H3D Networking Utilities Toolkit

1. Introduction

The H3D Networking Toolkit allows H3D scenes to be shared across a network. It is a 3rd party extension to H3D (www.h3d.org) with features for real-time collaboration. It enables users of the H3D API to link two machines together via the internet and have certain parts of the scenes on each machine interact and keep in sync. The developer achieves this by adding a few extra lines to their H3D X3D scene-graph file. It has been developed by Chris Gunn of the CSIRO Immersive Environments Team. (http://www.ict.csiro.au/staff/Chris.Gunn).

Most of this document (apart for the executive summary) is written assuming that the reader has knowledge of the H3D API, scene-graphs and object-oriented programming. However, no knowledge of networking protocols or software is needed to use the toolkit.

2. Overview

This toolkit allows:

1. Fields of objects to be connected (routed) from one machine to another
2. Bi-directional routing across a network without causing circular event propagation.
3. Dynamic objects that move when pushed by the haptic tool.
4. Grabbing and pulling of dynamic objects.
5. Dynamic objects that can be linked (routed) to each other across a network, and will move in unison, with some resilience to latency.
6. Ability to route together across a network a subset of an array of values (i.e. a subset of an MF field)
7. Mechanisms for compensating for connecting together computers running at different speeds.
8. A course-grained inter-object collision mechanism.
9. A hand-shaking or hand-guiding feature, that can update at haptics rates for smooth physical person-to-person guidance across a network.
10. A mechanism to reduce network latency and also jitter in the latency, to improve mechanical stability in dynamic objects.

An H3D scene-graph is a description of a three dimensional scene and how the objects in that scene interact. It consists of Nodes which are associated in a parent-child hierarchy (e.g. a Group node has a number of children, a Shape node has an Appearance and a Geometry node). Nodes have attributes, called fields (e.g. a Material node has the fields of diffuseColor, shininess etc). The fields of nodes can be connected to each other via routes. In that way, if one field is changed by user-interaction, an event can be sent along a route to some other field (usually in another node) which can also then change if necessary. Chains of routes can propagate events through the scene-graph. When a developer builds a scene-graph s/he connects together the parent child associations of nodes and also connects routes between fields.

The H3D Networking Toolkit extends the concept of ‘routes between fields’, so that a field in a scene-graph on one machine can effectively be routed to a field in a corresponding scene-graph on another machine. This is referred to within the toolkit as a “remote route”. Once in place, these remote routes allow a user’s interaction with objects in a scene to be reflected in another scene running on a different system, across a network. The remote routes are bi-directional, so that two users can simultaneously, and co-operatively, interact with the same scene-graph objects. The interaction involves both graphic and haptic effects, with the proviso that once haptics are involved, care needs to be taken to avoid feedback and instability issues when the latency of the network becomes significant. Components of the toolkit have been developed to directly address these issues and accommodate considerable latency under certain constraints. Using these components, it has been possible to have dual haptic interaction with scene objects over a network with latencies up to 290 mSec. As an example, this has allowed a haptic scene to be shared between a computer in Australia and one in the U.S. (using the Internet2 network). Examples of this interaction are two users grasping a simulated body organ, such as a liver, and stretching it between them, or one user haptically ‘holding the hand’ of the other to guide them within the scene. Also possible, is the ability to draw and indicate in the scene and have those annotations appear on the networked system.

It is important to note that creating a useful and efficient collaborative application may involve the careful selection of which nodes and fields within a scene that it makes sense to connected across a network, and how that connection is made. It may be wasteful to connect every possible field of every node to its equivalent across a network. Instead, it may be more suitable to do local processing on each connected machine for some behaviours, while limiting the network transmission to only those events that need to be synchronized. For example, a field of grass waving in the breeze would not need the motion of each blade transmitted and reproduced. Instead, a single wind vector could be sent and the grass motion reproduced locally. The design of the networking toolkit allows this selective network routing. As such, the toolkit is NOT a utility that will automatically find all fields and hook up two complete scenes without developer input. The ability to pick and choose what is connected to what, gives the developer the flexibility to introduce interesting behaviours, because fields on one machine do not necessarily need to be routed to the identical field or node on the remote scene. Using this, a developer can create a ‘master-slave’ system where actions are not exactly symmetrical. An example of this would be where one of the two connected systems does some physics calculations which are relayed to the other system. Another interesting example of an asymmetric connection, is a system we developed which allowed a user to sculpt clay spinning on a virtual potters wheel. A networked user could simultaneously work on the spinning clay, but could also choose to make the clay spin at a different speed or even around a different axis.

The toolkit is also *not* a system which will transmit a new 3D scene to a remote machine. It is assumed that the H3D file containing the scene has previously been sent to the remote machine so that both machines start up with the same scene in the same state.

As well as the Nodes directly related to networking, the toolkit also contains other Nodes that promote collaboration. The Nodes can be described in four categories:

- Networking Nodes – performing the network messaging
- Object Movement Nodes – allowing objects to be manipulated either individually or in collaboration, whilst accommodating network latency.
- Hand Guiding Nodes – allowing one user to hold and guide another user's hand (actually their haptic tool). These nodes also enable tele-operation of one haptic tool from another across a network.

The toolkit was developed to connect two machines together. However, since the networking has been encapsulated into a scene-graph Node, it is possible to add the node multiple times in the scene-graph. It therefore should be possible to build in a connection to more than one machine, thus allowing three-way (or even N-way) systems, although this has not been tested.

3. Package

The package consists of source files, a MSVisualStudio build environment (modeled on the H3D one), documentation and example x3d files.

The bin directory contains dlls built using MSVisualStudio V9 (2008). If those binaries do not work on your system, you may need to rebuild within your environment. (see section 4).

As with H3D, the dlls contain nodes that can be used in x3d files. You can also inherit from the nodes to develop your own nodes. In most cases, e.g. RemoteCoordPoint, an existing node can be used as a template on how to implement a similar node if needed.

