The TRIBES Engine Networking Model

or How to Make the Internet Rock for Multi-player Games

by

Mark Frohnmayer and Tim Gift
Mark.Frohnmayer@Dynamix.com Tim.Gift@Dynamix.com

Abstract

This paper discusses the networking model developed to support a "real-time" multi-player gaming environment. This model is being developed for TRIBES II, and was first implemented in Starsiege TRIBES, a multi-player online team game published in December '98. The three major features of this model are: support for multiple data delivery requirements, partial object state updates and a packet delivery notification protocol.

Overview

Starsiege TRIBES supports two modes of play: single player or multi-player over a LAN or the Internet. The multi-player mode supports up to 128 human or AI controlled players in a single game. Performance over the Internet drove the design of the networking model. The model supports low end modem connections and is designed to deal with low bandwidth, high latency and intermittent packet loss.

The model deals primarily with the delivery of data and a key concept is the classification of delivery requirements. All data is classified into one of several requirement categories and the design of each component in the model centers around meeting those requirements. We organize transmitted data as follows:

1. ***Non-guaranteed data is data that is never re-transmitted if lost.***
2. ***Guaranteed data is data that must be retransmitted if lost, and delivered to the client in the order it was sent.***
3. ***Most Recent State data is volatile data of which only the latest version is of interest.***

4. ***Guaranteed Quickest data is data that needs to be delivered in the quickest possible manner.***

The networking model is divided into three major components as shown in Figure 1:

1. ***A Connection Layer that deals with notification and delivery of packets between client and server. The features of this layer along with a stream class provide the general infrastructure on which the other layers are built.***

2. ***A Stream Layer which provides packet stream management. This layer employs five stream managers to deal with events, object mirroring, input move management, static data and string compression. Each of the five stream managers provides different data delivery guarantees.***

3. ***A Simulation Layer which manages all objects in the simulation. A full description of this layer is outside the scope of this article but several items are relevant: the advancement of time, object scoping and client prediction.***

Though Starsiege TRIBES employs a client-server connection model, only a few of the stream managers are asymmetrical. Nothing prevents this model from being used in peer-to-peer or multi-server architectures.

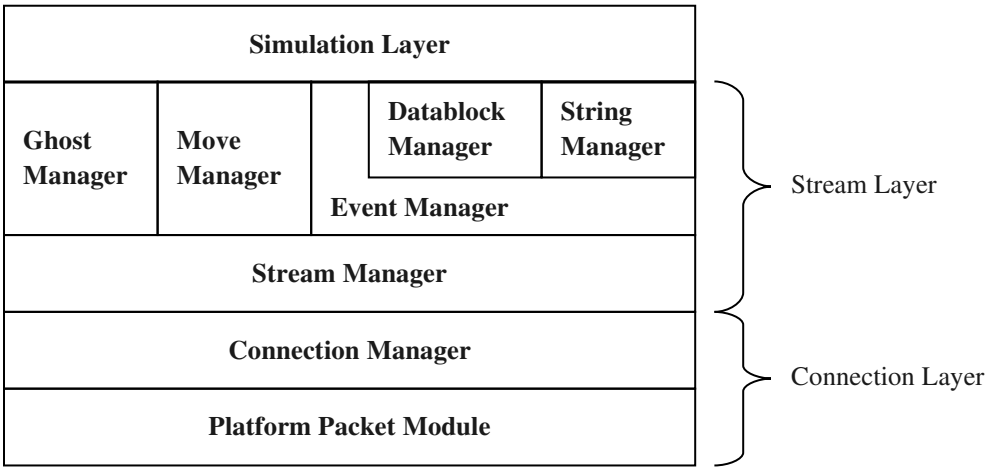

Figure 1

Persistent Objects

Though polymorphic object persistence is not an integral part of the networking model, it is a feature that is used extensively. A large percentage of data transmission is done using persistent objects and the polymorphic nature of the objects used to hide the type and content of the data from the networking code.

