Lab 4 **Part 1**
Trojan Blast Bandwidth conservation!

In this lab, you will extend the work you've done on TrojanBlast, adding a DeliveryNotificationManager (similar to the Tribes Connection Manager) and a ReplicationManager (similar to the Tribes Ghost Manager).

Many of the data structures have been created for you and will be discussed in class. Your job is to fill in the missing functionality.

In Part 1, you will implement the DeliveryNotificationManager, and in Part 2 the ReplicationManager . **Part 1 is due Tuesday, April 8th, at 11:59 pm**. Although a working DeliveryNotificationManager will be necessary for the completion of Part 2, you will not receive credit for work done on the DeliveryNotificationManager after Part 1 is due. **THIS IS A MANDATORY DEADLINE**.

You should already be familiar with how to run Trojan Blast- that remains the same. For this lab, though, use the version of TrojanBlast provided in the TrojanBlastLab4Part1 folder in svn. Make sure to commit your work in regularly.

1) Examine
DeliveryNotificationManager
This is the class you will be implementing in this part of the lab.

2) Examine
DeliveryNotificationManager::AckRange
This is a class for keeping track of a range of acks- it allows us to ack several packets at once.

3) Examine
DeliveryNotificationManager::InFlightPacket
This is a class for keeping track of a dispatched packet before it has been ack'd or nack'd.

4) Implement
DeliveryNotificationManager::InFlightPacket*
DeliveryNotificationManager::WriteSequenceNumber(MemoryOutputStream& inMemoryStream)
This should write the next **mNextOutgoingSequenceNumber** to the packet and increment it. Additionally, if **mShouldProcessAcks** is true, it should create a new **InFlightPacket**, add it to the **mInFlightPackets** list and return it, so that in Part 2 you can store Transmission data in it. MAKE SURE TO INCREASE YOUR **mDispatchedPacketCount** here.

5) Implement

DeliveryNotificationManager::ProcessSequenceNumber(MemoryInputStream& inMemoryStream)
This function should read a sequence number from the packet and respond appropriately.
It should use and update **mNextExpectedSequenceNumber**.
If the sequence number is as expected, add an ack to the pending ack list by calling **AddPendingAck**.
If the sequence number is too low, silently drop it (we don't reuse sequence numbers, so we don't have to ack it)
If the sequence number is too high, it means some packets were dropped. Adjust your expected sequence number appropriately, and do whatever else is appropriate.
Return true if the packet should be processed, or false if it should be ignored.

8) Implement
DeliveryNotificationManager::AckRange::Write(MemoryOutputStream & inMemoryStream) const
and
DeliveryNotificationManager::AckRange::Read(**MemoryInputStream & inMemoryStream**)
To save bandwidth, these methods should make use of the fact that the **AckRange mCount** will usually be 1. Also, under no circumstances should they send a count of more than 127, so that at most 7 bits are required for mCount. If any acks get dropped because of this, it will just be handled automatically as if the packets were dropped so you don't have to worry about it.

6) Implement
DeliveryNotificationManager::WriteAckData(MemoryOutputStream & inMemoryStream)
This function writes ack data into the packet.
Efficiently write data into the packet to indicate whether you have any acks in **mPendingAcks** or not. If you do, write the first **AckRange** in **mPendingAcks** and remove it from the list.

7) Implement
DeliveryNotificationManager::ProcessAcks(MemoryInputStream& inMemoryStream)
check if the packet has acks in it. If so, read the **AckRange** and respond appropriately.
For our current strategy, let's assume that if any **InFlightPacket** has a sequence number LESS than any ACK that comes in, then we consider the packet dropped.
Call **HandlePacketDeliveryFailure** to report a dropped packet.
Make sure to clear out the **mInFlightPackets** list of any packets you can.
If a packet is successfully delivered, make sure to increment **mDeliveredPacketCount**.

9) Implement
DeliveryNotificationManager::ProcessTimedOutPackets()
This method should run through the in-flight-packets and report any as dropped that aren't ack'd within **kDelayBeforeAckTimeout**. Take advantage of the fact that the packets in the list should be ordered by time.

Now run the game! Hopefully everything is working. If it isn't, fix it until it works.

10) Notice your client upstream bandwidth increased because of all the acks you're sending. Figure out a way to decrease your upstream bandwidth without altering gameplay. Get your bandwidth back down to under 300 Bps. You should be able to do this without altering any of the code you've written for Lab 4 Part 1. You'll need to find a function that's currently using more bandwidth than necessary and optimize it. Think about what the client is sending to the server and figure out how to reduce it WITHOUT affecting game play in any way.

11) As a true test of your system, enable the packet dropping simulation. Find the line:

//NetworkManagerClient::sInstance->SetDropPacketChance(0.2f);

In the client and uncomment it. Run the game. When you do, you'll notice that it's all choppy, because we haven't built client-side-prediction yet (that's going to be in lab 5).

Play for around 120 seconds. Every 5 seconds you'll notice the server logging out the delivery and drop rate to each of the connected clients. Specifically, you'll see lines like

Client 'Jaqen' destructor. Delivery rate xx%, Drop rate xx%

except the x's will be numbers. This is the server's measurement of how many packets were dropped and delivered to a client based on your notification system. Don't worry if the numbers don't add up exactly to 100- there might be packets in flight or rounding errors. However, after playing for 120 second, those numbers SHOULD be close to 80% and 20%, because the simulated drop chance is 0.2 (which would mean roughly 20% of the packets get dropped). Try this with simulated drop rates of 0.05 and 0.1 as well and make sure your results are sensible. If they are, and the game still runs as expected, then you've done the lab correctly, good work!