4. Running the examples

There is an example folder with x3d files testing all the classes. The file ExamplesList.txt specifies which x3d file uses which Node. These can be used to see the effect of each node and also as examples of how to use them.

Some of the x3d files will run by themselves with H3DLoad, just like normal H3D files.

Others require a server and client program to run. These all have matching server and client x3d files, identified by the "Server" or "Client" at the end of their filenames.

e.g. BufferedMFieldTestServer.x3d and BufferedMFieldTestClient.x3d

It makes most sense to run these on different machines. If you are doing that, you will need to edit the client version of the file, to contain the hostname or IP address of the server machine in place of the word "server_hostname" in one or two places in the files.

You can, however, run both on one machine, for testing purposes. If that is the case, you will need to run at least one of the two programs in ‘mouse-mode’. To do this, you can put the word localhost” in the file in place of “server_hostname”, but comment out the line

```
<IMPORT inlineDEF="H3D_EXPORTS" exportedDEF="HDEV" AS="HDEV"/>

And uncomment the lines:
<!--MouseSensor DEF="mouse"/>
   <DeviceInfo>
      <MouseHapticDevice DEF="HDEV">
         <MouseSensor USE="mouse" containerField="mouseSensor"/>
         <RuspiniRenderer/>
      </MouseHapticDevice>

   </DeviceInfo-->
```

5. Building the library

Pre-built dlls are included with the package. They have been built using Visual Studio V9. However, the configuration for this build may not match your system. If the examples fail to run, try building as detailed here.

This library is built in the same way as H3D.

Run CMake:

Set source code pointing at your H3DNetworkingUtils/build

Set binaries pointing at your H3DNetworkingUtils/build/mySubDir

Configure

Set CMAKE_INSTALL_PREFIX pointing at your H3DNetworkingUtils

Set EXECUTABLE_OUTPUT_PATH pointing at your H3DNetworkingUtils/bin

Set LIBRARY_OUTPUT_PATH pointing at your H3DNetworkingUtils/lib

Configure again

Generate

Run H3DNetworkingUtils/build/mySubDir/ H3DNetworkingUtils.sln

View/Solution Explorer

Choose Debug

Choose INSTALL

Choose Release

Choose INSTALL

Build solution

6. Detailed Description

The toolkit is open source using the MPL license (`http://www.mozilla.org/MPL`). It consists of C++, X3D and python source code. The C++ code compiles into a dll (dynamically linked library). Individual class documentation is available in the doc directory. You will need the usual H3D environment to work with this library.

The toolkit allows you to link a field of a node on one machine to a similarly typed field of any node on another machine, such that they keep each other informed of their current values. We refer to this connection as a 'RemoteField', since it is analogous to the field routing system within a H3D scene-graph, except that it works remotely, i.e. to another machine.

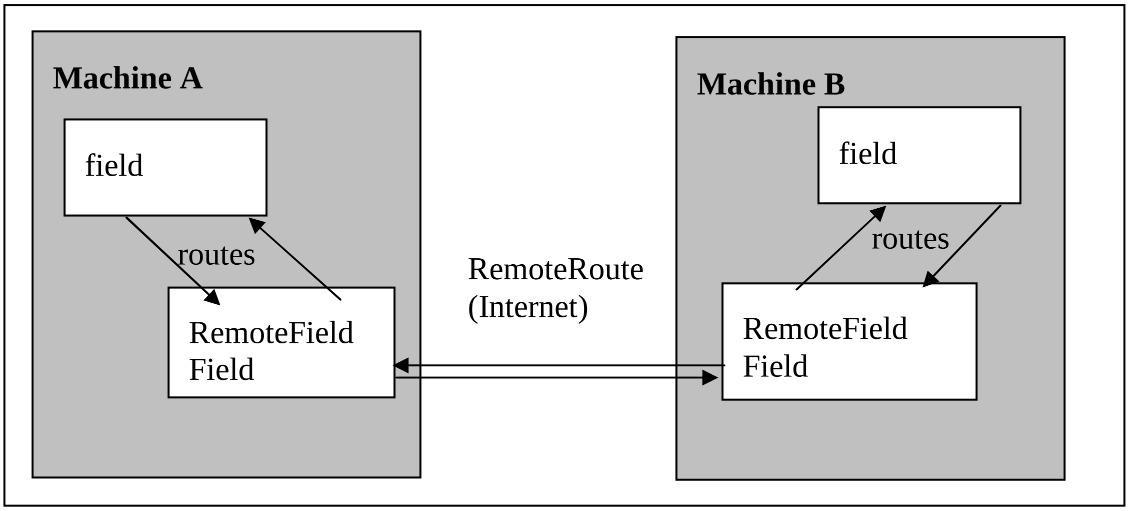

Figure 1. RemoteField connected via field routes.

Typically you would route an H3D field to the **RemoteField** in your scene-graph. The H3D field update system will update the **RemoteField** as soon as an event occurs (**RemoteFields** are always 'AutoUpdate'). The **RemoteField** then sends the field's value to the far machine. Typically, the far machine will have its **RemoteField** routed to some field in its scene-graph. That field will then receive the new value.

Although you can think of the **RemoteField** as a single field, it is, in fact a Node and it contains several fields of its own (see below). One important field of a RemoteField node, is its **fieldId**. Each RemoteField in the scene must have a *unique* **fieldId** number. Unfortunately, in the current implementation, there is no check on this – *beware!*

The advantage of having a node control messages travelling in both directions is that it can prevent a circular event lockup – a message arriving and being sent on to a scene object cannot be reflected back to its source.

6.1. Client, Server, Ports and Sockets

The two collaborating machines must be declared as a *client* and *server*. It doesn't matter which machine is the client or server, as it is only in the initial connection that they differ. The client and server have slightly different scene-graph code, but this amounts to a difference of only two or three lines in the H3D file. The difference is that the server must be ‘listening’ for a connection from *any* client, whereas the client must be told the specific name of the server that it needs to connect to.

