
CORONA® AUTOLAN DOCUMENTATION V 1.2

TABLE OF CONTENTS

Introduction	1
Step 1: Opening the Game.....	3
Step 2: Designing the game scene	4
Step 3: AutoLAN Specifics.....	5
Step 4: Making a connection.....	6
Step 5: Our Game Networking Protocol.....	8
Step 6: The Game Loop	9
Other Tutorials.....	11
The Internet.....	12
AutoLAN Specifics.....	13

INTRODUCTION

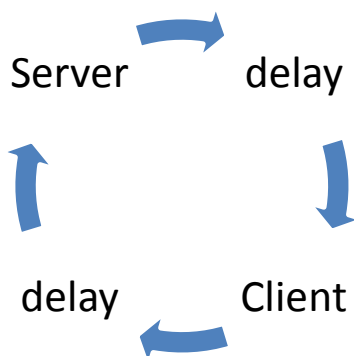
First of all thank you all for supporting our Corona® tools, we want to be able to make more tools for you and we appreciate your support.

Well we will show you how to make a multiplayer game from the ground up using AutoLAN and the Corona® physics engine. Before getting into the implementing the code, let us consider some common multiplayer design considerations.

Determinism

Your code must run exactly the same on all clients and servers. Anything that perturbs this balance will result in your two games going out of sync. In fact if you accomplish this diligently then you can get away with only sending game commands when they happen and just blindly rely on the game being in sync. However, there are some problems with this that we will discuss shortly.

Network latency



Sending a packet from point A to point B takes time and depending on the network traffic this can be significant even on a local network. If you are designing a turn based game like chess then this is not so much an issue but it becomes a problem for real time action games like air hockey. You will need to design some sort of client side prediction algorithm that can anticipate what the current state is based on the state sent by the server (which is indeed from the past.) The Corona® physics engine will help us a lot here because it is deterministic and allows us to accurately predict the next game state effortlessly. We will be designing a physics based pong game to demonstrate some of these problems and provide a strategy for handling

client/server mismatches.

FIGURE 1: DATA FROM ANYWHERE ELSE BUT HERE IS OLD DATA.

Protocol

TCP is not designed for network gaming. Although it is great for transferring files because it ensures packets arrive and arrive in the right order, sometimes we do not want to wait for packets to be present for time critical applications. Also, in most fast paced games we do not care if packets are out of order, we are just interested in the most recent state of the game. For these reasons we do not use TCP for games but instead create our own protocol on top of UDP. UDP provides no guarantees that your packets will arrive at all or in the proper order but it is much faster and uses less overhead than TCP. Our protocol allows you to choose which packets are important and which packets we can tolerate being lost. For example, in a real time air hockey type game we must ensure the clients know that one of the players has scored a point so that the game can be reset.

Network Traffic Control

UDP provides no flow control, you may send as many packets as you wish however fast as the network interface will allow. However, this presents problems when you send more packets than can be physically transferred by the network and these excess packets enter into buffers. Eventually these buffers will overflow and packets will be dropped resulting in poor network performance. Our best advice for the programmer to prevent this from happening is to use the fewest amount of packets possible and to use the smallest packet sizes. If your game state is simple then you can probably get away with sending out the game state every frame, however, for more complicated game states you should send out the differences rather than the entire state. We discuss a few tricks and tips when we start designing the sample game.

Random numbers

Fortunately, a piece of code on one machine runs pretty much exactly the same as a piece of code on another machine and therefore we can count on the two machines remaining in sync for a reasonable amount of time. Obviously random numbers can ruin this predictability unless you make this run exactly the same on all clients/servers. You can do this by sending the random number seed or by tabulating a random sequence and synchronizing it among all clients. Again, we cannot stress enough that the game logic code **must be deterministic** and run **exactly** the same on client and server alike or the games will eventually diverge.

Floating point numbers

Floating point numbers have been notorious for acting differently across different processors and platforms. For example, 5.0-1.0 can be 3.999999 on one architecture and 4.0 in another. This tiny error can lead to a butterfly effect that may eventually cause the simulations to diverge. Although we cannot do anything about this in lua as all number values are floats, we must be aware of this phenomenon and occasionally send entire game state synchronizing packets.