Each class that is declared persistent registers a "class representative" (ClassRep) object with the Persist manager and is assigned a unique Class ID. The ClassRep is used by the Persist manager to construct objects of the type it represents. By mapping assigned Class IDs to ClassReps the Persist manager can construct any persistent object given its class ID. This construction by Class ID along with virtual read and write methods provides basic polymorphic IO.

Writing an object into a stream is a two step process. First its class ID is written, then the object's virtual write method is invoked. Reading an object involves reading the Class ID, constructing the object using the Persist manager, and then invoking its virtual read method. In this way both the object's class, and the data it reads and writes are hidden from the stream managers.

Connection Layer

The Connection layer provides transmission of packets between host machines and is divided into two modules, a platform packet module and the Connection manager. The platform packet module provides basic connectionless unreliable packet delivery, currently implemented using standard UDP sockets. The Connection manager provides a virtual connection between two hosts and while it does not provide delivery guarantees, it does provide packet delivery status notifications.

This notification guarantee is very important to the architecture; a packet layer that supports only guaranteed or non-guaranteed packets would not be sufficient to support the four basic data delivery modalities outlined above. How each delivery mode is handled is delegated to a higher level -- the connection manager only guarantees the correct notification of a sent packet's status. I.e., if a packet is notified as dropped it was either dropped or delivered out of order (and subsequently dropped), and if a packet is notified as delivered, it has been delivered.

The Connection manager notifies the Stream layer of the status of each packet in the order that they were sent. An overview of this relationship is shown in Figure 2. Dropped packets are never re-transmitted by the Connection manager; the Stream layer, and its associated managers, handle all data guarantee mechanisms. Since packets are never retransmitted, they are freed immediately after transmission.

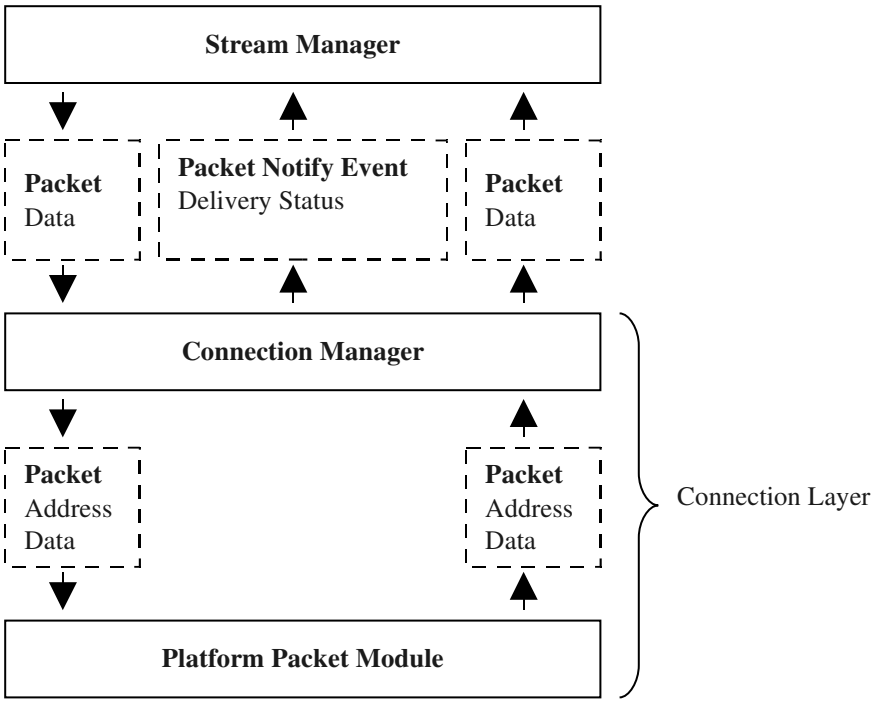

Figure 2

The Connection manager employs a sliding window protocol in order to track the delivery of packets. When the window is full, transmission stops until an Acknowledgment is received. Acknowledgment of packets is only used to advance the window and generate notification events. This protocol imposes an average overhead of 3 bytes per packet.