They communicate via a *port*, which is identified by a port number. It is convenient to think of a port as the termination of a communications wire from one machine to another. This is obviously not correct, as a computer can have a large number of ports open simultaneously, and typically it only has one physical communications cable. But *logically* this can be assumed. A port is often associated with a *socket*. The socket and port can both be referring to the same communications line termination. A *socket* is a term for software providing the low level code that handles the communications protocol. The *port* is a reference the operating system uses for the line termination, regardless of the socket code using it. The toolkit makes available two socket types: TCP and UDP, which can be used simultaneously in the same scene, for different fields. TCP is slower than UDP, but guarantees that each and every message gets through. (More on this later).

6.2. RemoteConnection

Before a **RemoteField** can send values back and forth, some ‘hand shaking’ and initialization must happen. It is convenient to encapsulate this in a node, called a **RemoteConnection**. A **RemoteConnection** firstly creates extra threads to handle the reading and writing to the communication line (your H3D program probably already has more than one thread running). The **RemoteConnection** then connnects to its corresponding **RemoteConnection** node on the remote machine and prepares for the sending of data. A **RemoteConnection** is, in fact, a **Group** node that is a container for a number of **RemoteField**s. Having done its setup, it then services its children **RemoteField**s, sending their data when necessary. Its **RemoteField**s can be sending several different data types: Vec3f, Vec2f, Float, Int32, Rotation, Bool, Time etc. It can also send arrays of these types.

You won’t actually see the word, “**RemoteConnection**”, in an H3D file, however, since it is actually an abstract node. The implementations for either a server or client are in one of the nodes: **RemoteServer** and **RemoteClient**, which both inherit from RemoteConnection. Then **RemoteTCPServer**, **RemoteTCPClient**, **RemoteUDPServer** and **RemoteUDPClient** inherit from them and are the final concrete instantiations that appear in your H3D file.

There is no difference between the server and client nodes once they are running - they differ only in their startup hand-shaking. A **RemoteServer** starts by ‘listening’ on a port number, when it is opened. This means that any machine that tries to contact it on that port will be ‘heard’. Consequently, a **RemoteServer** needs a field, **listeningPort**, which is an integer providing the port number on which to listen. (It should be noted that care should be taken when choosing a port number – see below).

A **RemoteClient** needs to be told either the host name or IP address of the machine it needs to connect to, as well as the port number. It therefore has these two fields - **remoteHost** and **remotePort**. When it is opened, it sends a message to the specified host and port, and if that host is in fact running a **RemoteServer** on that port number, they will set things going.

The **RemoteServer** or **RemoteClient** should be included in your H3D file within the same scene-graph as your scene objects.

Here is an example of how to include these new nodes in the scene-graph for the server machine:

```
<RemoteTCPServer DEF="server"
      listeningPort="40000"
      open="TRUE">
      <RemoteSFVec3f DEF="graphic_tool_pos" fieldId="0"/>
      <RemoteSFBool DEF="button" fieldId="1"/>
      <RemoteSFInt32 DEF="choice" fieldId="2" />
      <RemoteSFColor DEF="clr" fieldId="3" />
      <RemoteSFFloat DEF="transp" fieldId="4" />
      <RemoteSFRotation DEF="rot" fieldId="5" />
      <RemoteSFString DEF="str" fieldId="6" />
      <RemoteSFTime DEF="time" fieldId="7" />
      <RemoteSFVec2f DEF="particleSize" fieldId="8" />

   </RemoteTCPServer>
```

Figure 2. Example server H3D code

Here is an example of how to include these new nodes in the scene-graph for the client machine:

```
<RemoteTCPClient DEF="client"
      remoteHost="localhost"
      remotePort="40000"
      open="TRUE">
      <RemoteSFVec3f DEF="graphic_tool_pos" fieldId="0" />
      <RemoteSFBool DEF="button" fieldId="1" />
      <RemoteSFInt32 DEF="choice" fieldId="2" />
      <RemoteSFColor DEF="clr" fieldId="3" />
      <RemoteSFFloat DEF="transp" fieldId="4" />
      <RemoteSFRotation DEF="rot" fieldId="5" />
      <RemoteSFString DEF="str" fieldId="6" />
      <RemoteSFTime DEF="time" fieldId="7" />
      <RemoteSFVec2f DEF="particleSize" fieldId="8" />

   </RemoteTCPClient>
```

Figure 3. Example client H3D code

You will note that there are no **RemoteFields** mentioned and that there are, instead, new node types like “**RemoteSFVec3f**”. This is because **RemoteField** is an abstract class. The concrete specializations of it are:

RemoteSFBool for a remote single field of Boolean.
RemoteSFColor, for a remote single field of color.
RemoteSFFloat for a remote single field of float.
RemoteSFInt32 for a Reomte single field of int.

RemoteSFRotation for a remote single field of rotation
RemoteSFString for a remote single field of string.
RemoteTime for a remote single field of time.
RemoteSFVec2f, for a remote single field of Vec2f.
RemoteSFVec3f, for a remote single field of Vec3f.
RemoteSFVec3fPair, for a remote single field consisting of a pair of Vec2f.
RemoteMFBool for an array of boolean
RemoteMFFloat for an array of float
RemoteMFInt32 for an array of int
RemoteMFString for an array of string
RemoteMFVec3f for an array of Vec3f

Note that in the code above, each **RemoteField** has a **fieldId** number. The **fieldId**s of the **RemoteFields** in both the client and server must match. For example if a remote field, of type **RemoteSFVec3f**, has fieldId '7' on one machine, there should be a **RemoteSFVec3f** with fieldId '7' on the other machine, to allow data to flow between the two. The fieldId is used by the system to deliver the correct data to the correct destination, as the system actually multiplexes the different elements of data during the streaming process. Note also that, although they are called "RemoteFields", each RemoteField is not a field at all, but a Node, containing fields of its *own*. The **fieldId** is one of those fields. This is because, although logically it helps to think of **RemoteFields** as fields which can be routed together, they need to have their own attributes associated with them, and therefore these attributes are set via the usual H3D field mechanism.