So in summary

1. All packets you receive are from the past
2. Your game must run exactly the same on clients and servers alike
3. Only send what you need to send and not as often as you can

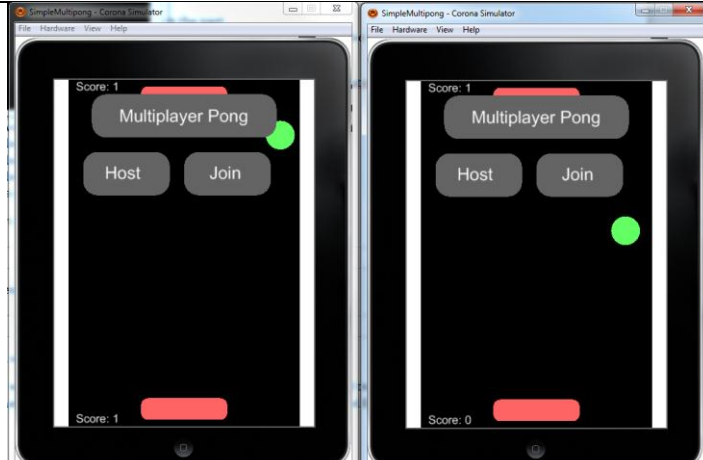
Now let us start our walkthrough of making a multiplayer game with autoLAN. Here are the features we will implement.

1. Two player physics based pong game
2. Allow multiple servers to exist on the same network
3. Allow players to choose to host or to join
4. Allow players to choose what servers to play on

5. If a server is full allow players to spectate matches
6. Once we have a fully functioning local network game, we will extend it to the internet!

Ok let's get started!

STEP 1: OPENING THE GAME

	<p>Please open up two instances of the Corona® Simulator and open up tutorials/SimpleMultipong/main.lua in both of them. Your screen will look like this:</p>
	<p>Now let's click "Host" on one simulator and "Join" on the other. You may also build the program for your device or download it from the app stores (iTunes may not be approved at the moment.) Now your simulator should look like this. Note that if you are beta testing the internet version local games will be white color whereas internet games will be in blue.</p>
	<p>The paddle in blue indicates you can control this paddle with the mouse/touch. Please click on the server that shows up. If you have multiple servers then you should see multiple buttons on the menu. Now your screen should look like this:</p>

Now you should be able to control both paddles and see them move on both screens in synchrony. Any clients joining after this point will be spectators. There is no limit to how many clients you can have (of course network bandwidth and performance willing.)

STEP 2: DESIGNING THE GAME SCENE

Let's open up main.lua in Tutorial/ and look at some code.

The scene will consist of 4 walls to keep the ball in the game screen and two paddles that will interact with the ball. Two collision listeners are added for the top and bottom wall to keep track of score. Currently none of the paddles are controllable and we will make it more interactive in some later steps.

Here are some important multiplayer topics worth discussing. Notice if you replay the program it will evolve the exact same way every single time. This is because of the deterministic nature of the physics engine. In fact, even if we have no network connection, two simulations started at roughly the same time will stay in sync. This will allow us to stream simple game commands instead of the entire game state. Only occasionally will we need to stream the entire game state to account for small errors in the simulation.

Position Interpolation

Strategy A

```
paddle.touchJoint = physics.newJoint( "touch", paddle, paddle.x, paddle.y )  
paddle.touchJoint:setTarget(event.x, body.y)
```

If you noticed we gave each of the paddles a touch joint and we intend to use this to control the x and y position of the paddles. Why not just move the paddle based on the touch event like this?

Strategy B

```
body.x,body.y = event.x,event.y --do not do this!!!
```

The answer is because we want to smoothly interpolate between server updates. If we just set body.x and body.y directly then if a packet gets lost or arrives a little bit late then the paddle will jump across the screen and gameplay will not be smooth. Also, with strategy B we will have to stream packets from the server to the client every frame for the animation to appear smooth. This is a waste of bandwidth and this may not be possible in most circumstances. Using a physics touch joint will effectively smooth out movements by "filling in" the x,y positions of the paddle in between server updates. Best of all it uses the physics engine which is deterministic and a movement on the server will result in the same movement on the clients.

Now you may ask the question: what about the ball? Where are the touch joints? How will we do interpolation? There are no touch joints on the ball because we do not need them. The ball's movements are totally controlled by the physics engine and its movement will be deterministic and predictable even without any server correction. In an ideal world we would never need to correct the ball's position but because of reasons stated earlier in this document (namely network latency and floating point error) we must occasionally send the ball's position and velocity. Now let us get to some network programming!

