SkyNet Server Event Time Synchronization

By: Justin, with help from Andy and Will

It seems clear that the Model aspect of the project depends on the order in which events occur. For the purposes of synchronization, consider this scenario: two tanks, A and B. A wishes to attack tank B with a disabling weapon; tank B wishes to move into an empty nearby game tile. If A were to attack B first, B would be unable to move. However, if B moves first, A’s attack misses. The outcome must be the same on both servers.

Furthermore, consider that these 3 tiles (our ‘region’) is being mirrored across two servers for redundancy purposes. So what I imagine looks something like this:
It is not necessarily the case that that’s how the model wishes to represent the game. However, it is a useful test-case scenario for determining how synchronization will occur because it easily demonstrates how two servers may get out of synch.

The basis of the synchronization system we devised is that all servers share a coordinated time. It is not necessary that the servers’ clocks are set the same (although that is preferred) because they can simply calculate the delta between their local time and the agreed on coordinated time.

Using a coordinated time means that events can be ordered in time. If server X declares that an event happened at \( t = 100 \), then that event time can be properly ordered with another event occurring at \( t = 101 \) by every server in the network.

The system for synchronizing events works like this: say server X receives some event, for example, the user moving tank B to another tile. Server X attaches its current time to the event and sends the event out to the rest of the network. At the same time, server Y receives an attack command for tank A against tank B. That event is given a time by server B and send out to the network.

Now, the network layer is buffering these events. All servers in the network agree on a time delta, probably based on network latency. Whenever an event is sent out, the time delta \( \Delta \) is used to say, “when an event arrives with originating time \( t \), the time we handle this event is \( t + \Delta \)”. If the delta is larger than any network latency, then all events can be processed in a uniform way on all servers.

In reality, the server system would probably buffer messages a long time before they are actually delivered to the model. The model will likely run on a tick-based system, which means the network is buffering some number of ticks in advance (possibly determined by network latency). In this way, the network can guarantee that any event sent out at some certain time is ordered properly with other events near the same time generated on other servers.

Another important concept to discuss is the idea of a single event being generated simultaneously across all servers. In the case mentioned above, a user issues a command to move a tank, and that command originates on a single server. However, consider a time bomb. All servers will know about the bomb, and thus when it explodes, all servers may end up generating the same “explosion” event notification.

Synchronization of this explosion is possible because that type of event should not be given the timestamp of the local server. Instead, that event should receive a timestamp relative to the originating time of the bomb. That is, if a bomb is created at time \( t \) and it explodes 10 seconds later, the servers all share creation time \( t \) and generate explosion events at time \( t + 10 \) (instead of whatever the server time is then, which may be slightly different at each server).
What the model cares about

The key to the system is the buffering. By buffering some number of ticks ahead, all events are sure to arrive before that tick is actually “processed” inside the model. The only reason buffering is necessary is to ensure that once a message is sent to the model, every message before it has been processed. Buffering gives the network time to guarantee that all messages are delivered.

The model depends on this behavior, however. I do not think the model can depend upon ticks as a matter of game play dynamics. That is, ticks should not be considered in any part of the model other than that which interacts with network code. Ticks may only be an artifact of the reality that it might be easiest/best to deliver network events to the model in bunches.

It is mostly irrelevant how the network delivers the model its messages so long as they are properly ordered. I would suggest, however, that the network consider an entirely event-based system. To clarify, I mean this: (1) Messages come into the network code (2) As soon as the network can determine that no other message comes before a given message, that message is delivered to the model (3) The model changes its state accordingly.