Though not part of the protocol, an important feature of the architecture is bit-packing provided by a custom bit stream class. This class provides bit level read and write functions, including read/write functions for: a single bit, variable length integers, variable length normalized floats, and Huffman compressed strings. All packet data, including the header and the sliding window protocol, are accessed through this stream. These packing features are used extensively and virtually all data is transmitted using the smallest number of bits possible.

Examples of bit packing using the bit stream.

Writing into the packet:

```
if (stream->writeBool(updateDamage)) {    // Uses 1 bit
      stream->writeInt(mDamageState, 2);  // Uses 2 bits
      if (mDamageState != Dead)
            stream->writeInt(mDamageLevel,6);
      if (stream->writeBool(mRepairActive))
```

```
                stream->writeInt(mRepairRate,4);
        }
```

The matching read method:

```
        if (stream->readBool()) {
                mDamageState = stream->readInt(2);
                if (mDamageState != Dead)
                        mDamageLevel = stream->readInt(6);
                mRepairActive = stream->readBool();
                if (mRepairActive)
                        mRepairRate = stream->readInt(4);
        }
```

Stream Layer

The Stream layer is comprised of a Stream manager and the Event, Ghost, Move, Datablock and String managers. The Connection manager deals with the frequency of packet transmission as well as packet size and stream manager ordering. The individual stream managers deal with the packing and delivering of data including the various guarantee mechanisms. The String manager, dealing primarily with the compression of string data, is not covered in this article.

The Stream manager allocates and transmits packets to its counterpart on a remote host. To control bandwidth, each Stream manager has a packet update rate and size. These parameters are set by the remote host's Stream manager and represent the amount of data it's capable of receiving. Clients connecting to a dedicated server set these values to represent the bandwidth of their Internet connection. If a client is connecting to an ISP with a 28.8 modem then it could set a rate of 10 packets per second with a size of 200 to produce about 2K of data per second. The client may change these parameters on the fly in response to changes in line quality. To control its own out-going bandwidth, the server imposes a maximum bandwidth per client based on its own connection quality.

Packets are allocated by the Stream manager and filled by the Move, Event, and Ghost managers. Since the opportunity to write into the stream is given to the managers in a fixed order, this order forms a fixed priority among them. Once the requested packet size is exceeded or all the managers are done, the packet is handed off to the Connection manager for transmission. Figure 3 shows an example packet.

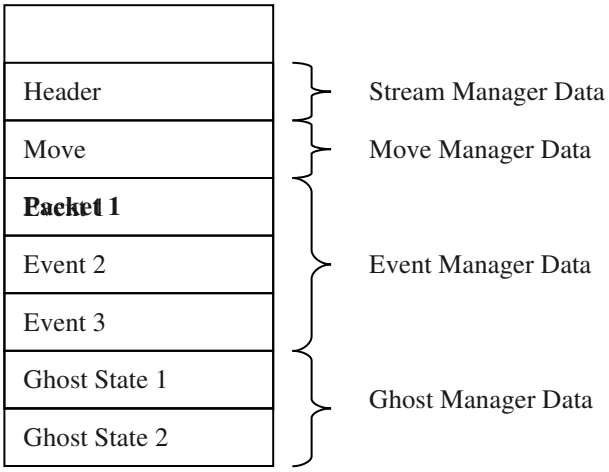

Figure 3: Example Packet

When a packet is delivered to the Stream layer for reading, it simply hands off the packet to each stream manager in the same fixed order and each manager is responsible for reading the data written by its counterpart.

The Stream manager also provides a Transmission Record for each packet that it constructs (shown in Figure 4). When a stream manager stores data into a packet, it stores information regarding that data in the Transmission Record. When a Connection manager Notify Event occurs for a packet, the Transmission Record for that packet is processed by the Stream, Event and Ghost managers, and is used to provide delivery guarantees. Since the Connection manager always produces Notify Events for each packet in the order they were sent, the Transmission Records are stored in a simple FIFO.