STEP 3: AUTOLAN SPECIFICS

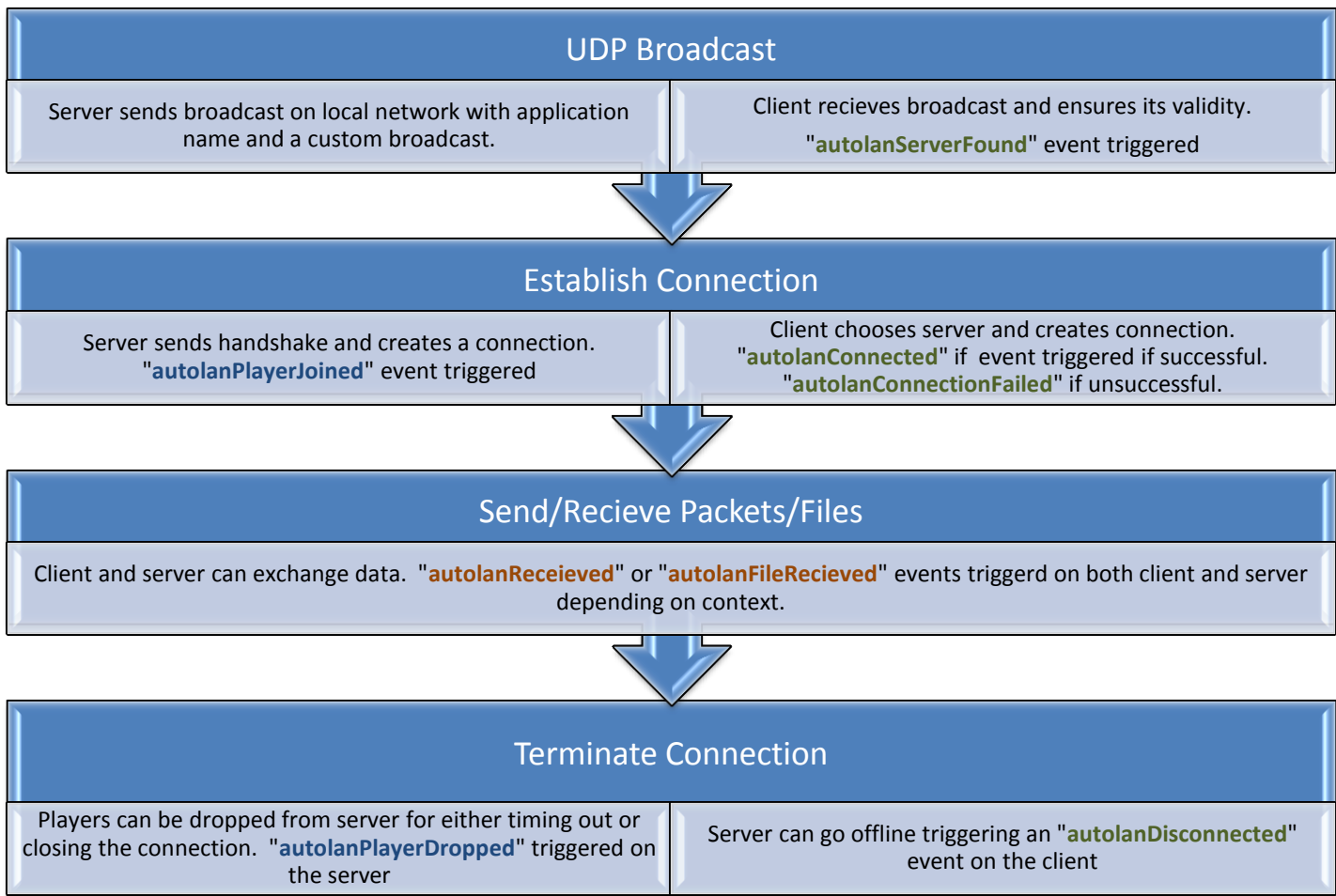


FIGURE 2: CONNECTION FLOW DIAGRAM

Above is a diagram of a typical connection. AutoLAN is 100% non-blocking meaning that it never waits for packets to arrive or to be sent and it will not slow down your code. AutoLAN notifies you of certain events that take place so that you can react to them (i.e. update UI elements, update game state, view files, etc.) Please look through the diagram to understand when these events are called. Server side events are in **blue**, client side events are in **green**, both client/server side events are in **orange**. We will be using these events to drive everything in our game.

STEP 4: MAKING A CONNECTION

Now that you have a conceptual idea of what is going on, we are now ready for some code. Below is a table walking you through how we can implement a multiplayer lobby system and how to establish a connection to a server.

