This document describes the protocols used in Contract Whist Scorecard. The approach used is to have an abstraction layer which provides a consistent(ish) interface to the underlying application. Below this, specific protocols are implemented. To date the following protocols are supported

* Apple Multi-peer Connectivity Nearby protocol (MCNearby)
* Rabbit MQ using the RMQClient cocoa pod
* Loopback – this is still embryonic to allow a single player to play against computer players

This can be represented as shown below

Application

Abstraction Layer

Multi-Peer Connectivity

Interface Layer

Rabbit MQ Connectivity

Interface Layer

Apple Multi-Peer Connectivity Class

RMQClient

Cocoa Pod

# Abstraction Layer

The abstraction layer has a quite similar interface to the Apple MCNearby interface. To some extent it is trying to make other protocols look and feel like the MCNearby interface. However, this protocol relies on the devices being close to each other, allowing connection to be initiated based on nearby devices. In other cases, this will not be applicable; for example, when devices will connect across the internet. At the moment 3 kinds of communication protocol are contemplated

* Online protocols where hosts invite specific clients to join (e.g. Rabbit MQ)
* Nearby protocols where hosts advertise, and clients browse for nearby hosts (e.g. Multi-peer connectivity)
* Loopback protocols where the host and clients are connected programmatically

## Instantiating a communication handler

The abstraction layer provides a class (CommsHandler) which allows the application to instantiate communication handlers. This class has two static methods which allow the application to create either a client handler or a server handler. These will return a communications handler class conforming to the CommsClientHandlerDelegate or CommsServerHandlerDelegate protocols respectively.

These method calls have identical parameters:

**proximity:** CommsConnectionProximityThe proximity of the communication required. Allowed values are nearby, online and loopback.

**mode:** CommsConnectionMode The connection mode required. Allowed values are broadcast, invite and loopback.

**serviceId:** StringThe service identifier string for the connection. For broadcast mode (currently Multi-peer Connectivity) this will be the connection type which is advertised. For online mode, this might be the service URI. Currently it is not used in Rabbit MQ implementations as the service URI is derived within the communication handler. However, this should not be assumed in. the application which should provide an ID in case a future communications protocol requires it.

**deviceName:** StringThe device name of the device initiating the request.

From the proximity and the mode, the abstraction layer will deduce which type of communication protocol is appropriate and will create an instance of it and return it. The underlying application should not need to be aware of the actual communication method chosen but should simply invoke the methods in the protocol that the communications handler conforms to.

## The CommsHandlerDelegate protocol

Both the CommsServerHandlerDelegate and the CommsClientHandlerDelegate conform to this protocol and hence the properties and methods defined by this protocol are available to the application for both server and client handlers.

## The CommsServerHandlerDelegate protocol

This protocol defines how the server should be instantiated (this will have been done automatically in the CommsHandler.Server method) and its properties and methods. Methods are provided by this protocol and the combined client/server protocol above to allow the application to request the handler to:

* Start / stop advertising for / inviting connections
* Send data to an existing connection
* Disconnect
* Reset a connection

## The CommsClientHandlerDelegate protocol

This protocol defines how the client should be instantiated (this will have been done automatically in the CommsHandler.Client method) and its properties and methods. Methods are provided by this protocol and the combined client/server protocol above to allow the application to request the handler to:

* Start / stop browsing for advertised servers or checking for invitations
* Initiate a connection to a server
* Send data to an existing connection
* Disconnect a connection
* Reset all connections
* Check online invitations (online/invite connections only)

## Handler → Application Delegates

Several delegates are provided to allow an application to receive updates from the communication handler.

### Client and Server delegates

**CommsStateDelegate** Allows clients and servers to detect when the state of a connection changes. The state of the connection is available peer associated with the connection (CommsPeer see below). It is defined by the CommsConnectionState enumeration and has values notConnected, connecting, connected, reconnecting and recovering.

**CommsDataDelegate** Allows clients and servers to receive data from a connection.

### Server only delegates

**CommsConnectionDelegate** Allows a server to be notified on an incoming connection request

**CommsServerHandlerStateDelegate** Allows a server to detect changes to the state of the communication handler. The state is defined in the enumeration CommsServerHandlerState and has values of notStarted, advertising, inviting, invited and reconnecting.

### Client only delegates

**CommsBrowserDelegate** Allows clients to detect when advertising/inviting servers start or stop advertising

**CommsBroadcastDelegate** Allows clients to receive a broadcast from a queue. Only applicable on queue mode connections.