In the examples so far we have set up a server and client. You may have noted that both the client and server have an **open="TRUE**" field setting. Before connection can happen, the server needs to be "open". However, if the client is "open' at startup, it would repeatedly try to connect to the specified server, and this may be a nuisance if the server was not running yet, as it blocks the main graphics thread occasionally while it tries to connect. A more typical situation would have the client start up with **open** set to **FALSE**, and have some user-interface widget, such as a button, or perhaps a keyboard key, routed to the **open** field, to turn it on when the user decided to try a connection. However, for simplicity in our examples, we will set the client to be open from the start. This will still work, even if the server is started up after the client.

6.3. Routing to the RemoteFields

Having a remotely connected scene will not actually do anything unless we route values in and out of these **RemoteFields**. We have already come across the **fieldId** field of a **RemoteField**. The two other most important fields in these nodes are **toBeSent** and **received.** We route any value that we want to be sent to the other machine, into the **toBeSent** field**.** We route the **received** field to any value that we want to change when the remote end changes (see figure 2)..

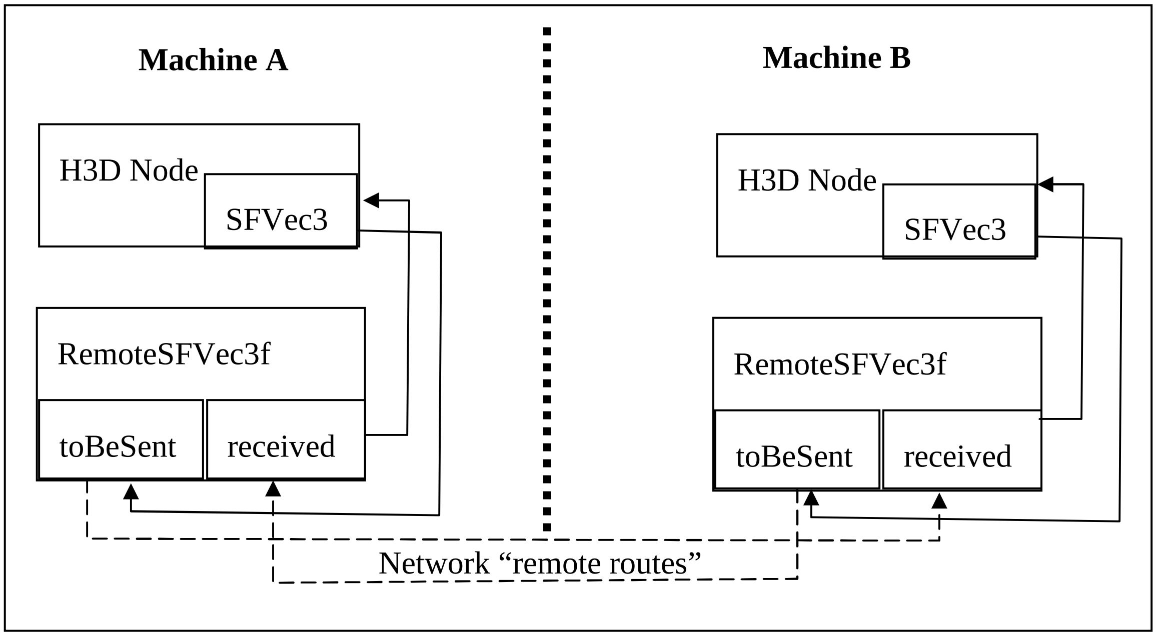

Figure 2. Routing to RemoteFields

You may notice from Figure 2, that there seems to be a circular connection: The SFVec3f on machine A is connected through to the SFVec3f on machine B, but that is also connected back to the original SfVec3f. The toolkit has an in-built mechanism that prevents circular lockups of events. (The data sent across the networked is time-stamped so that the system can identify which event caused a data transmission and can therefore block the same event being reflected back to the originating machine.)

6.4. A Simple Scenario

Suppose we want to display each user's stylus in the scene of each other's machine. Let's assume that, to differentiate it from the local user's stylus, we want to display it as a simple cylinder. We could use the code in figure 3, below (also reproduced in Example1Server.x3d). This creates a graphic to represent the other user's haptic tool, a RemoteClient containing 3 RemoteFields, and routes between the relevant fields.

```
<Group>
   <ImportLibrary url="..\bin\H3DNetworkingUtils_vc9_d.dll"/>

   <IMPORT inlineDEF="H3D_EXPORTS" exportedDEF="HDEV" AS="HDEV"/>

   <!--Represents the other user's tool-->
   <Transform DEF="rem_graphic">
      <Transform rotation="1 0 0 -1.57">
         <Shape>
            <Appearance>
               <Material DEF="mat1" diffuseColor="0.6 0.9 0"/>
            </Appearance>
            <Cylinder radius="0.005" height="0.1"/>
         </Shape>
      </Transform>
   </Transform>

   <RemoteTCPClient DEF="client"
      remoteHost="152.83.70.187"
      remotePort="40000"
      open="TRUE">
      <RemoteSFVec3f DEF="graphic_tool_pos" fieldId="0"/>
      <RemoteSFRotation DEF="graphic_tool_or" fieldId="1"/>
      <RemoteSFBool DEF="button" fieldId="2"/>
   </RemoteTCPClient>

   <!--Sending and receiving the tool position-->
   <ROUTE fromNode="HDEV" fromField="proxyPosition"
          toNode="graphic_tool_pos" toField="toBeSent"/>
   <ROUTE fromNode="graphic_tool_pos" fromField="received"
          toNode="rem_graphic" toField="translation"/>

   <!--Sending and receiving the tool orientation-->
   <ROUTE fromNode="HDEV" fromField="trackerOrientation"
   toNode="graphic_tool_or" toField="toBeSent"/>
   <ROUTE fromNode="graphic_tool_or" fromField="received"
          toNode="rem_graphic" toField="rotation"/>

   <!--Sending and receiving the button-->
   <PythonScript DEF="ps" url="Example1.py"/>
   <ROUTE fromNode="HDEV" fromField="mainButton" toNode="button" toField="toBeSent"/>
   <ROUTE fromNode="button" fromField="received" toNode="ps" toField="boolToFloat"/>
   <ROUTE fromNode="ps" fromField="boolToFloat" toNode="mat1" toField="transparency"/>

</Group>
```