	Client Side	Server Side
Initialization	<pre> local function makeClient() client = require("Client") client:start() client:scanServers() isClient = true end </pre>	<pre> local function makeServer() server = require("Server") server:setCustomBroadcast("1 Player") server:start() isServer = true menuGroup:removeSelf() end </pre>
Discussion	<ul style="list-style-type: none"> client:start() will start the client side code. You should set options before calling this but in our case the default is good enough. client:scanServers() – this will begin scanning the local network for available servers. As servers are found they are reported as events. 	<ul style="list-style-type: none"> server:start() will start the server side code. You should set the options before calling but in our case the default is fine. server:setCustomBroadcast()- you can tell clients the status of this server (ie how many players) as they discover it on the network.
Choose Servers	<pre> local numberOfServers = 0 local function createListItem(event) <i>--displays found servers</i> local item = display.newGroup() item.background = display.newRoundedRect(item,20,0,screenW-50,60,20) item.background.strokeWidth = 3 item.background:setFillColor(70, 70, 70) item.background:setStrokeColor(180, 180, 180) item.text = display.newText(<u>item,event.serverName.."</u> <u>..event.customBroadcast, 50, 30, "Helvetica-Bold", 18)</u> item.text:setTextColor(255) <u>item.serverIP = event.serverIP</u> <i>--attach a touch listener</i> function item:tap(e) client:connect(self.serverIP) menuGroup:removeSelf() menuGroup = nil end item.addEventListener("tap", item) item.y = numberOfServers*70+180 numberOfServers = numberOfServers+1 group:insert(item) end Runtime:addEventListener("autoLanServerFound", createListItem) </pre>	Nothing here, client side code.
Discussion	<p>The code above will listen for “autoLanServerFound” events and create UI elements to let the user choose which server to join. It also displays the customBroadcast received from that server; currently these are the number of players in the game but it can be anything you like, even a lua table.</p> <p>When you receive a autoLanServerFound event, you must record the event.serverIP field as this is passed to the connect() function.</p>	
Make a connection	<pre> local function connectionAttemptFailed(event) print("connection failed, redisplay manu") numberOfServers = 0 menuGroup = display.newGroup() spawnMenu(menuGroup) end Runtime:addEventListener("autoLanConnectionFailed", connectionAttemptFailed) local function connectedToServer(event) print("connected") end Runtime:addEventListener("autoLanConnected", connectedToServer) </pre>	<pre> local numPlayers = 1 local clients = {} local function addPlayer(event) print("player joined") local client = event.client <i>--this is the client object, used to send messages</i> <i>--look for a client slot</i> local index = 1 while(clients[index]) do index = index+1 end clients[index] = client client:sendPriority({1,playerNumber = index}) </pre>

		<pre> numPlayers = numPlayers+1 server:setCustomBroadcast(numPlayers.." Players") end Runtime:addEventListener("autolanPlayerJoined", addPlayer) </pre>
Discussion	<p>After we call the <code>client:connect()</code> function we can receive one of two events: "autolanConnectionFailed" or "autolanConnected"</p> <p>A connection attempt will fail if the server went offline between discovery and connection. This is handled by respawning the menu which also begins scanning again. The <code>autolanConnected</code> event simply tells the client that the handshake sequence was successful. Actual data transfer will occur next.</p>	<p>When a player joins the game <code>autoLAN</code> dispatches a "autolanPlayerJoined" event. The <code>event.client</code> object represents the client and this must be saved within the event listener. We suggest you implement an array of client objects so you can send commands to them. We also update the custom broadcast to reflect the new server state. We also send the client its index so that proper adjustments can be made client side. Notice we used the <code>sendPriority</code> function to ensure this first packet arrives since it will drive the game on the client side. We will explain why we send that information in the next section.</p>

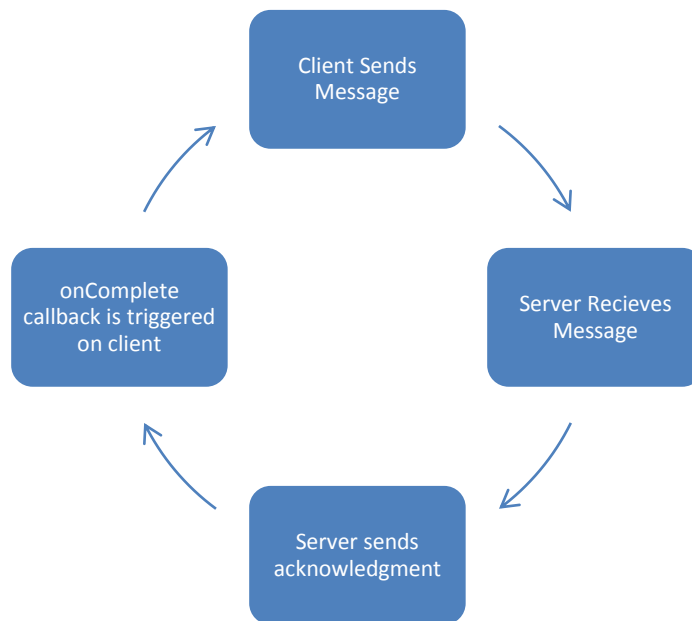


Figure 3: `sendPriority` cycle. Not only is your message guaranteed to be sent to the receiver, but the sender is also notified when the acknowledgement is received. This can be used to drive situations where you absolutely need your client/server to stay in sync.

STEP 5: OUR GAME NETWORKING PROTOCOL

Before we get into lua code we need to design our own network protocol language specific to our game. We will be sending messages to and from server/client but in order to make any sense of it we need to know what kind of message it is. Is it a full game state sync frame or is it a differential? Is it an initialization packet or a normal game loop packet? Here we devised a simple protocol to make order out of chaos. Think of it as a contract between the server and client. Our packets will consist of lua tables. We can save bandwidth by not using associative arrays because we do not need to send the table keys if we agree on the order.

