Video Calling Architecture: Detailed Call Flow

1. Video Calling System Architecture Overview

This document provides a detailed explanation of the video calling implementation in the Flutter application, with particular focus on:

* The complete information exchange between caller and receiver
* The detailed notification process flow
* How calls are accepted and declined
* Room creation and lifecycle management
* WebRTC connection establishment and state management

2. Call Information Exchange

When a call is initiated, several key pieces of information are exchanged between the caller and receiver to establish the connection:

Information	Purpose	Where Stored/Sent
Room ID	Unique identifier for the WebRTC session	Firestore document ID + Notification payload
Caller ID	Identifies who is calling	Stored in notification and room data
Caller Name/Email	Shows caller identity to receiver	Included in notification payload
SDP Offer	WebRTC connection parameters from	Stored in Firestore room document
SDP Answer	WebRTC response parameters from	Stored in Firestore room document
ICE Candidates	Network path information for connection	Stored in room subcollections
Call Status	Tracks state of call	Stored in room document
Timestamps	Records timing of events	Stored with status changes

This information forms the core of the signaling process, allowing the two devices to establish a direct peer-to-peer connection through WebRTC.

Video Calling Architecture: Detailed Call Flow

3. Detailed Call Flow: Room Creation & Notification

3.1 Creating a Room in Firestore

When a user initiates a call, a new 'room' document is created in Firestore. This document serves as the signaling channel between the caller and receiver:

Dart code:

```
Future<String> createRoom(RTCVideoRenderer remoteRenderer) async {
  // Create a new document in Firestore with auto-generated ID
  FirebaseFirestore db = FirebaseFirestore.instance;
  DocumentReference roomRef = db.collection('rooms').doc();
  hasNotifiedReceiver = false;

  // Create WebRTC peer connection with specified configuration
  peerConnection = await createPeerConnection(configuration);

  // Add local media tracks to peer connection
  localStream?.getTracks().forEach((track) {
    peerConnection?.addTrack(track, localStream!);
  });

  // Set up ICE candidate collection
  var callerCandidatesCollection = roomRef.collection('callerCandidates');
  peerConnection?.onIceCandidate = (RTCIceCandidate candidate) {
    callerCandidatesCollection.add(candidate.toMap());
  };

  // Create an offer and store it in Firestore
  RTCSessionDescription offer = await peerConnection!.createOffer();
  await peerConnection!.setLocalDescription(offer);

  Map<String, dynamic> roomWithOffer = {
    'offer': offer.toMap(),
    'created_at': FieldValue.serverTimestamp(),
    'caller_id': FirebaseAuth.instance.currentUser?.uid,
    'caller_name': FirebaseAuth.instance.currentUser?.email ?? 'Unknown',
    'status': 'pending'
  };

  await roomRef.set(roomWithOffer);
  roomId = roomRef.id;

  return roomId!; // Return the room ID for notification
}
```

3.2 Sending Call Notification

Video Calling Architecture: Detailed Call Flow

Once the room is created, the application sends a notification to the receiver with all necessary information to join the call:

Dart code:

```
Future<void> _sendCallNotification(String roomId) async {
  final currentUser = _auth.currentUser;
  if (currentUser != null && !signaling.hasNotifiedReceiver) {
    await _notificationService.sendCallNotification(
      receiverUserId: widget.receiverUserId,  // Who should receive the call
      callerName: currentUser.email ?? 'Unknown',  // Who's calling
      callerId: currentUser.uid,  // Caller's unique ID
      roomId: roomId,  // Where the WebRTC session info is stored
    );
    signaling.hasNotifiedReceiver = true;
  }
}
```

3.3 Notification Service Implementation

The notification service sends a push notification to the receiver with critical call data in the payload:

Dart code:

```
// Inside NotificationService
Future<void> sendCallNotification({
  required String receiverUserId,
  required String callerName,
  required String callerId,
  required String roomId
}) async {
  try {
    // Store notification data in Firestore (triggers cloud function)
    await FirebaseFirestore.instance.collection('notifications').add({
      'type': 'call',
      'receiverId': receiverUserId,
      'callerId': callerId,
      'callerName': callerName,
      'roomId': roomId,
      'timestamp': FieldValue.serverTimestamp(),
      'status': 'pending'
    });

    // The cloud function sends FCM notification with data payload:
    // {
    //   "notification": {
    //     "title": "Incoming Call",
    //     "body": "Call from $callerName"
    //   },
    //   "data": {
```

```
    //     "type": "call",
    //     "roomId": "$roomId",
    //     "callerId": "$callerId",
    //     "callerName": "$callerName"
    //   },
    //   "android": {
    //     "priority": "high",
    //     "notification": {
    //       "channel_id": "calls"
    //     }
    //   },
    //   "apns": {
    //     "headers": {
    //       "apns-priority": "10"
    //     },
    //     "payload": {
    //       "aps": {
    //         "category": "INCOMING_CALL"
    //       }
    //     }
    //   }
    // }
  } catch (e) {
    print('Error sending call notification: $e');
  }
}

// Method to cancel call notifications
Future<void> cancelCallNotification(String roomId) async {
  try {
    await FirebaseFirestore.instance.collection('notifications')
      .where('roomId', isEqualTo: roomId)
      .where('status', isEqualTo: 'pending')
      .get()
      .then((snapshot) {
        for (var doc in snapshot.docs) {
          doc.reference.update({'status': 'cancelled'});
        }
      });

    // The cloud function would then cancel any active notification
  } catch (e) {
    print('Error cancelling notification: $e');
  }
}
```

Video Calling Architecture: Detailed Call Flow

4. Receiving & Processing Incoming Calls

4.1 Notification Handling

When the receiver's device gets the push notification, the system processes it and presents an incoming call UI:

Dart code:

```
// In the app's notification handling logic (typically in main.dart or a
service)
void handleIncomingCallNotification(Map<String, dynamic> data) {
  final String roomId = data['roomId'];
  final String callerId = data['callerId'];
  final String callerName = data['callerName'];

  // Display UI for accepting/declining the call
  showCallNotification(roomId, callerId, callerName);
}

void showCallNotification(String roomId, String callerId, String callerName) {
  // On Android, this might use a full-screen intent notification
  // On iOS, this might use CallKit

  // Create UI with accept/decline buttons
  showDialog(
    context: context,
    barrierDismissible: false,
    builder: (context) => IncomingCallDialog(
      callerName: callerName,
      roomId: roomId,
      callerId: callerId,
      onAccept: () => acceptCall(roomId, callerId, callerName),
      onDecline: () => declineCall(roomId),
    ),
  );
}
```

4.2 Accepting a Call

When the user accepts the call, the app navigates to the CallPage with the roomId and other necessary information:

Dart code:

```
void acceptCall(String roomId, String callerId, String callerName) {
  // Close the incoming call dialog
  Navigator.pop(context);
```

```
  // Navigate to the call page with incoming=true flag
  Navigator.push(
    context,
    MaterialPageRoute(
      builder: (context) => CallPage(
        receiverUserId: callerId,
        receiverUserEmail: callerName,
        isIncoming: true,
        roomId: roomId,
      ),
    ),
  );
}
```

4.3 Joining the Call Room

When the CallPage initializes for an incoming call, it joins the existing room:

Dart code:

```
Future<void> _initializeCall() async {
  await _localRenderer.initialize();
  await _remoteRenderer.initialize();

  // Set up callbacks
  signaling.onAddRemoteStream = ((stream) {
    _remoteRenderer.srcObject = stream;
    // Handle remote stream...
  });

  // Open media devices
  await signaling.openUserMedia(_localRenderer, _remoteRenderer);

  if (widget.isIncoming && widget.roomId != null) {
    // Join the room if this is an incoming call
    await signaling.joinRoom(widget.roomId!, _remoteRenderer);
  } else if (!widget.isIncoming) {
    // Create a room for outgoing call (covered earlier)
  }

  setState(() {
    _isCallStarted = true;
  });
}
```

4.4 Detailed Room Joining Process

The joinRoom method connects to the existing Firestore room and creates the WebRTC answer:

Video Calling Architecture: Detailed Call Flow

Dart code:

```
Future<void> joinRoom(String roomId, RTCVideoRenderer remoteVideo) async {
  FirebaseFirestore db = FirebaseFirestore.instance;
  DocumentReference roomRef = db.collection('rooms').doc(roomId);

  // Listen for room state changes
  var roomSubscription = roomRef.snapshots().listen((snapshot) async {
    if (!snapshot.exists) {
      print("Room no longer exists");
      onCallEnded?.call();
      return;
    }

    Map<String, dynamic> data = snapshot.data() as Map<String, dynamic>;

    // Check if room was marked as ended
    if (data['ended'] == true) {
      print("Call ended by other party");
      onCallEnded?.call();
      return;
    }
  });

  // Cancel notification
  NotificationService().cancelCallNotification(roomId);
  hasNotifiedReceiver = true;

  // Create peer connection
  peerConnection = await createPeerConnection(configuration);

  // Add local media tracks
  if (localStream != null) {
    localStream!.getTracks().forEach((track) {
      peerConnection?.addTrack(track, localStream!);
    });
  }

  // Handle ICE candidates
  var calleeCandidatesCollection = roomRef.collection('calleeCandidates');
  peerConnection?.onIceCandidate = (RTCIceCandidate candidate) {
    calleeCandidatesCollection.add(candidate.toMap());
  };

  // Handle remote tracks
  peerConnection?.onTrack = (RTCTrackEvent event) {
    remoteVideo.srcObject = event.streams[0];
    remoteStream = event.streams[0];
    onAddRemoteStream?.call(event.streams[0]);
  };
```

```
  // Get the offer from Firestore
  var roomSnapshot = await roomRef.get();
  var data = roomSnapshot.data() as Map<String, dynamic>;
  var offer = data['offer'];

  // Create answer
  await peerConnection?.setRemoteDescription(
    RTCSessionDescription(offer['sdp'], offer['type']),
  );
  var answer = await peerConnection!.createAnswer();

  // Set local description and update room
  await peerConnection!.setLocalDescription(answer);
  Map<String, dynamic> roomWithAnswer = {
    'answer': {'type': answer.type, 'sdp': answer.sdp},
    'status': 'active',
    'connected_at': FieldValue.serverTimestamp()
  };
  await roomRef.update(roomWithAnswer);

  // Listen for caller's ICE candidates
  roomRef.collection('callerCandidates').snapshots().listen((snapshot) {
    for (var change in snapshot.docChanges) {
      if (change.type == DocumentChangeType.added) {
        var data = change.doc.data() as Map<String, dynamic>;
        peerConnection!.addCandidate(
          RTCIceCandidate(
            data['candidate'],
            data['sdpMid'],
            data['sdpMLineIndex'],
          ),
        );
      }
    }
  });
}
```

Video Calling Architecture: Detailed Call Flow

5. Declining Calls & Call Management

5.1 Declining an Incoming Call

When a user declines an incoming call, the app updates the room status and cancels notifications:

Dart code:

```
void declineCall(String roomId) {
  // Close incoming call dialog
  Navigator.pop(context);

  // Cancel the notification
  NotificationService().cancelCallNotification(roomId);

  // Update Firestore to indicate call was declined
  FirebaseFirestore.instance.collection('rooms').doc(roomId).update({
    'status': 'declined',
    'declined_at': FieldValue.serverTimestamp()
  });
}
```

5.2 Detecting Declined Calls (Caller Side)

The caller detects when a call is declined through their Firestore room listener:

Dart code:

```
// Inside createRoom method or separate listener
roomRef.snapshots().listen((snapshot) async {
  if (!snapshot.exists) return;

  Map<String, dynamic> data = snapshot.data() as Map<String, dynamic>;

  // Check if call was declined
  if (data['status'] == 'declined') {
    // Notify user the call was declined
    if (onCallDeclined != null) {
      onCallDeclined!();
    }

    // Show UI notification
    if (mounted) {
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(
          content: Text('Call was declined'),
          duration: Duration(seconds: 3),
        ),
      );
```

```
    }

    // Clean up resources
    hangUp(localRenderer);
  }
});
```