Event Stream Manager

The Event Stream manager is responsible for providing guaranteed and non-guaranteed delivery of event objects from one host to another. Guaranteed events are also guaranteed to process in the order they were sent. A sliding window is used to track the status of events. All window management is performed using the Transmission Records.

When the Event manager is given a packet to write into, it pops events off its outgoing queue and writes them into the stream until either the packet size is exceeded, the queue is empty, or the event window is full. Events are persistent objects and are written using the methods discussed earlier.

Once events are written into a stream, they are linked together and attached to the Stream manager's Transmission Record. When

the Stream manager is notified of a packet's status, the Transmission Record for that packet is passed to the Event manager. If the Connection manager signals the successful delivery of the packet then the events attached to the record are deleted and the sliding window updated. If the Connection manager signals non-delivery then the events associated with the lost packet are simply pushed onto the head of the event queue for re-transmission.

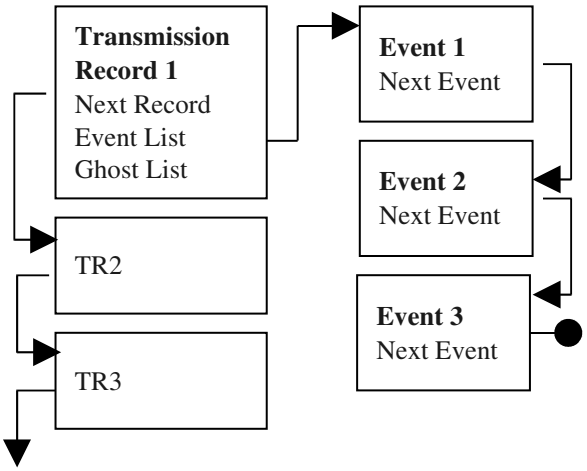

Figure 4: Transmission queue including record with events

When the manager is given a packet stream to read, it unpacks and constructs all the events. If the event is marked as guaranteed, it is added to an ordered queue used to provide ordered processing. Processing either happens immediately upon receipt (for non-guaranteed events) or as the ordered queue is advanced.

Since neither the transmitting nor the receiving Event managers store delivery status information into the packet stream, this protocol imposes very little overhead above that already imposed by the Connection manager, normally 3 bits per packet and 1 bit per event. If a packet is dropped an additional 7-14 bits per packet may be written.

In packing, delivering and guaranteeing Event objects, the Event manager provides a fundamental service used by many other subsystems in the TRIBES II engine. Since events are packed, unpacked and processed using virtual functions and the Persist manager, the Event manager itself has no knowledge of the type or contents of these events.

An example of a simple event:

```
class Signal: public Event {
      enum {
```

```
            SignalBits = 4,
      };
      U8 signal;
public:
      Signal(U8 s) {
            guaranteed = true;
            signal = s;
      }
      void packData(Bitstream* stream) {
            stream->writeInt(signal,SignalBits);
      }
      void unpackData(Bitsream* stream){
            signal = stream->readInt(SignalBits);
      }
      void process(NetConnection* ps){
            printf("Recieved signal %d from host %s",
                  signal,ps->getObjectName());
      }
};
```

Ghost Stream Manager

The Ghost manager provides two key functions: the "ghosting" of objects from one host to another, and the transferring of state information between the original object and its ghost. A ghost is a copy of an object persisted and transmitted to a remote host. An object may only have one ghost per Ghost manager (and thus per remote host), but may be ghosted by several Ghost managers at once (to different clients, for example). Ghosts are created using a form of guaranteed delivery also used to support partial state updates between an object and its ghost. State data is considered volatile and transferred using a "Most Recent State" algorithm.

Since ghosting an object involves network overhead, the ghost manager does not ghost all objects in the simulation, but instead has a concept of "scope." Objects may come in and out of scope for a manager for a number of different reasons (this process is managed in the Simulation layer). When an object comes into scope, its ghost is transferred to the remote host; when an object goes out of scope its ghost is deleted. While an object is in scope, state data is transferred between the object and its ghost and is updated at a rate based on its priority and state mask. The manager also supports the notion of "ghost always" objects, which are always in scope.