Initialization Packet (From server to clients)

Table index#	1 (packet type)	2 (contents)
contents	1	Player number. Since this is a two player game, we need to know how many players

The initialization packet is sent by the server to the client. Since we are designing a two player game, if the player index received is > 2 then the client will be a spectator and not control the paddle. This will also allow us to reassign control to another player if the one in charge loses connection.

Full Game State Packet (From server to clients)

Table index#	1	2	3	4	5	6	7	8	9
contents	2	Top paddle touch joint target X coordinate	bottom touch joint target X coordinate	Ball.x	Ball.y	Ball x velocity	Ball y velocity	Top Player score	Bottom Player Score

This packet contains the entire game state and is sent to any players that have just joined the game and occasionally sent to all players to make sure they stay in sync. We will send this as a high priority packet to make sure it does not get lost. Notice how we do not stream the Y coordinates because the paddles only move in the X axis and streaming the Y coordinates would be a waste of bandwidth.

Player Update Packet (From client to server)

Table index#	1	2
contents	3	player paddle touch joint target X coordinate

When the user interacts with the paddle on the client side we must inform the server of the change. This is the only packet that is streamed from the client to the server. In our two player game it will only be streamed by one client at a time.

Differential Game State Packet (From server to clients)

Table index#	1	2	3
contents	4	Top paddle touch joint target X coordinate	bottom touch joint target X coordinate

This packet will be streamed to all players updating them on the player's moves. These are the only variables the clients themselves cannot predict and must be streamed as soon as the server receives a player update packet or the paddle on the server side is moved.

Ball Update Packet (From server to clients)

Table index#	1	2	3	4	5
contents	5	Ball.x	Ball.y	Ball x velocity	Ball y velocity

Small errors in the physics simulation occur more so during collision events rather than during steady motion. We will send the ball state to ensure all clients are in sync after each collision on the server and assume the simulation remains reasonably in sync in between collisions.

And there we have it. We will be referring to this card a lot as we write the lua code for client and server side mechanics.

STEP 6: THE GAME LOOP

	Client side	Server side
Saving state	<pre> local function getPlayerUpdate() local state = {} state[1] = 4--protocol id state[2] = paddleUp.targetX return state end </pre>	<pre> local function getFullGameState() local state = {} state[1] = 2--protocol id state[2] = paddleUp.targetX state[3] = paddleDown.targetX state[4] = ball.x state[5] = ball.y vx, vy = ball:getLinearVelocity() state[6] = vx state[7] = vy state[8] = upScore state[9] = downScore return state end local function getDifferentialGameState() local state = {} state[1] = 4--protocol id state[2] = paddleUp.targetX state[3] = paddleDown.targetX print(paddleDown.targetX) return state end local function getBallState() local state = {} state[1] = 5--protocol id state[2] = ball.x state[3] = ball.y vx, vy = ball:getLinearVelocity() state[4] = vx state[5] = vy return state end </pre>
Discussion	<p>Here is where our protocol we designed earlier meets Lua code. The code above is the heart of our multiplayer game where we convert the local game state to a format we can send over the network and our clients will understand.</p>	
Sending the state	<pre> dragBodyClient = function(e) local body = e.target body.touchJoint:setTarget(e.x, body.y) body.targetX = e.x if "began" == e.phase then display.getCurrentStage():setFocus(body, e.id) elseif "ended" == e.phase or "cancelled" == e.phase then display.getCurrentStage():setFocus(body, nil) elseif "moved" == e.phase then --now forward an update to the server client:send(getPlayerUpdate()) end end </pre>	<pre> ballCollision = function(event) if(event.phase == "ended") then --send ball update packet to all clients for i=1, numPlayers do clients[i]:send(getBallState()) end end end sendFullFrame = function() for i=1, numPlayers do clients[i]:send(getFullGameState()) end end serverReceived = function(event) local message = event.message print("got message of type", message[1]) --since this message came from a client, it can only be of type 3: player update paddleUp:setTarget(message[2]) --now forward a differential update to all clients for i=1, numPlayers do clients[i]:send(getDifferentialGameState()) end end dragBodyServer = function(e) local body = e.target body.touchJoint:setTarget(e.x, body.y) body.targetX = e.x if "began" == e.phase then display.getCurrentStage():setFocus(body, e.id) elseif "ended" == e.phase or "cancelled" == e.phase then display.getCurrentStage():setFocus(body, nil) elseif "moved" == e.phase then --now forward a differential update to all clients for i=1, numPlayers do clients[i]:send(getDifferentialGameState()) end end end </pre>

