Lab 4 **Part 2**
Trojan Blast Bandwidth conservation!

This section of the lab is due at 11:59pm on Tuesday 4/15. This will give you a full week to complete it. Note that all work for this lab should be done in the svn folder for Lab 4 Part 2. It is completely separate from the code for Part 1.

In this part of the lab, you will implement the Replication Manager, our version of the Ghost Manager from Tribes. The Replication Manager will be responsible for sending and receiving "most recent state" for all replicated GameObjects in our world. It will send a create event when an object should be created, send an update even when an object should be updated, and send a destroy message when an object should be destroyed. This is as opposed to the way we did it in the previous lab, in which we sent the state for all objects every frame. The Replication Manager will also handle automatically resending the most recent state whenever a state packet is reported as dropped by the DeliveryNotificationManager.

0) Take a look at **ReplicationManagerServer**. There is one of these per **ClientProxy** now and the class will be responsible for encoding properties and statuses that need to be sent over the network. Take a look at **RepilcationManagerClient**. There is one of these on each client and it is responsible for decoding the data sent by the **ReplicationManagerServer** and applying it to the world.

1) Look at the enum **EShipReplicationState** defined in Ship.h. It has four state bits: one to signify that the location of the ship needs replication, one to signify that the rotation needs replication, one to signify that the color needs replication, and one to signify that the player id needs replication.

2) Since we want to be able to read and write only the properties of the ship that need updating, we have to update **Ship::Write** to take the dirty state, and then only write that state. Take a look at **Ship::Write** and notice that it now has a **uint32_t inDirtyState** parameter. Edit the function so that it writes the location of the ship *only* if the **ESRS_Location** bit is dirty. Remember that when implementing a system like this, you need to first send a flag bit to indicate whether the property is going to be written or not- that way the matching **Read** function can check that bit and decide whether to read the property or not. Make sure to also update the **writtenState** local variable with the **ESRS_Location** bit so that the caller of the function knows that that bit was successfully written.

3) Now do the same for ship's rotation, color and player id. Note that you should _always_ write the **mIsThrusting** bit. This is because it's only one bit, and it would take one bit just for the flag bit to indicate if **mIsThrusting** is present. Therefore it's more efficient to just write it every frame.

4) Update **Ship::Read** so it can read the data the way you wrote it in **Ship::Write**(...). Note that **Ship::Read** does not take a new param for dirty state, because the state is already encoded in the memory stream. All you have to do is check the flag bits to see which properties to read.

5) Take a look at **ETommyCoinReplicationState** defined in TommyCoin.h. Do the same thing for **TommyCoin::Read** and **TommyCoin::Write** that you did for **Ship::Read** and **Ship::Write**. Note that TommyCoin uses **ETCRS_Pose**. If the Pose bit is dirty, then you should write location *and* rotation. We'll assume one doesn't change without the other.

6) Now we have to make sure that when the ship changes location or rotation on the server, it sets the appropriate dirty state. At the beginning of **ShipServer::Update,** cache the location and rotation of the ship in stack variables. Then, at the end of the function, if those values have changed, call **NetManagerServer::sInstance->SetStateDirty(int inNetworkId, uint32_t inDirtyState)** to set the appropriate state bit(s) dirty. This will tell the **ReplicationManagerServer** in each **ClientProxy** that that particular ship needs that particular state replicated.

7) Look at **ReplicationManagerServer::WriteCreateAction, ReplicationManagerServer::WriteUpdateAction** and **ReplicationManagerServer::WriteDestroyAction**. When the **ReplicationManagerServer** decides that it has to replicate a Create, Update or Destroy action for a **GameObject** to a client, it calls those functions.

8) Similarly, the **ReplicationManagerClient** has **ReadAndDoCreateAction, ReadAndDoUpdateAction and ReadAndDoDestroyAction** which receive the data from the server versions of the functions and should act on that data. These functions are left for you to implement. Specifically:

Implement **ReplicationManagerClient::ReadAndDoUpdateAction**. This should first attempt to find the referenced **GameObject** by calling **NetworkManagerClient::sInstance->GetGameObject** and passing in the network Id. Then, it should call **Read** on that object to read the object's state data from the packet.

Implement **ReplicationManagerClient::ReadAndDoCreateAction**. This should read the object four cc name and then create the object if necessary- note that you should check for the existence of the object by network id before creating it: you might receive duplicate create packets due to an ack for a create being dropped, or some other reason. After the object is created, again read the object's state.

Implement **ReplicationManagerClient::ReadAndDoDestroyAction**. Try to find the object (again, you might have received a double packet so the object might no longer exist) and mark it as wanting to die. Make sure you also remove it from the network id to game object

map by calling **NetworkManagerClient::sInstance->RemoveFromNetworkIdToGameObjectMap**.

9) Finally, it's time to implement reliability! Find **ReplicationManagerTransmissionData::HandleCreateDeliveryFailure.** This gets called when a create packet fails to deliver. You need to tell the **mReplicationManagerServer** to replicate another create action (check the class declaration for the appropriate function). For the initial state, find the object you're replicating and call **GetAllStateMask**- this will give you a mask of all the state, which is what you want to send when creating something. Also, make sure the object still exists- it's possible it's already been destroyed on the server, in which case there's no point replicating the create again.

10) Similarly, implement **ReplicationManagerTransmissionData::HandleDestroyDeliveryFailure**. Just tell the mRepilcationManager to replicate the destroy action again.

11) For the final method implement **ReplicationManagerTransmissionData::HandleUpdateStateDeliveryFailure**. This needs to tell the mReplicationManager to **SetStateDirty** on any state that failed to deliver, and which hasn't already been sent in an inflight packet already (because remember, more up-to-date data might already be on the way). To do this, you must get the inflight packet list from the DeliveryNotificaitonManager, run through the inflight packets to see if any contain updates for this network object, and if so, remove the state that is updated in them from the state that you need to resend. When you're all done, if there's any state left to resend, call **SetStateDirty** on the mReplicationManagerServer with the state.

12) finally you're ready to test! start up the game and check out how much lower your bandwidth is than it used to be (because I'm sure everything is working, right?) Most of the bandwidth that's left is actually the super inefficient scoreboard replication system that's left over.

13) If everything worked out, now turn on simulated packet loss at the client- in Client::Client, uncomment the line: **NetworkManagerClient::sInstance->SetDropPacketChance(0.6f);** Start up a game with 4 clients. Your game should be very stuttery, but you should at least be getting correct information, and all your ships should eventually be in sync (start them all flying around.) Also, all your ships should get their color information correctly. Finally, any coins that you pick up in one game should disappear on all the clients (eventually- it might take a second or two for the destruction of the coin to replicate everywhere without getting dropped)

Congratulations, you're done with lab 4, and you've implemented an efficient reliability layer on top of UDP!