TAGE NPC control, Behavior Trees, Tick+Think

Server Side:

```
public class GameAIServerUDP extends GameConnectionServer<UUID>
{  NPCcontroller npcCtrl;

   public GameAIServerUDP(int localPort, NPCcontroller npc)
   {     super(localPort, ProtocolType.UDP);
         npcCtrl = npc;
   }

   // --- additional protocol for NPCs ----

   public void sendCheckForAvatarNear()
   {  try
      {  String message = new String("isnr");
         message += "," + (npcCtrl.getNPC()).getX();
         message += "," + (npcCtrl.getNPC()).getY();
         message += "," + (npcCtrl.getNPC()).getZ();
         message += "," + (npcCtrl.getCriteria());
         sendPacketToAll(message);
      }
      catch (IOException e)
      {  System.out.println("couldnt send msg"); e.printStackTrace(); }
   }

   public void sendNPCinfo()
   { ... }
   public void sendNPCstart(UUID clientID)
   { ... }

   @Override
   public void processPacket(Object o, InetAddress senderIP, int port)
   { ...
         // Case where server receives request for NPCs
         // Received Message Format: (needNPC,id)
         if(messageTokens[0].compareTo("needNPC") == 0)
         {  System.out.println("server got a needNPC message");
            UUID clientID = UUID.fromString(messageTokens[1]);
            sendNPCstart(clientID);
         }

         // Case where server receives notice that an av is close to the npc
         // Received Message Format: (isnear,id)
         if(messageTokens[0].compareTo("isnear") == 0)
         {  UUID clientID = UUID.fromString(messageTokens[1]);
            handleNearTiming(clientID);
   }  }  }

   public void handleNearTiming(UUID clientID)
   {  npcCtrl.setNearFlag(true);
   }

   // ------------  SENDING NPC MESSAGES -----------------
   // Informs clients of the whereabouts of the NPCs.
   public void sendCreateNPCmsg(UUID clientID, String[] position)
   {  try
      {  System.out.println("server telling clients about an NPC");
         String message = new String("createNPC," + clientID.toString());
         message += "," + position[0];
         message += "," + position[1];
         message += "," + position[2];
         forwardPacketToAll(message, clientID);
      } catch (IOException e) { e.printStackTrace(); }
   }
```

```
public class NetworkingServer
{  private GameAIServerUDP UDPServer;
   private NPCcontroller npcCtrl;

   public NetworkingServer(int serverPort)
   {  npcCtrl = new NPCcontroller();
      // start networking server

      try
      {  UDPServer = new GameAIServerUDP(serverPort, npcCtrl); }
      catch (IOException e)
      {  System.out.println("server didn't start"); e.printStackTrace(); }

      npcCtrl.start(UDPServer);
   }

   public static void main(String[] args)
   {  if(args.length == 1)
      {  NetworkingServer app = new
                              NetworkingServer(Integer.parseInt(args[0]));
} } }
```

```
public class NPC
{  double locationX, locationY, locationZ;
   double dir = 0.1;
   double size = 1.0;

   public NPC()
   {  locationX=0.0;
      locationY=0.0;
      locationZ=0.0;
   }

   public void randomizeLocation(int seedX, int seedZ)
   {  locationX = ((double)seedX)/4.0 - 5.0;
      locationY = 0;
      locationZ = -2;
   }

   public double getX() { return locationX; }
   public double getY() { return locationY; }
   public double getZ() { return locationZ; }
   public void getBig() { size=2.0; }
   public void getSmall() { size=1.0; }
   public double getSize() { return size; }

   public void updateLocation()
   {  if (locationX > 10) dir=-0.1;
      if (locationX < -10) dir=0.1;
      locationX = locationX + dir;
} }
```

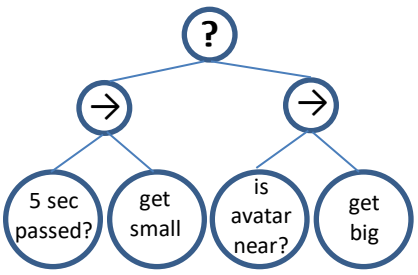
```
import tage.ai.behaviortrees.BTCondition;
public class AvatarNear extends BTCondition
{  NPC npc;
   NPCcontroller npcc;
   GameAIServerUDP server;

   public AvatarNear(GameAIServerUDP s, NPCcontroller c, NPC n,
                                                   boolean toNegate)
   {  super(toNegate);
      server = s;  npcc = c;  npc = n;
   }

   protected boolean check()
   {  server.sendCheckForAvatarNear();
      return npcc.getNearFlag();
} }
```