Discussion	<p>In both cases, client and server, we send data packets only when an unpredictable event occurs and let the clients interpolate the rest of the game state based on these “keyframes”. Unpredictable events are:</p> <p><u>Player moving the paddle</u> (on both client and server)</p> <p>This is handled in the touch listener for the paddle (dragBodyServer and dragBodyClient)</p> <p><u>Ball colliding with a surface</u></p> <p>This is handled in the ball collision listener on the server.</p> <p>Everything else should be predictable because of the determinist nature of the physics engine but occasionally we will send full game sync packets to make sure.</p>	
Receiving the state	<pre> clientReceived = function (event) local message = event.message print("message", message, message[1], message[2]) --figure out packet type if(message[1] == 1) then print("got init packet") if(message[2] == 1) then --we are the first player to join, let us take control of the ball paddleUp:addEventListener("touch", dragBodyClient) paddleUp:setFillColor(100,100,255) ballControl = true end elseif(message[1] == 2) then restoreGameState(message) elseif(message[1] == 4) then print("got differential packet",message[2],message[3]) paddleDown:setTarget(message[3]) if(not ballControl) then paddleUp:setTarget(message[2]) end elseif(message[1] == 5) then print("got ball update packet") restoreBallState(message) end end </pre>	<pre> serverReceived = function(event) local message = event.message print("got message of type", message[1]) --since this message came from a client, it can only be of type 3: player update paddleUp:setTarget(message[2]) --now forward a differential update to all clients (some are spectators) for i=1, numPlayers do clients[i]:send(getDifferentialGameState()) end end </pre>
Discussion	<p>The receive function on the client side is a bit more complex on the server because the server does not need to restore game state. This is the client’s implementation of the contract and every protocol id is processed differently. Notice when we receive an initialization packet we check our player ID (contained in the packet itself) and if the client is the first one to join it takes control of the paddle.</p>	<p>The server only receives packets from the client in charge of the paddle. Its only job here is to update the paddle position and relay the update to all the clients.</p>
Dropped players	<pre> connectionAttemptFailed = function(event) print("connection failed, redisplay menu") numberOfServers = 0 menuGroup = display.newGroup() spawnMenu(menuGroup) end </pre>	<pre> playerDropped = function(event) local clientDropped = event.client --go through the table and find the client that dropped for i=1, numPlayers do if(clients[i] == clientDropped) then table.remove(clients, i) --remove this client numPlayers = numPlayers - 1 end end server:setCustomBroadcast(numPlayers.." Players") --now let us try to find a spectator client to retake control of the paddle if(clients[1]) then clients[1]:sendPriority({1,1}) -- initialization packet with playerId = 1 so client can control paddle end print("player dropped because", event.message) end </pre>

Discussion	When the server disconnects we want the main menu to reappear so we can just reuse the same code we used when a server connection attempt failed.	The process is a bit more involved on the server. When a player disconnects we are given the client object, the same one we used to send information. We will then need to find it in our clients table and remove it from the list. If there are other clients (spectators) then we will send an initialization packet to give control of the paddle to that client
Adding Listeners	<pre>Runtime.addEventListener("autolanReceived", clientReceived) --all incoming packets are sent to clientReceived Runtime.addEventListener("autolanConnectionFailed", connectionAttemptFailed) Runtime.addEventListener("autolanDisconnected", connectionAttemptFailed)</pre>	<pre>Runtime.addEventListener("autolanPlayerDropped", playerDropped) Runtime.addEventListener("autolanPlayerJoined", addPlayer) paddleDown.addEventListener("touch", dragBodyServer) --assign bottom paddle to server Runtime.addEventListener("autolanReceived", serverReceived) --all incoming packets sent to serverReceived ball.addEventListener("collision", ballCollision) timer.performWithDelay(2000, sendFullFrame, -1)</pre>
Discussion	Here is where we assign the client/server identity. The client only sends packets when the paddle has been dragged in dragBodyClient. The only client that sends packets is the one controlling the paddle; spectators do not send packets in our game. We also have listeners to handle server disconnects and failed connection attempts.	The server sends packets when it receives a packet from the client, when the ball collides with a surface, when the paddle is moved by the player, and periodically (every 2 seconds) sends a full sync packet to all clients. We also have listeners to handle playerJoined and playerDropped events.

OTHER TUTORIALS

The tutorials folder has other tutorials showcasing the features of multiplayer. A brief description of these are shown below

Custom Broadcast	<p>Please open this as well as QuickStartClient in the Corona® Simulator. Shows you how to use the custom broadcast feature. Custom broadcasts are set up easily using the command:</p> <pre>server:setCustomBroadcast("This will be sent to unconnected clients")</pre>
File Transfers	<p>Please open this as well as QuickStartServer in the Corona® Simulator. This shows you how to transfer files. Files can be transferred bidirectional from client to server and server to client. Just use the command:</p> <pre>client:sendFile("Icon.png", system.ResourcesDirectory)</pre> <p>The first parameter is the filename, the second is the path. A third parameter can also be given to specify what the saved file will be named on the receiving party. All files will be stored in the documents directory when received. All file transfers happen in the background without blocking your code.</p>
Priority Callback	<p>Please open this as well as QuickStartServer in the Corona® Simulator. Sometimes it is useful to know when a priority message was successfully received. When the acknowledgement packet arrives from the server a custom priority callback function is called letting you know the transaction was completed. From within this function you can send more priority callbacks establishing a simple type of flow control. The command to do this is:</p> <pre>client:sendPriority("hello world", {callback = priorityCallback})</pre> <p>You should only use priority messages if you need them. The overhead for sending them is a bit more and the messages need to be buffered in case they need to be resent thereby taking up more memory as well. This is useful for initialization packets, full game state sync packets, or turn based games.</p>
Quick Start Client	<p>Please open this as well as QuickStartServer in the Corona® Simulator. This shows you how to get started with autoLAN as fast as possible. The notable commands are:</p> <pre>local client = require "Client" client:start() client:autoConnect()</pre>