5.3 Missed Call Handling

If the receiver doesn't respond, the system handles it as a missed call:

Dart code:

```
// In the calling code, implement a timeout
Future<void> _startCallTimeout() async {
  // Set a timer for the call to be considered missed
  callTimer = Timer(const Duration(seconds: 45), () {
    if (mounted && !_isCallConnected) {
      // Call wasn't answered in time
      _firestore.collection('rooms').doc(roomId).update({
        'status': 'missed',
        'missed_at': FieldValue.serverTimestamp()
      });

      // Show missed call UI
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(
          content: Text('Call wasn't answered'),
          duration: Duration(seconds: 3),
        ),
      );

      // End the call attempt
      _endCall();
    }
  });
}
```

5.4 Active Call Management

During an active call, both parties can control various aspects:

Dart code:

```
// Toggle microphone mute state
void _toggleMute() {
  final localStream = _localRenderer.srcObject;
  if (localStream != null) {
    final audioTracks = localStream.getAudioTracks();
    if (audioTracks.isNotEmpty) {
```

```
      for (var track in audioTracks) {
        track.enabled = !track.enabled;
      }
      setState(() {
        _isMuted = !_isMuted;
      });
    }
  }
}

// Toggle camera on/off
void _toggleCameraEnabled() {
  final localStream = _localRenderer.srcObject;
  if (localStream != null) {
    final videoTracks = localStream.getVideoTracks();
    if (videoTracks.isNotEmpty) {
      for (var track in videoTracks) {
        track.enabled = !track.enabled;
      }
      setState(() {
        _isCameraEnabled = !_isCameraEnabled;
      });
    }
  }
}

// Switch between front and rear cameras
Future<void> _toggleCamera() async {
  final localStream = _localRenderer.srcObject;
  if (localStream != null) {
    final videoTracks = localStream.getVideoTracks();
    if (videoTracks.isNotEmpty) {
      await Helper.switchCamera(videoTracks[0]);
      setState(() {
        _isUsingFrontCamera = !_isUsingFrontCamera;
      });
    }
  }
}
```

Video Calling Architecture: Detailed Call Flow

6. Ending Calls & Room Lifecycle

6.1 Ending an Active Call

Either party can end the call, which updates the Firestore document and cleans up resources:

Dart code:

```
Future<void> _endCall({bool isRemoteEnded = false}) async {
  if (!mounted || _isEnding) return;
  _isEnding = true;

  try {
    // Cancel the call notification if we have a room ID
    if (widget.roomId != null) {
      await _notificationService.cancelCallNotification(widget.roomId!);
    }

    // Clear video renderers first
    if (mounted) {
      setState(() {
        _hasRemoteVideo = false;
        _isCallStarted = false;
      });
    }

    // Cleanup call in signaling service
    await signaling.hangUp(_localRenderer);

    if (mounted) {
      if (isRemoteEnded) {
        ScaffoldMessenger.of(context).showSnackBar(
          const SnackBar(
            content: Text('Call ended by other party'),
            duration: Duration(seconds: 3),
          ),
        );
      }

      // Navigate back
      Future.microtask(() {
        if (mounted && Navigator.canPop(context)) {
          Navigator.pop(context);
        }
      });
    }
  } catch (e) {
    print('Error ending call: $e');
    // Still try to navigate back on error
```

```
    if (mounted) {
      Future.microtask(() {
        if (mounted && Navigator.canPop(context)) {
          Navigator.pop(context);
        }
      });
    }
  }
}
```

6.2 Detailed Hangup Process

The hangUp method in the Signaling service handles the WebRTC connection teardown and room cleanup:

Dart code:

```
Future<void> hangUp(RTCVideoRenderer localVideo) async {
  // Stop all tracks in the local stream
  final tracks = localVideo.srcObject?.getTracks();
  if (tracks != null) {
    for (var track in tracks) {
      track.stop();
    }
  }

  // Close the peer connection
  if (peerConnection != null) {
    peerConnection!.close();
    peerConnection = null;
  }

  if (roomId != null) {
    // Mark the room as ended in Firestore
    var db = FirebaseFirestore.instance;
    var roomRef = db.collection('rooms').doc(roomId);

    // Update the room document to mark it as ended
    await roomRef.update({
      'ended': true,
      'ended_at': FieldValue.serverTimestamp(),
      'ended_by': FirebaseAuth.instance.currentUser?.uid,
    });

    // Cancel any active notifications for this room
    await NotificationService().cancelCallNotification(roomId!);

    // Delete the room after a short delay to ensure all clients receive the
'ended' update
    await Future.delayed(const Duration(seconds: 2));
```

```
    // Delete ICE candidate subcollections first
    var callerCandidates = await roomRef.collection('callerCandidates').get();
    for (var doc in callerCandidates.docs) {
      await doc.reference.delete();
    }

    var calleeCandidates = await roomRef.collection('calleeCandidates').get();
    for (var doc in calleeCandidates.docs) {
      await doc.reference.delete();
    }

    // Finally delete the room document
    await roomRef.delete();
  }

  // Clean up resources
  localVideo.srcObject = null;
  localStream?.dispose();
  localStream = null;
  roomId = null;
  hasNotifiedReceiver = false;
}
```

6.3 Detecting Remote Call Ended

When the other party ends the call, the app detects this through the Firestore listener:

Dart code:

```
// Inside room listeners
roomRef.snapshots().listen((snapshot) async {
  if (!snapshot.exists) {
    print("Room no longer exists");
    onCallEnded?.call();
    return;
  }

  Map<String, dynamic> data = snapshot.data() as Map<String, dynamic>;

  // Check if room was marked as ended
  if (data['ended'] == true) {
    print("Call ended by other party");

    if (mounted) {
      // Show notification to user
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(
          content: Text('Call ended by other party'),
          duration: Duration(seconds: 3),
        ),
```

```
      );
    }

    onCallEnded?.call();
    return;
  }
});
```

6.4 Complete Room Lifecycle

The Firestore room document follows this complete lifecycle:

1. Created: Initial state with 'pending' status
   * Contains caller information and SDP offer
   * Created timestamp

2. Ringing: When notification is sent to receiver
   * Status updated to 'ringing'
   * Notification timestamp added

3. Accepted/Active: When receiver accepts call
   * Status updated to 'active'
   * Contains both offer and answer
   * Connected timestamp added

4. Declined: If receiver explicitly declines
   * Status updated to 'declined'
   * Declined timestamp added

5. Missed: If receiver doesn't respond in time
   * Status updated to 'missed'
   * Missed timestamp added

6. Ended: When either party ends active call
   * Status updated to 'ended'
   * Contains information about who ended
   * Ended timestamp added

7. Deleted: After call completion
   * Room and ICE candidate collections removed
   * Typically happens 2-3 seconds after ending

Video Calling Architecture: Detailed Call Flow

7. Call Information Flow Summary

The complete call flow involves exchange of the following information in sequence:

Call Flow Sequence:

1. CALLER creates room with unique roomId
2. CALLER generates SDP offer and stores in room
3. CALLER sends notification with roomId, callerId, callerName
4. RECEIVER gets notification with call information
5. RECEIVER accepts call and navigates to CallPage with roomId
6. RECEIVER joins room and retrieves SDP offer
7. RECEIVER creates SDP answer and updates room
8. CALLER receives SDP answer through Firestore listener
9. BOTH PARTIES exchange ICE candidates through subcollections
10. WebRTC connection established for media streaming
11. EITHER PARTY can end call, marking room as 'ended'
12. OTHER PARTY detects 'ended' status and cleans up
13. Room is deleted after short delay

This comprehensive signaling system ensures reliable call establishment while keeping the actual media streams peer-to-peer for optimal performance and minimal server load.

The key features of this architecture include:

* Reliable notification delivery through Firebase Cloud Messaging
* Clean and systematic room state management in Firestore
* Efficient WebRTC connection negotiation
* Proper handling of all call states (pending, ringing, active, declined, missed, ended)
* Resource cleanup to prevent memory leaks and unnecessary server costs
* Comprehensive error handling and fallback mechanisms

This implementation provides a robust foundation for video calling features while minimizing server costs, as the actual media streams travel directly between users after the connection is established.