*Similar classes for **GetBig**, **GetSmall**, **OneSecPassed***

```
public class NPCcontroller
{  private NPC npc;
   Random rn = new Random();
   BehaviorTree bt = new BehaviorTree(BTCompositeType.SELECTOR);
   boolean nearFlag = false;
   long thinkStartTime, tickStartTime
   long lastThinkUpdateTime, lastTickUpdateTime;
   GameAIServerUDP server;
   double criteria = 2.0;

   public void updateNPCs()
   {  npc.updateLocation();
   }

   public void start(GameAIServerUDP s)
   {  thinkStartTime = System.nanoTime();
      tickStartTime = System.nanoTime();
      lastThinkUpdateTime = thinkStartTime;
      lastTickUpdateTime = tickStartTime;
      server = s;
      setupNPCs();
      setupBehaviorTree();
      npcLoop();
   }

   public void setupNPCs()
   {  npc = new NPC();
      npc.randomizeLocation(rn.nextInt(40),rn.nextInt(40));
   }

   public void npcLoop()
   {  while (true)
      {  long currentTime = System.nanoTime();
         float elapsedThinkMilliSecs =
            (currentTime-lastThinkUpdateTime)/(1000000.0f);
         float elapsedTickMilliSecs =
            (currentTime-lastTickUpdateTime)/(1000000.0f);

         if (elapsedTickMilliSecs >= 25.0f)
         {  lastTickUpdateTime = currentTime;
            npc.updateLocation();
            server.sendNPCinfo();
         }

         if (elapsedThinkMilliSecs >= 250.0f)
         {  lastThinkUpdateTime = currentTime;
            bt.update(elapsedThinkMilliSecs);
         }
         Thread.yield();
   }  }

   public void setupBehaviorTree()
   {  bt.insertAtRoot(new BTSequence(10));
      bt.insertAtRoot(new BTSequence(20));
      bt.insert(10, new OneSecPassed(this,npc,false));
      bt.insert(10, new GetSmall(npc));
      bt.insert(20, new AvatarNear(server,this,npc,false));
      bt.insert(20, new GetBig(npc));
}  }
```

TICK

THINK



Client Side:

```
public class MyGame extends VariableFrameRateGame
{  ...
   private ObjShape npcShape;
   private TextureImage npcTex;

   public ObjShape getNPCshape() { return npcShape; }
   public TextureImage getNPCtexture() { return npcTex; }
```

```
public class GhostNPC extends GameObject
{  private int id;

   public GhostNPC(int id, ObjShape s, TextureImage t, Vector3f p)
   {  super(GameObject.root(), s, t);
      this.id = id;
      setPosition(p);
   }

   public void setSize(boolean big)
   {  if (!big) { this.setLocalScale((new Matrix4f()).scaling(0.5f)); }
      else { this.setLocalScale((new Matrix4f()).scaling(1.0f)); }
   }
   ...
}
```

```
public class ProtocolClient extends GameConnectionClient
{  ...
   private GhostNPC ghostNPC;

   // ------------ GHOST NPC SECTION ------------
   private void createGhostNPC(Vector3f position) throws IOException
   {  if (ghostNPC == null)
         ghostNPC = new GhostNPC(0, game.getNPCshape(),
                                    game.getNPCtexture(), position);
   }
   private void updateGhostNPC(Vector3f position, double gsize)
   {  boolean gs;
      if (ghostNPC == null)
      {  try
         {  createGhostNPC(position);
         }  catch (IOException e) { System.out.println("error creating npc"); }
      }
      ghostNPC.setPosition(position);
      if (gsize == 1.0) gs=false; else gs=true;
      ghostNPC.setSize(gs);
   }
   ...
      // more additions to the network protocol to handle ghosts:

      if (messageTokens[0].compareTo("createNPC") == 0)
      {  // create a new ghost NPC
         // Parse out the position
         Vector3f ghostPosition = new Vector3f(
            Float.parseFloat(messageTokens[1]),
            Float.parseFloat(messageTokens[2]),
            Float.parseFloat(messageTokens[3]));
         try
         {  createGhostNPC(ghostPosition);
         }  catch (IOException e) { ... } // error creating ghost avatar
      }  }

      // also for "mnpc" and "isnear" incoming messages
```