When a new object comes into scope, the object is tagged with a Ghost Record containing a Ghost ID and a State Mask. The Ghost ID is assigned from a limited range and is used by the local manager to address ghost objects on the remote host. The remote Ghost manager maintains a dictionary, which translates Ghost IDs into local ghost objects. When the manager transfers information from an object to its ghost, its Ghost ID is embedded in the stream so that the remote manager can properly deliver the data. The Ghost

Record's State Mask represents state data that an object is interested in transferring and is the heart of the "Most Recent State" algorithm. Each bit in the State Mask represents a class dependant set of related data, or state, that will be transferred. An object may represent its position and velocity as one state bit, changes in rotation as another, and possibly a change in animation state as a third. (TRIBES simulation objects typically have upwards of 20 state flags.) Each state is tracked and transferred independently of the others, providing the ability to perform partial updates of an object's total state.

When the ghost manager is given a packet to fill by the stream manager, it performs two basic actions. First, it builds an update list which includes every object with a status change or non-zero State Mask. This update list is ordered first by status change, then by object priority. Status changes include the transferring or deleting of ghosts from the remote host; an object's priority is a value assigned by the Simulation layer as part of the scoping process. Next, the update list is traversed in order writing the Ghost ID, status and object state information into the packet until the packet is filled or the list is empty. For each object that contains data in the stream, a transmission structure is constructed containing the status change requested and the State Mask. This structure is attached to the Transmission Record for the packet as shown in Figure 5. The State Mask bits in the Transmission structure represent the state data written into the stream by the object.

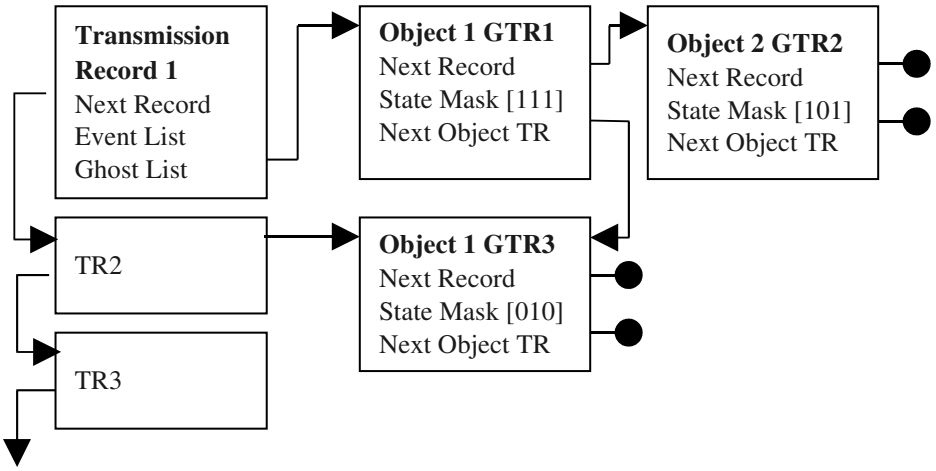

Figure 5: Transmission queue including ghost transmission records for objects 1 & 2

When a ghost manager receives a packet, it reads each set of Ghost ID and status flags in order. If a status indicates a new ghost, the Persist manager is called to construct the object and the new ghost is entered into the Ghost ID dictionary. If the status change is a deletion request, the ghost object is obtained

from the dictionary and deleted. If there is no status change, the packet contains only state data. The ghost object is obtained from the ID dictionary and its unpack method is called with the current packet bit stream. Which state mask bits were used by the source object to pack data is not transmitted to the ghost. The object is responsible for encoding such information into the bit stream.

