Networked Virtual Environments and Distributed Rendering

IGS HT 2003-09-24

Lars Pettersson

Background and history

Classification of multiplayer games

• Multiplayer
• Massively multiplayer (MMORPG)
• Peer-to-peer
• Client/server

Military simulations

• 1983 - 1990 SIMET (SIMulator NETworking)
• 1989 - … DIS (Distributed Interactive Simulation)

Communities, chats, concept visualization, e-Learning

Fundamentals

Host A

Host B

• Message may or may not arrive
• Message may arrive as it was sent or as something else (it might be garbled)
• Message may arrive 0, 1 or more times
• If there are several messages, they may arrive in a different order than in which they were sent

How can we do anything useful under these premises?
Fundamentals

We only need to worry about TCP and UDP!

how to use them, not how they work

Fundamentals

TCP - Connection oriented, stream oriented, reliable service

Two hosts can create a virtual stream and send data to each other. The data is guaranteed to arrive, and to arrive in the correct order.

UDP - Connectionless, datagram oriented, unreliable service

The data is divided into discrete datagrams. A datagram might be lost or arrive out of sequence in relation to other datagrams (we have to deal with this).

Multicast

A method for delivering the same data to several hosts at the same time. Works only for UDP. Not widely implemented across the Internet, but might be in the future...

Network layer

IP

Responsible for routing data between hosts on different networks

This is a best-effort service

Hosts are identified using IP-numbers, e.g. 130.238.162.197.
DNS translates between names and numbers, e.g. ulldb.ull.uu.se → 130.238.162.197

Access layer

ethernet, PPP etc

Responsible for getting ones and zeros into the cable and out at the other end between hosts on the same network

Fundamentals

Some parameters to take into consideration

ISDN line 64kbps -> Approx 6400 bytes / sec (due to overhead)
30 fps -> 6400/30 = 213 bytes each frame
10 fps -> 6400/10 = 640 bytes each frame

Toolkits for Networked virtual environments

Which issues should a Toolkit deal with?

• State sharing (shared memory)
• Mutual exclusion of resources
• Data serialization
• Dead reckoning

TCP / IP

UDP / IP

Ports are used to identify different applications on a given host, e.g. port 80 are used for http, port 21 for ftp.
Networking paradigms

Different ways to abstract out the actual sending and receiving of bits

Stream-based, like a ‘pipe’
Stream is contiguous or discrete
Both TCP and UDP fit here

Request-Response
Like making a function call
RPC, CORBA, COM, HTTP fits here

Shared data
Sharing a common set of data
SQL databases
Persistent information

The shared data paradigm is useful for NVE development

Networking paradigms

Network applications can be classified as peer to peer or client-server.

Both have advantages and disadvantages.
Distributed systems research community favor the peer to peer model.

Achieving mutual exclusion

Mutual exclusion - a centralized approach

Potential problem: The coordinator (server) crashes

Achieving mutual exclusion

What if the coordinator crashes?
Other methods have been proposed ...

Distributed
Ask everyone if it’s OK.

Token ring
A token is passed around a ring.
Only the host holding the token may do critical updates.

Ops… these algorithms fail if any participant crashes, however this is detectable using timeouts.

Achieving mutual exclusion

Mutual exclusion - comparison of methods

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Messages per entry / exit</th>
<th>Delay before entry (in message times)</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>3</td>
<td>2 * (n-1)</td>
<td>Coordinator crashes</td>
</tr>
<tr>
<td>Distributed</td>
<td>2 * (n-1)</td>
<td>2 * (n-1)</td>
<td>Any participant crashes</td>
</tr>
<tr>
<td>Token ring</td>
<td>2 * (n-1)</td>
<td>2 * (n-1)</td>
<td>Any participant crashes</td>
</tr>
</tbody>
</table>

Distributed shared memory

• All participating hosts share a single “memory”
• Transparency important
• Ideally no more difficult to program than a single computer
• Can also be called
  • shared state repository
  • shared variables
  • shared database
Distributed shared memory

Distributed shared memory - consistency models

- **Strict consistency**
  - The distribution is totally transparent.
  - The shared memory acts exactly like local memory.

- **Weak consistency**
  - Explicitly tell the system when we want to use the memory, and when we are done.
  - When we want to use the memory, request contents from other hosts.
  - No need to send updates to others unless they request it.

- **Release consistency**
  - Send updates to others when we are done using the memory.
  - No need to send requests.

Distributed shared memory - implementation issues

- Strict consistency is very difficult to achieve
  - We have to detect each variable read or write (implement our own programming language)
  - Or use paging like virtual memory swapping (implement our own operating system)

- Entry or release consistency is a more realistic approach
  - Entry consistency if we have more writes than reads.
  - Release consistency if we have more reads than writes.