Figure 3. Code for a simple server. Example 1

The code for the corresponding server would be identical, except for three changes:

1. the node **RemoteTCPClient** would be replaced with **RemoteTCPServer**
2. the "**remoteHost**" line would be removed
3. "**remotePort**" would be replaced with "**listeningPort**"

The server code is contained in Example1Server.x3d

Note that in the figure 3, the field that is routed into the **toBeSent** field is not the one that the **received** field at the other end routes to. In our scenario, we are displaying the *remote* user's stylus - we want our *own* stylus to move quite differently. This is an example of asymmetric connection of fields. Connecting up two scene-graphs is not usually a case of simply connecting every field of every node to its equivalent on the other machine.

Note the line:

```
<ImportLibrary url="..\bin\H3DNetworkingUtils_vc9_d.dll"/>
```

This contains the Networking Toolkit and is a dll (dynamically linked library) which is loaded at run time.

6.5. TCP and UDP

The two types of **RemoteConnection** nodes in example1 (figure 3) - **RemoteTCPServer** and **RemoteTCPClient,** are implemented using TCP sockets. TCP sockets are reliable (i.e. they can't lose data and the data is guaranteed to arrive in the correct order), but can be slow or have irregular delivery rates (i.e. jitter). An alternative method is UDP, which is, faster and more regular, but unreliable. That is, a packet of data may become lost altogether and not arrive at all, or packets of data may arrive out-of-order. This may sound dire at first, but there are circumstances, especially in haptic interaction, where this is preferable to having the longer latency and jitter of TCP.

The toolkit has **RemoteUDPServer** and **RemoteUDPClient**, using UDP, which can be used in your scene-graph. In fact, you can have both TCP and UDP running simultaneously in the same scene – but you must make sure that they use different ports. The UDP code is encapsulated in the nodes: **RemoteUDPServer** and **RemoteUDPClient**.

You may be wondering why we might want to use something that is *unreliable*. In fact, a lot of the time the data that we send back and forth need not actually arrive every time – we can get away with missing a bit every now and then. With the reliable TCP, if a packet of data gets lost along the way, the TCP system repeatedly resends it until it is correctly received. This means that there is a delay while the system sorts itself out. To lose a bit of data may be preferable to the delay required to fix the problem. Where we have two parts of the scene that are tightly connected haptically, such as the two tools linked together directly with a simulated spring, we need minimum latency, and can tolerate an occasional lost value, because another, very similar value will be coming along straight away. In that circumstance, we have found that the UDP version works better. It doesn't matter that much if a small movement of one user's hand is lost, as long as we get the most up-to-date position that we can. We are better off getting the most up-to-date value, rather than spending time checking and resending one which would be out of date anyway.

However, is data arrived *out-of-order* it could in fact cause problems. An old position value coming in after the user's hand has moved smoothly further along could cause a momentary force backwards. This is something that cannot be tolerated in our scenes, as a data value arriving that is out of date could cause a violent jitter. The toolkit has its own sequence checking algorithm embedded in the code which will throw away out-of-order data.

In certain circumstances, however, we must use the TCP versions, as we can't afford to lose a single value. An example of this would be a Boolean field - if a change from true to false was lost, the scene behaviour could be dramatically different from that intended.

The UDP implementation has the added advantage that you can schedule it to send data at rates faster than the graphics refresh rate – right up to the haptics refresh rate (1000hz) in fact.

The same **RemoteField** nodes (**RemoteSFVec3f** etc) can be used with both TCP and UDP implementations.

6.6. Routing into the Haptics Thread

Typically, you would route from a **received** field of a **RemoteField** node into a field that is used by the graphics thread of your H3D program. An example might be from a slider value on one machine to a slider value on another, so that the second user would see the slider move. Note, however, that the value can arrive into the RemoteField asynchronously, since the data-reading code has its own thread that is not synchronized with the graphics thread. If the field mechanism fed the data immediately into the destination, it may change the value at an inappropriate time relative to the work that the graphics thread is currently doing.

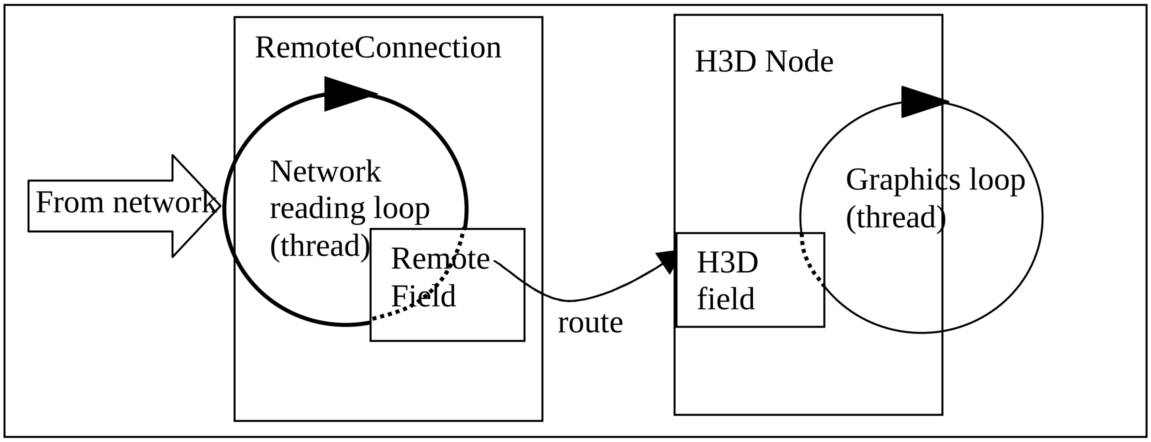

Figure 6. Receiving thread

H3DNetworkingUtils has an inbuilt protection mechanism that prevents this problem. This mechanism delays data updates reaching the destination field until an appropriate time in the graphics cycle.