The guarantee of object state information happens as follows; State Mask bits represent state changes, so if an object's position changes, it sets its "position state" bit in the State Mask. If there are multiple managers ghosting this object, the bit is set in the Ghost Record related to each manager. This set bit now represents state data that the Ghost manager needs to transfer to the object's ghost. When the manager is filling a packet, objects with a non-zero State Masks are asked to write into the packet stream given the current mask for that Ghost manager. In the example of the "position state", the object writes its current position. At this point the State Mask represents which states were written into the packet, and the mask is stored in a transmission structure which is linked to the Transmission Record. Once this is done, the State Mask is cleared. If the packet is notified as lost, then the State Mask for each object which had data in that packet is updated to include lost bits. State bits are only considered lost if no subsequently-sent packet contained the same state bit for that object. Since we guarantee only the latest state, if a later packet has already been sent with a later version of that state, then this requirement has been met and no other action takes place.

In Figure 5, if the packet for Transmission Record 1 is lost then bits [101] for object 1 and 2 are considered lost; whereas bit [010] for object 1 is not, because GTR 3, a later update, has that bit set. If a lost bit is added to an object's State Mask, then that state will get re-transmitted at the next opportunity. Status changes, such as object construction and deletion, are handled in a similar fashion.

Example of state masks:

> *An object's position changes and it sets its Position State bit to true. The ghost manager decides to pack the object's state and calls the object's pack method with the state mask. The object sees that the manager wants to transmit its position and writes its current position into the packet bit stream. The manager stores the state mask containing the Position State bit in the notification structure for that packet and the State Mask is then cleared.*
>
> *First possibility: the packet is delivered and the state has been successfully transferred. The object's position has not changed since the packet was sent and the State Mask is still 0 so no further action takes place.*

Second possibility: the packet is not delivered and in the meantime the object's position has not changed. The packet is notified as lost. Since no packets have been sent with new Position State data for the object, the Position State bit is set in the object's State Mask. In this case the same position will get resent the next time the object is packed.

Third possibility: the packet is not delivered, but in the meantime, the object's position has changed. Before notification of non-delivery, another packet was already sent with the latest position. In this case, the manager knows that Position State data has already been retransmitted and the loss of the state is ignored. This is possible because the Ghost manager only guarantees that the latest version of the state data is transferred and not any particular version or copy of that state. In this example, this is exactly the behavior we want -- the position information contained in the lost packet is stale and we don't want to retransmit it. Unless the second packet is notified as lost, we also don't want to re-transmit the current position, as it hasn't changed since the second packet was sent.

Fourth possibility: the packet is not delivered, and in the meantime, the object's position has changed. But the position change was recent and no second packet has been sent yet. Since no new packets have been sent with Position State data for the object, the object's Position State bit is set by the Ghost manager. The object will have already set the bit itself, so this will not change the object's behavior. The next time the object is packed it will write its current position, not the position sent in the lost packet. Similar to the third possibility, the position data lost in the dropped packet is never re-sent.

Like the Connection manager, Event, and other stream managers, a server will have a Ghost manager for each client connected to it. If there are several Ghost managers ghosting an object, the object will have a Ghost Record for each manager. Each manager will have a different Ghost ID and State Mask for that object representing the state of that object on its remote host. When an object sets one of its state bits, it updates the State Mask for every Ghost manager currently ghosting it and each manager tracks and updates state information independently.

This process of tracking state masks and guaranteeing the delivery of state information between object and their ghosts is a key feature of TRIBES networking. Game simulation objects rely almost exclusively on this mechanism to transfer information.

Move Stream Manager

The Move manager guarantees "soonest possible" delivery of client input moves to the server and "soonest possible" delivery of control object state data. A sliding window is used to track the delivery of moves and also to synchronize move processing between control objects and their ghosts. This manager is asymmetrical in that moves are only sent from a "client" connection to a "server" connection and Control Object state data from server object to client ghost.

Input moves are used to control simulation objects such as vehicles, cameras, and players. Input is gathered on the client by an Input Manager, which collects a move every 32 milliseconds. Moves consist of x, y and z translations, yaw, pitch and roll rotations as well as an array of trigger states. These moves are delivered to objects by the Move manager and are the sole means of user controlled object movement.

