SchordChannel.StatusListener` is suspended and returns an error when invoked. Therefore, use a timer to check for reconnection.

When Chord starts, the timer is set to a default of 34 seconds. You can also set the duration of the timeout.

The sequence for reconnecting to a network is as follows:

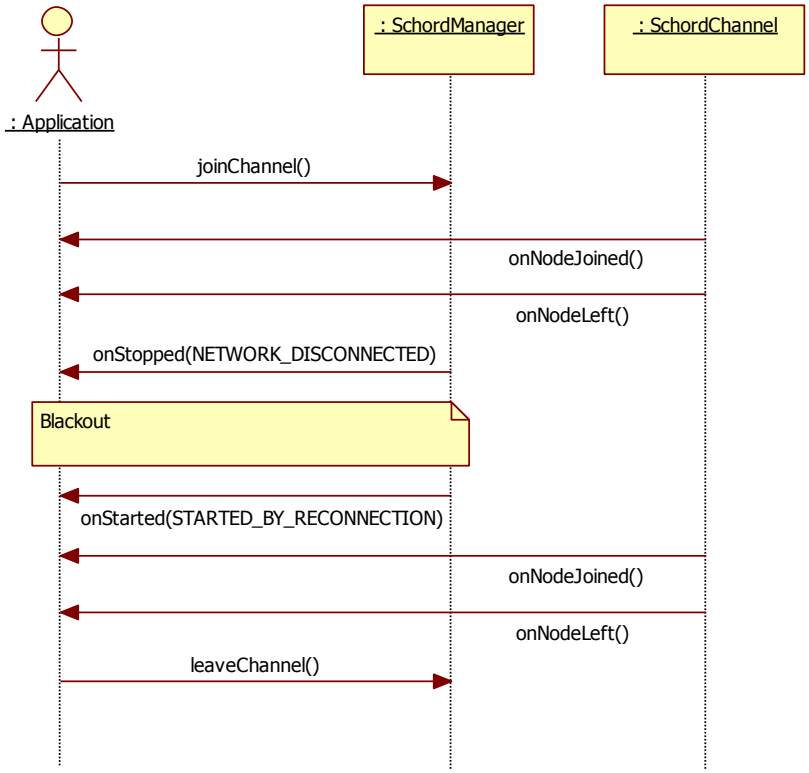

Figure 17: Reconnecting to a network after disconnection

```
mChordManager.start(SchordManager.INTERFACE_TYPE_WIFI, new
SchordManager.StatusListener() {

        @Override
        public void onStarted(String nodeName, int reason) {
            if (STARTED_BY_USER == reason) {
                // Chord is started successfully
            } else if (STARTED_BY_RECONNECTION == reason) {
                // Re-start by network re-connection
            }
        }

        @Override
        public void onStopped(int reason) {
            if (STOPPED_BY_USER == reason) {
                // Chord is stopped
            } else if (NETWORK_DISCONNECTED == reason) {
                // Stopped by network disconnection
            }
        }
});
```

For more information, see `SchordManager.StatusListener()` in HelloChordFragment.java in ChordSampleApp.

4.5. Listening for Network Changes

Chord provides the NetworkListener utility to listen for network changes even before Chord starts.

The following code demonstrates how to use NetworkListener.

```
mChordManager.setLooper(getMainLooper());
mChordManager.setNetworkListener(new NetworkListener() {

        @Override
        public void onConnected(int interfaceType) {
            // Update your UI
        }

        @Override
        public void onDisconnected(int interfaceType) {
            // Update your UI
        }
});
```

For more information, see `initChord()` in HelloChordFragment.java in ChordSampleApp.

Copyright

Though every care has been taken to ensure the accuracy of this document, Samsung Electronics Co., Ltd. cannot accept responsibility for any errors or omissions or for any loss occurred to any person, whether legal or natural, from acting, or refraining from action, as a result of the information contained herein. Information in this document is subject to change at any time without obligation to notify any person of such changes.

Samsung Electronics Co. Ltd. may have patents or patent pending applications, trademarks copyrights or other intellectual property rights covering subject matter in this document. The furnishing of this document does not give the recipient or reader any license to these patents, trademarks copyrights or other intellectual property rights.

No part of this document may be communicated, distributed, reproduced or transmitted in any form or by any means, electronic or mechanical or otherwise, for any purpose, without the prior written permission of Samsung Electronics Co. Ltd.

The document is subject to revision without further notice.

All brand names and product names mentioned in this document are trademarks or registered trademarks of their respective owners.

For more information, please visit http://developer.samsung.com/