Sometimes we may want to use data as soon as it arrives (e.g. with values that are to be used in the haptics thread). If this is the case, we can turn off this protection mechanism, to allow the data to go straight through. This can be used in conjunction with the RealTimeAttractor node, which uses locking to extract the data and set it into the haptics thread for immediate use. To do this, the field isHapticField (which by default is FALSE) can be set to be TRUE.

Sometimes you need a value sent as soon as a new connection to the remote system is made, even when the local value has not changed. This is achieved by setting the sendOnConnect field of RemoteField node to TRUE.

As we saw in Figure 3, we are displaying the movement of the *remote* user's stylus and our *own* stylus quite differently. However, there are circumstances when you would like the two networked representations of the *same* logical object to stay 'in synch' with each other - i.e. you want both users to see a single object move in the same way to give the impression that they are interacting with the one, single object. In that case, you can route the same fields of the same nodes in each scene together. There is a mechanism to prevent circular events in this case.

6.7. Making sure you get all values

In an earlier section it was mentioned that, using the TCP version, delivery of all data is guaranteed. This is true within the reading thread of the program. However, it can happen that two or more updates to the same data can arrive on the network within the time taken for a single graphics cycle (remember that the graphics thread is cycling independently of the reading thread). In that case a value may change two or more times before the graphics thread cycles around and is aware of any change at all. This can typically happen if the two machines communicating have different capabilities and are running at different frame rates.

There are certain circumstances where this is undesirable. Take, for example, the case where you are sending a 'button-down' Boolean, generated by the user pushing a button momentarily. If the data for the "down=true" and "down-false" arrive during one graphics cycle, the fact that the button was pushed at all can be lost. To prevent this loss, we can set the **bufferReceivedVals** field of **RemoteField** to TRUE. The effect of this is for the receiving thread to buffer *all* values received, and to release them to the graphics thread one at a time – i.e. the graphics thread is passed one of the new received values each graphics cycle, until the buffer is exhausted. Associated with this is another setting on the RemoteField, **bufferStrategy**. If bufferStrategy is NONE, no buffering is done. If bufferStrategy is SET_ONE_PER_CYCLE, a single received value is released from the buffer to the graphics cycle on each time round the graphics loop. If it is SET_ALL, each received value is released in sequence to the graphics thread, but they all in the released in the next single loop of that thread. SET_ALL only has an effect if the field that you are routing into is AutoUpdate.

6.8. Port Numbers and Firewalls

As mentioned above, we need to allocate a port number for our communications.

The IANA (Internet Assigned Numbers Authority), control and assign various port numbers.

The port numbers 0 through 1023 (called '*well-known'* ports), are assigned for certain common applications. For example port 80 is assigned for web servers.

Then the numbers 1024 through 49151 (*registered* ports), are listed by the IANA as a convenience for the community to develop conventions. For example ports 6000 through 6063 are registered for X window servers.

Numbers from 49152 through 65535 are available for any use. To be safe from conflict, choose your port numbers from this group.

Also, it is common for a computing system to have a firewall installed, protecting it from unauthorized access via the internet. The toolkit will only be able to connect if there are no firewalls between the communicating machines, or, if there are firewalls, a '*pinhole'* through the firewall has been set up. A pinhole is simply an entry in the firewall configuration table that specifies that data from a particular IP address is permitted to a particular port number. This needs to be set up for both TCP and UDP. You systems administrator should be able to do this. Usually administrators like to open up a port to specified external host names or IP addresses only, so they may require both the port numbers and host names to which you will be connecting.

6.9. Multi-valued fields

The toolkit contains some multi-valued **RemoteField**s, e.g. **RemoteMFVec3f**. These work in the same way as the **RemoteSField** nodes, but they send an array of values from one H3D MField to another on another system. However, the MField has no way of telling the **RemoteMField** which of the array values have changed. Because of this, the **RemoteMField** must send *all* the values across the network. For large arrays, this may be wasteful of network resources. The **RemoteNodeField** nodes were created to overcome this. The currently implemented instances of **RemoteNodeField** are:

- **RemoteCoordPoint**,
- **RemoteNormalVector**,

To use these nodes you need to know which values in the array have changed. You may know this from the user input that caused the change. For example, if the user is sculpting a surface, the code that detects the user touching the surface and changes it probably knows which vertices on the surface are being changed. You need to route those indices, along with the MField being changed, to the **RemoteNodeField** node. It then extracts only those changed values and sends them, along with their corresponding indices, across the network to the other machine. The receiving machine then explicitly sets only those values at the other end. The name **RemoteNodeField** comes from the fact that it is designed to work on a particular node and field within that node. For example, a **RemoteCoordPoint** will work on the point field in a Coordinate node.

6.10. Haptic Effects

One of the more interesting possibilities of networked haptic programs is the ability to feel things that the other user is doing. Using the methods described so far, we can move objects in the scene and another user, connected through the Networking Toolkit, will feel that movement if they are grasping the same object. They are feeling the motion of the object, they are not feeling what the other user is feeling.

If we connect through the field network, as we have been discussing so far, the motion is occurring during the graphics loop, typically at about 30 Hz. If a user is directly grasping the object, they will feel this as a vibration or ‘grittiness’ during movement. It would be more desirable to feel the movement at haptics rates, ~1000Hz, as this would feel much smoother. This can be done with the toolkit. To get the best effect we need to firstly use UDP transmission of data, and also we need to receive the data into the haptics thread, not the graphics thread. One useful implementation of this technique is in hand guiding. Using this, one user can effectively grasp another user’s haptic tool and “pull them around” in the 3D scene. This is sometimes referred to as “hand shaking”, but it can be much more useful than the simple novelty of shaking someone’s hand across the internet. It can be used by an instructor to guide a student to a particular spot, and act in a particular way. This can be especially useful in surgical training, but is possibly applicable to other domains as well.