## Connection Identifiers (Peers)

A class (CommsPeer) is used to pass information about a connection (or a client or a server before a connection is connected). These are known as peers largely to keep in line with the terminology of the MCNearby framework which has a similar concept of MCPeer. They contain the following properties:

**deviceName:** String Remote device name

**playerEmail:** String? Remote player email

**playerName:** String? Remote player name

**state:** CommsConnectionState Current state of connection. Possible values are notConnected, connecting, connected, reconnecting, recovering

**reason:**  String? Reason for last disconnect if state notConnected

**autoReconnect:** Bool (read-only) Whether connection will auto-reconnect

**mode:** CommsConnectionMode (read-only) Mode of the parent handler

**proximity:** CommsConnectionProximity (read-only) Proximity of the parent handler

**type:** CommsConnectionType (read-only) Type of the parent handler

# The Multi-peer Connectivity Communication Handler

This is an interface to the Apple Multi-peer connectivity framework (MCNearby) which is well documented. The handler tries to hide some of the more esoteric elements of the framework from the application.

A server can start advertising on a service ID. Connections are then received from remote clients. The connection request contains the remote player email / name. A single handler can maintain connections (sessions) with multiple remote clients. However, it is assumed that only one connection will exist with any specific remote device.

When a connection is made, it can (and usually will) be made with an option to reconnect if the connection drops. The only way reconnection will not be attempted is if the server sends a disconnect request with a reason other than “Reset”. In this case the client will send a disconnect and not try to reconnect.

# The RabbitMQ Communication Handler

This is much more complicated than the multi-peer connectivity handler as the underlying framework (AMQP client) is not a connection-based protocol. It is simply a message queue protocol. Connections / sessions are built on top of this in the communication handler. To complicate things further the provider used to host the AMQP service (CloudAMQP) limit the number of queues available on the free service to 100 (it was originally 20), and the number of messages per month to 1,000,000. Therefore, an effort has been made to limit the number of queues (and to a lesser extent the number of message) where possible.

Also, there is no concept of advertising a service in RabbitMQ. Therefore, this has been implemented using the iCloud database and notifications.

## The Server Invitation Process

When a Server Handler is started, it is supplied with the email of the local player and a list of the emails of the players to invite. At this point it does the following:

1. Creates a queue with a unique name (generally just a UUID). This queue will be shared by all remote players. Each message will contain a list of player emails. If a device receives a message on a queue for an email other than itself, it simply ignores it. If a server is starting as part of a recovery process (where the server device has crashed), it will receive the name of the queue to use from the application and it will simply re-uses it.
2. Notifies the application that a connection has been received from each of the remote players. This is not really true, but it allows the queue UUID to be made available to the application and the connections are still flagged as notConnected.
3. The handler state is set to ‘inviting’ (or ‘reconnecting’ if it is recovering).
4. Deletes any previous invitations from this player in the iCloud database
5. Sends an invitation to each remote player in turn as follows:
   1. Creates an invite for the player containing the name, email and device of the player inviting them, the name of the queue, and an expiry date/time for the invite.
   2. If the remote player has accepted notifications from the Whist app, then they will automatically receive a notification that an invite for them has been created. On a simulator (where notifications are not supported), if it has been configured then the player will be notified using a separate queue with a UUID of “notification”.
6. The server sends a reset broadcast to the queue with a mode of “start” (or “recover” if it is recovering).
7. The handler state is set to ‘invited’ (or ‘notStarted’ if any of the above fails).

If the server process subsequently stops, then it will delete the invitations from the iCloud database.

## The Client Invitation Receipt Process

If notifications from the Whist app are enabled the client device will receive a notification (even if the Whist app is not running). If the user clicks the notification, they will be taken to the Play Game option in the app. Alternatively the user can just go into the Play Game option manually.

This option will create an online Client Handler and a nearby Client Handler. It will start both handlers.

Starting the online handler will cause it to check the online invitations (in the iCloud database). If it finds any that have not expired, they will be displayed and if the user has entered Play Game by clicking on a notification, it will automatically select this invitation. Note that the expiry date/time is not checked in recovery mode. This involves:

1. Disconnecting any pre-existing connections to this device by sending a disconnect message to the queue.
2. Creating a session UUID for the session. This will be used to filter all subsequent messages to/from the server.
3. Sending a connect request message to the queue for the remote server device. This contains the session UUID, the local player name and the local player email.
4. When the server receives the request, it checks that the email was in its most recent invitation list. If so, it sends back a connection response with a status of success. Otherwise it sends a failure response. It also updates the application via the state delegate that the connection is now in a connected state.

Data is now passed in messages up and down the queue (marked with the session UUIDs they are targeted at). Additionally, either end can send a reset message which asks the other end to disconnect and reconnect and for the server to re-send the current state.

## Queue Connections

The online queue client handler also supports queue connections. These just open a queue (with a name provided by the application). The application can then send and receive messages to/from the queue. No filtering is carried out. This is primarily used for functionality such as remote logging rather than inter-device connection.