	The autoConnect command will automatically look for servers and connect to the first one it finds. If you would like to make a game lobby type system, please refer to the multiplayer pong game described earlier in this documentation. The multipong game features a simple lobby system.																		
Quick Start Server	Please open this as well as QuickStartClient in the Corona® Simulator. Shows you how to get started with autoLAN as fast as possible. The notable commands are: <pre>local server = require "Server" server:start()</pre>																		
Template	Contains all the autoLAN events and event parameters in a simple to use template file. Breaks down both client side and server side events to let you copy and paste what you need.																		
Options	<p>Gives you a rundown of the options available to you. You can set options on both client and server side like this:</p> <pre>client:setOptions(params)</pre> <p>Where params is a table containing any number of these entries.</p> <p>On the client side:</p> <table border="1"> <tr> <td>timeoutTime</td><td>Number of ms to wait before server is considered dead</td></tr> <tr> <td>applicationName</td><td>Name of the application, server and client must have the same application name to be discoverable. Prevents two applications using autoLAN from discovering each other.</td></tr> <tr> <td>circularBufferSize</td><td>Set this to a power of 2. If you do not use files or priority messages then you probably can leave this low (around 4) but for file transfers you should increase this to the (size of file/packet size) and round to the nearest power of 2.</td></tr> <tr> <td>packetSize</td><td>Size of the packet to split up files into. Maximum UDP packet size allowed by lua is 8192 bytes according to Lua sockets documentation.</td></tr> </table> <p>On the server side:</p> <table border="1"> <tr> <td>broadcastTime</td><td>Number of milliseconds between broadcasts for network discovery. The longer this number, the longer the time it takes to discover but the less network traffic used.</td></tr> <tr> <td>timeoutTime</td><td>Same as client</td></tr> <tr> <td>applicationName</td><td>Same as client</td></tr> <tr> <td>circularBufferSize</td><td>Same as client except must be considerably larger since all clients share the same send buffer.</td></tr> <tr> <td>connectTime</td><td>Frequency to check if a client wishes to connect. Should be the same or faster than the broadcast time.</td></tr> </table>	timeoutTime	Number of ms to wait before server is considered dead	applicationName	Name of the application, server and client must have the same application name to be discoverable. Prevents two applications using autoLAN from discovering each other.	circularBufferSize	Set this to a power of 2. If you do not use files or priority messages then you probably can leave this low (around 4) but for file transfers you should increase this to the (size of file/packet size) and round to the nearest power of 2.	packetSize	Size of the packet to split up files into. Maximum UDP packet size allowed by lua is 8192 bytes according to Lua sockets documentation.	broadcastTime	Number of milliseconds between broadcasts for network discovery. The longer this number, the longer the time it takes to discover but the less network traffic used.	timeoutTime	Same as client	applicationName	Same as client	circularBufferSize	Same as client except must be considerably larger since all clients share the same send buffer.	connectTime	Frequency to check if a client wishes to connect. Should be the same or faster than the broadcast time.
timeoutTime	Number of ms to wait before server is considered dead																		
applicationName	Name of the application, server and client must have the same application name to be discoverable. Prevents two applications using autoLAN from discovering each other.																		
circularBufferSize	Set this to a power of 2. If you do not use files or priority messages then you probably can leave this low (around 4) but for file transfers you should increase this to the (size of file/packet size) and round to the nearest power of 2.																		
packetSize	Size of the packet to split up files into. Maximum UDP packet size allowed by lua is 8192 bytes according to Lua sockets documentation.																		
broadcastTime	Number of milliseconds between broadcasts for network discovery. The longer this number, the longer the time it takes to discover but the less network traffic used.																		
timeoutTime	Same as client																		
applicationName	Same as client																		
circularBufferSize	Same as client except must be considerably larger since all clients share the same send buffer.																		
connectTime	Frequency to check if a client wishes to connect. Should be the same or faster than the broadcast time.																		

THE INTERNET

Now that we have a good grasp on how to use the library for making local network games, we will now turn our attention to extending this to the internet. First of all let us discuss some of the difficulties of making an internet game and some special considerations compared to local network gaming you must take into account.