To demonstrate the capabilities of the toolkit we will work through the addition of hand guiding into Example1. Firstly we will need to add a **RemoteUDPServer** and **RemoteUDPClient** for haptics-related updates. We will add **RemoteFields** to these, to send an attraction point from one user’s haptic tool position to the other’s scene and vice-versa. In this way the two tools will be attracted to each other. We will keep the **RemoteTCPServer** and **RemoteTCPClient** in the scene to perform the graphic updates as before. We will also take advantage of two other nodes in the toolkit, **ToolPosDetector** and **RealtimeAttractor**.

ToolPosDetector simply detects the haptic tool position and provides an output field containing it. This sounds just like the proxyPosition or devicePosition in the H3D HapticsDevice node. The difference is that the position is updated at *haptics* rates (~1000Hz) not graphics rates. This gives us a means of sending data across the network faster than the graphics rate, thus helping with a smooth feel at the receiving end.

RealtimeAttractor is a spring force towards a specified point in the scene. It actually inherits from **Attractor**, but differs in that the point in the scene can be changed at rates faster than the graphics frame rate (up to 1000Hz). The changed point is used in the haptics loop to modify the force. It has fields relating to how strong the attraction is and how far it reaches. It also has a **withOffset** field that can prevent the first ‘grab’ of the tool causing a jerk towards the other tool position. If this is set to TRUE, when the attractor becomes active, it uses the current offset of the position as a zero force point and creates forces when either user deviates from there. This allows a user to grasp the other tool back along the stylus shaft, or at some distance from the tip, leaving the tip itself clear of any graphic associated with the grasping tool.

An attractor has an **enabled** field which, when TRUE switches on the hand guiding force.

The extra lines that we need to add to Example1 are shown in figure 7. The full H3D code can be found in Example2Server.x3d and Example2Client.x3d.

```
   <!-- For hand guiding -->
   <ToolPosDetector DEF="haptic_tool_pos"
      enabled="TRUE"
   />

   <RealtimeAttractor DEF="attractor"
      point="0 0 0"
      radius="0.2"
      strength="8.0"
      deviceIndex="0"
      localEnabled="TRUE"
      remoteEnabled="TRUE"
      withOffset="FALSE"
   />

   <RemoteUDPServer DEF="udp_server"
      listeningPort="40002"
      open="TRUE"
      periodicSend="TRUE"
      periodicSendRate="800"
      simulatedLatency="0.000">
      <RemoteSFVec3f DEF="rf_haptic_toolpos"
         isHapticField="TRUE"
         fieldId="0">
      </RemoteSFVec3f>
   </RemoteUDPServer>

   <ROUTE fromNode="rf_haptic_toolpos" fromField="received"
          toNode="attractor" toField="realtimePoint"/>
   <ROUTE fromNode="haptic_tool_pos" fromField="localPos"
          toNode="rf_haptic_toolpos" toField="toBeSent"/>
   <ROUTE fromNode="button" fromField="received"
          toNode="attractor" toField="enabled"/>
   <ROUTE fromNode="HDEV" fromField="mainButton"
          toNode="attractor" toField="enabled"/>
```

Figure 7. H3D code required for hand guiding (server)

You will notice from Example2 that the **RemoteConnections** have the fields **periodicSend** and **periodicSendRate**. Normally a value is sent across the network only when it changes. If **periodicSend** is TRUE, a value is sent to the other end at the **periodicSendRate**, regardless of whether it has changed or not. In our case the tool tip position is being detected at about 1000Hz. We are setting the **periodicSendRate** at 800Hz (i.e. a lower rate than is available) because by experimentation it was found to still provide a smooth response, and would reduce the network traffic. UDP is a fairly “unsociable” protocol, in that it can monopolize a network if it floods it with data, to the detriment of other users. Although the haptic interaction typically uses a much lower bandwidth than, say, video, minimizing network traffic should always be a consideration.

There is also a **simulatedLatency** field. This can be used for testing when you are working with a low latency LAN but you eventually want your code to work on a WAN with significant latency. If non-zero, it simulates a latency by queuing all network data and releasing it at the correct rate after the specified latency. Be aware that it does not simulate any jitter, however, and the jitter on a WAN can also have an effect on object behaviour in a haptic scene. A WARNING – make sure you have reverted your **simulatedLatency** value to zero before deploying your system!

4. Remotely Connected Dynamic Objects

As mentioned in section 3.6., you may like to keep two networked representations of the same logical object to stay 'in sync' with each other so that they behave as one, single object. In that case, you can route the same fields of the same nodes in each scene together.

However, care must be taken when connecting equivalent fields of remote machines. If the local system has a mechanism determining the movement of an object (such as exists in the **DynamicTransform** nodes in the H3D API), it may conflict with any data coming in from a remote machine. In particular, if two machines are separately calculating the position of a DynamicTransform, and sending their calculated positions to each other, the objects may jitter between two positions on the display, as the delay (latency) in the network may cause a discrepancy between the value received and that calculated locally. This jitter can, in some cases, accelerate until the system becomes unstable and unusable.

The toolkit contains nodes which overcome this limitation under certain circumstances.

The solution used in the toolkit involves having *master* and *slave* dynamic nodes in the scene. The master can be any of the **GrabableDynamic** nodes (see below), and the slave is implemented as a **GrabableDynamic** with **slaveMode** set to TRUE.

Another requirement is that users push and pull objects around via virtual springs. In that way, a number of users can be attached to one object and co-operatively move it. Of course, the springs can be quite stiff, so that the feel is as if you are grabbing the object directly. The alternative, of routing a user's tool position directly to the object's position, fails when more than one user is interacting with it, because the movement can often be in opposition, and the object can jitter between the two alternatives.

The **Dynamic** node is an extension of the H3D::**DynamicTransform** node. It adds any user forces from the haptic tool to the node's motion, so that you can bump and push a Dynamic around in the scene. It also has an optional spring force anchoring the node to a point in space.