  Requires a method to achieve mutual exclusion

Traditional shared memory systems

- Often developed with scientific calculations in mind
- Transparency and consistency very important!
- Uses special preprocessors or programming languages

Data Serialization

Data serialization

Different architectures use different byte ordering, such as Intel and Motorola.

This should be handled as transparently as possible by the system.

Different approaches

- Explicitly pack each primitive (float, integer, etc)
- Provide structure description at runtime
- Use precompiler to create special serialization code

Networked virtual environments - dead reckoning

A method for approximating an object’s position

Requires less frequent state updates

Simplest approach:

Time 3:
Position (4,3)
Velocity (3,2)

Predict new position using
\[ p_{new} = p_{0} + (t - t_{0}) \cdot v_{0} \]

Time 4:
Position (7,5)

Predict new position using
\[ p_{new} = p_{1} + (t - t_{1}) \cdot v_{1} \]

Using a blending-factor \( b \) and a convergence-period \( c \)

\[ p_{new} = (1 - b) \cdot (p_{0} + (t - t_{0}) \cdot v_{0}) + b \cdot (p_{1} + (t - t_{1}) \cdot v_{1}) \]

\[ b = \frac{t - t_{1}}{c} \]

\[ b = 1 \] if \( t \geq t_{1} + c \)

The object will converge from the earlier path into the new one in time \( c \).

Drawback: introduces extra latency! (by time \( c \))
Networked virtual environments - dead reckoning

Better prediction
Use higher order polynomials

\[ \text{Predict new position using} \]
\[ P_{\text{pred}} = P_0 + (t-t_0)v_0 + (t-t_0)^2a_0 \]

Higher order polynomials can also be used for convergence
As a rule of thumb 10 updates/sec is enough for 60-70 fps

Networked virtual environments - application design

Should the logic reside in the clients or in the server?
Server
- The clients can be made more general
- Easier to guarantee consistency

Clients
- The server can be more general
- Communicates at a higher abstraction level (usually less traffic)

Should we use a reliable or an unreliable protocol?
Unreliable protocol requires frequent updates
Reliable protocol propagates slower and is less efficient

When should data be sent?
Everything always → Only when something changes

Hybrid approaches:
Use UDP, but send the whole state using TCP once a second.
Use UDP for things that move, but TCP when it stops.

What data should be shared?
- Transformations
- Geometries
- Textures
- Materials

<table>
<thead>
<tr>
<th>General / specialized client</th>
<th>Reliable / unreliable protocol</th>
<th>Peer to peer / Client - server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quake</td>
<td>Specialized</td>
<td>Unreliable</td>
</tr>
<tr>
<td>DIS, NetWIDE</td>
<td>Specialized</td>
<td>Unreliable</td>
</tr>
<tr>
<td>DIVX</td>
<td>General</td>
<td>Reliable</td>
</tr>
<tr>
<td>ActiveWorld</td>
<td>General</td>
<td>Both</td>
</tr>
</tbody>
</table>
Java Shared Data Toolkit

Uses three abstractions for data communication

- **ByteArrays**: Allows the programmer to create shared memory areas that are kept consistent.
- **Tokens**: Can be grabbed and released, for mutual exclusion of resources.
- **Channels**: For stream-based communication.


CavernSOFT

Communication mainly streambased

Uses concept of ‘reflectors’ for group communication

TCP and UDP reflectors available

Provides higher-level database classes for state-sharing based on the reflector concept.


Streep (State Repository)

- Based on shared semi-transparent memory
- Memory is allocated using functions resembling standard C functions.
- Updates are explicitly sent using ‘channels’
- Provides function for locking memory blocks
- Can automatically serialize memory given a description string
- Uses TCP, UDP or multicast

Server:
```c
SR *sr;
int *i;
sr=srCreateServer(1200, "udp tcp");
i=srAllocMem(sr, "test", sizeof(int), SR_NONE);
while (1)
{
    *i=*i+1;
srNotify(sr, i, 0);
srProcess(sr, 100, 100);
}
```

Client:
```c
SR *sr;
int *i;
sr=srCreateClient("localhost", 1200);
i=srGetMem(sr, "test", SR_NONE);
srChannelSubscribe(sr,0);
while (1)
{
    printf("%d\n",*i);
srProcess(sr, 100, 100);
}
```

VRT

Functions for sharing node transformations and on/off state exist in VRT

```c
extern int VRT_NetConnect(int port);
extern int VRT_NetDisconnect();
extern VRT_Node *VRT_SharedNodeNew(VRT_Node *parent, char *name);
```

**VRT_NetConnect**

Searches to LAN for a server running the specified port. If no server exists one will be created. The first one to start will be the server.

**VRT_SharedNodeNew**

All nodes with the same name on the network will share their transformations and their switch state. Functions such as VRT_NodeSetTranslation, VRT_NodeSetRotation, VRT_SwitchOn will send information to others.