Network Address Translation (NAT)

The current implementation of IPv4 uses 32 bits per address resulting in somewhere around 4.3 billion unique addresses. This was far more than needed back when the internet was conceived, but things have changed by then. There just are not enough addresses to assign to every single mobile device in the world so NAT was conceived to effectively ameliorate this problem. NAT separates the public global network (the internet) from private networks in homes and businesses, in effect allowing many devices to use the same public IP address. Unfortunately for us this makes it nontrivial to talk to a particular device within a network because not only must you know the public IP address used by

the router, but you must also find a way to address the device itself in the private network. Let us do a walkthrough of how our system works and we will explain things on the way.

Say Bob and Bill want to play a game over the internet. Suppose Bob wishes to be the host. His device will claim an available local port (say port 5100) on the local address (say 192.168.1.101) to start sending and receiving UDP packets. Bob's device will now send a packet to our server at www.mydevelopersgames.com (translated to 64.90.49.111 by the DNS) with port 54613 (we randomly designate this on the server side) informing us that he is now waiting for Bill. When we receive Bob's UDP packet we can read the address and port from which it came from. The address that we see is not 192.168.1.101 but a public IP assigned to Bob by his ISP (say 98.1.1.206). The port we see may be 5100 but on some NATs this can be a random port (let's say 6702). However, the key here is that when we send a packet to the address 98.1.1.206 and port 6702, Bob's router will forward this to the local address 192.168.1.101 and port 5100.

Similarly Bill will send a packet to our matchmaking server and we will log his IP and port. Next we will tell Bob about Bill's IP and port and vice versa. This way Bob and Bill will know where to send their packets and a peer to peer connection can occur without any further intervention from the matchmaker. This procedure is known as "NAT punchthrough."

AUTOLAN SPECIFICS

Assuming you have already designed and debugged your game on the local network it is very easy to extend this to the internet. We suggest you first get everything working on the local network as it is easier to debug and you will be able to get faster code iteration this way (you can have client and server on the same computer on the local network.) Let us take our Multipong example and extend it to the net.

Local network	Internet
<pre>client = require("Client") client:start() client:scanServers() server = require("Server") server:start()</pre>	<pre>client = require("Client") client:start() client:scanServersInternet() server = require("Server") server:startInternet()</pre>

After changing just two lines of code you should get the exact same behavior as local network with all the same events and functions you are familiar with, except that you will be able to discover users on not just the local network but also on the internet. In the current implementation the client will be given a maximum of 10 servers ordered from closest to farthest by geographic location. There are other functions that are worth discussing here that are more important over the internet.

Application Name

It is now more important than ever to give your app a unique application name, as collisions are inevitable with a public matchmaking server. You can fix this problem for good by running the matchmaker script on your own server but If you plan to use our own matchmaker server at www.mydevelopersgames.com port 51650 then you must be **absolutely** sure that your application name will not collide with anyone else as it is impossible for us to determine which application names are colliding. Our suggestion is you go with your package name (like com.MYDevelopersgames.Multipong). To set the application name just simply call this **before** starting the client/server.

```
server:setOptions{applicationName = "someUniqueName"}
client:setOptions{applicationName = "someUniqueName"}
```

Of course, the application name must be the same on client and server for them to be discoverable on both the local network and internet.

Private Matchmaker Server

As stated in the user agreement, we make no guarantee that our matchmaker server will be operational 24/7 since we currently do not have a dedicated server account with Dreamhost. We have no control over who is using server resources at any particular time and how much memory/cpu they are using. We ask that you limit your custom broadcasts to a reasonable size (less than 50 bytes if possible) as these are currently stored in server RAM. Each client/server connection takes an approximate

Ip address	4 bytes
Port	2 bytes
Latitude	4 bytes (float?)
Longitude	4 bytes (float?)
Device ID	Up to 10 bytes
Custom broadcast	Anything (requesting less than 50 bytes)
Total	Around 70 bytes

These are rough estimates but gives us an idea how much memory overhead to expect on the server. In future updates we will store all this stuff in a database but for now just storing them in arrays will do the job. If you wish to use your own webserver then you will need to tell AutoLAN the URL and port to connect to. You can do it like this

```
client:setMatchmakerURL("www.someurl.com", 54613)
server:setMatchmakerURL("www.someurl.com", 54613)
```

Of course, both client and server must point to the same url. Note that we require 2 consecutive ports (in this case port 54613 and 54614.) If your server currently uses those ports you must update the provided php file to another port.

In order to start the php service you must have shell access on your webserver. Simply upload the php files to your server and log in via SSH (we used the putty client) and run php testServerTCP.php. You should not run it as a background process so you can report any errors that occur. Next just open the provided multiplayer pong example on separate devices with separate internet connections. You can even use the 4g or 3g connection on your phone.