A **CollidableDynamic** adds a course-grained inter-object collision mechanism to this. The collision mechanism is turned off by default. Te granularity of the collisions can be set manually by the developer, or automatically by the system. Care should be taken when using the aut-generation mechanism, however. If it is set too fine, the number of collision cells can cause the frame rate to drop very low. This can, in turn, cause dynamic object instability.

The **GrabableDynamic** inherits from this and adds the ability to grasp and pull on objects as well push them with the haptic tool.

A **DampedDynamic** is a **GrabableDynamic** that has specialized physics that can accommodate latency. This is the best one to link up with a slave (a GrabableDynamic with Mode set to TRUE) to solve the stability problem mentioned above.

When in slave mode, it does not do any local calculating of its motion, it simply collects all forces acting upon it and sends them (across RemoteFields) to the master dynamic node on the other machine. The master node, collects its own local forces (e.g. local user input, gravity, springs or other forces) along with those it receives across the network, and uses the resultant force to calculate the new position of the object. It then renders this new position locally and also sends the new position back to the slave, for rendering there. The system is not perfect as the rendering still occurs on the systems at slightly different times (i.e. the network latency) but it has proved to solve the instability effect in many cases. This logic is contained in the dynamic movement nodes of the toolkit.

As well as this, experimentation discovered that the Newtonian physics model (i.e. the equations of motion involving force, acceleration, velocity, displacement) , interfered with the stability of the system when network latency is significant. The toolkit overcomes this with the DampedDynamic node by removing any momentum completely from the equations. This produces an unrealistic result – objects only move while forces act on them and immediately stop when the resultant force drops to zero. However, for many applications (especially those that don't involve projectiles), this is sufficient.

6.11. Testing a networked system on one machine

As we have seen, networked systems require two H3D programs, a server and a client, running, so that they can communicate. Often, when developing a program, it is inconvenient to use two separate computers. It is possible to run two H3D programs on one computer, with one haptics device (or even with no haptics devices) connected. To do so, you need to use the **MouseHapticsDevice** node, supplied with the toolkit. It simulates a haptic device with a computer mouse, the left button 'grabbing' the haptic proxy, while movement of the mouse moves it in the x and y direction. Movement in the z direction can be attained by also pressing the right mouse button. The middle mouse button corresponds to the mainButton of the haptic device. Note, however, that the motion is only a graphics rates, and has some jittering artefacts. So actual forces felt on a connected program with a real haptic device will be rough and not indicative of what you would feel in a normal setup.

The method of having one phantom and one mouse haptics device on the same machine is:

1. Have one of your haptic scenes (either server or client – for our example we will use server), configured with a MouseHapticDevice instead of the real one.
2. Open the H3D Settings GUI.
3. Choose Haptics device = None and Apply changes

4. Start the server
5. Choose Haptics device = Any Phantom device (or whatever you normally chose) and Apply changes
6. Start the client.

Then you should have mouse control on the server and your haptic device on the client.

7. Examples and test programs

The nodes in the toolkit have example X3D files associated with them. They are located in the examples subdirectory. These were used for unit testing, but may also be a valuable source of information on how to use the classes.

There is a table matching individual nodes to examples in the file, ExamplesList.txt.

The examples referred to in this document are in the doc subdirectory.

TroubleShooting

1. “Could not create <Some Node>. It does not exist in the H3DNodeDatabase. Check that you have either
   ```
   <ImportLibrary url="..\bin\H3DNetworkingUtils_vc9_d.dll"/>  or
   <ImportLibrary url="..\bin\H3DNetworkingUtils_vc9.dll"/>
   ```
   at the top of your x3d file and that the path is correct to find the dll.

2. A Client program does not connect to the corresponding server.
   a. Make sure that the server is running, that the IP address or hostname in the client matches the server’s IP or hostname and that the port numbers match.
   b. Check that firewalls do not prevent access (try ‘ping’ing the server from the client.)
3. “packet ID does not match any known field”.
   a. Check that the remoteField id’s match in both server and client. This message is generated when a remoteField on one of the machines has an ID that has no match on the other machine.
4. remoteFields linked together have strange, or no behaviour.
   a. Check that remoteField ID’s on client and server match. This problem can occur if a remoteField of one type is connected (via matching IDs) a remoteField of some different type (e.g. float to int).
5. “HL_DEVICE_ERROR(The operation could not be performed.)”.

a. You may be trying to run two H3D programs on the same machine, both trying to connect to the one haptic device. Change one of the programs to use the MouseHapticsDevice instead (see Running the examples).

Appendix

There is a misconception with networked haptics that it is possible to feel what another user is feeling, or experience what another user is experiencing in the real or virtual world. That is, if two users are haptically connected, when one user moves and interacts with an object (either real or a virtual object in the scene), the other user could feel what the experience is like. It has been said that skills could be taught or transferred in this way.

This is, in fact, not possible. When a person moves their hand towards an object, their muscles and tendons are tensing or stretching to move the hand forward, and their brain is commanding them to do so. When the hand collides with an object, a reactive force works against that forward motion. The understanding of what that object feels like is a combination of the knowledge of the muscles moving forward and the reactive force. With a haptic system, we can transmit that reactive force, but we cannot make the other's hand and muscles move in the same way as the originator was doing. Therefore the reactive force just arrives 'out of the blue' without any movement for it to work against. In the extreme case, the distant user just has their hand lying relaxed holding the haptic device when the force arrives. It would be a sudden jerk in a certain direction and would be meaningless.

A suggested alternative has been to use the haptic force to pull the distant user's hand in the same movement as the local user before the collision. However, in that case, with the same haptic device providing both the guiding force and the collision reactive force when the collision occurs, the two forces cancel each other out (that is why the local user's hand stops moving). So the resulting force to be transmitted to the other end would therefore be zero at the time of collision. So before collision there would be a dragging force towards the collision and at collision time, the force drops to zero. Obviously not what is desired.

The only possible way of doing it may be with two haptic devices, one perhaps joined to the far users forearm and the second one held normally in the hand. However, this setup would, I expect, still not be very convincing.