The Move manager, similar to the Event and Ghost managers, writes information into every packet stream allocated by the Stream manager. Unlike the other managers, it makes no use of the Transmission Record or the Connection manager's notification system. Instead, the Move manager provides "soonest possible" delivery of moves to the server by writing moves into every packet sent. Each move is transmitted three times in consecutive packets. Each packet transmitted from the server contains the last move received, which the client uses to advance a sliding window of outstanding moves. If the window becomes full, the client simulation is halted until the server advances the window. If 3 packets are dropped in a row then any moves unique to those packets are lost.

In Starsiege TRIBES, every move in the current window was transmitted in every packet guaranteeing 100% delivery of all moves, but this was sometimes the cause of a negative feedback loop. A dropped packet would increase the size of the next packet by widening the sliding window and causing more moves to be transmitted. On low bandwidth Internet connections the increased packet size could exceed the connection's bandwidth, causing more packets to be dropped. Though this means moves are no longer guaranteed delivery, the client responds better when large numbers of packets are dropped. Losing moves can cause the client and server control object to get out of sync which may produce a momentary "warping" for the client.

The Control Object for a client is the object that receives the current move being produced. Both the Control Object on the server and its ghost on the client receive every move. Control Objects have two major requirements that differentiate them from other objects. The first is the deterministic processing of moves. Both the original server object and its ghost are given moves generated by the player and both, given the same starting state, must produce the same end state. The second is that the control object must be able to transfer its current state on

demand. Though Control Objects are ghosted, and thus can transfer information using the ghost State Mask, the Move manager provides a separate "Soonest possible" delivery of a Control Object's state to its ghost. This state information is similar but not identical to information transmitted through the Ghost manager. State information sent to a ghost is normally limited to that needed to render the object, or to reasonably predict its motion. Control Object state information must include everything needed to deterministically process input moves. "Soonest possible" delivery of this control state information is achieved by having the Control Object write its current state into every packet sent from the server. This combination of state transfer and deterministic move processing is used to keep the Control Object and its ghost synchronized, and to allow the client to predict the motion of the control object ahead of server updates.

Delivering moves and Control Object state updates as soon as possible along with client prediction of move processing allows smooth and immediate response to player input, while providing full validation of all player moves by the server.

Datablock Stream Manager

The Datablock manager provides "latest state" delivery of datablocks from the server to the client. Datablocks are objects which contain relatively static data and the manager guarantees delivery only of the latest state at a very low frequency. Like the Ghost manager, the Datablock manager is asymmetric, and datablocks are only transmitted from server to client. The most common use of datablocks is to transmit initialization or reference data for ghosts.

Datablocks are copied to the client in a stream of guaranteed events. Since datablocks are considered static they are not updated during the normal course of a session, but only at fixed times such as when first connecting to a server or during a map change. A global "last modified" key is used to track changes to datablocks. Each time a datablock is modified it is assigned the current value of this key, after which the key is incremented. The manager keeps the highest key value transmitted to its remote host. Since the manager only guarantees "latest state", if a Datablock is modified several times between updates, this information is lost to the client. Further, since the changes to datablocks are low in frequency, the Ghost manager's overhead of partial state updates is not necessary -- a changed datablock is retransmitted in its entirety.

Unlike the other Connection stream managers, the Datablock manager does not write into, or read from the packet bit stream produced by the Stream manager. Updates are invoked directly from the scripting language and involve looping through all the existing datablocks and transmitting every datablock with a modified key greater than manager's key. The actual transfer of datablocks is performed by sending guaranteed Datablock Events

using the Event manager. A Datablock Event contains the datablock object to transmit, and the datablock is packed as part of the event.

As an example, the Vehicle class has an associated VehicleData Datablock class. The Datablock class contains numerous vehicle property attributes such as shape, maximum speed, maximum acceleration, suspension properties for wheels, etc. When a Vehicle object is constructed, it is assigned an instance of the VehicleData Datablock. Each instance of the VehicleData Datablock represents a different set of attributes describing a different type of vehicle such as a tank, scout, transport, etc. The Vehicle class itself represents the instance of that vehicle type in the simulation and provides all the code and dynamic attributes necessary to simulate the vehicle.

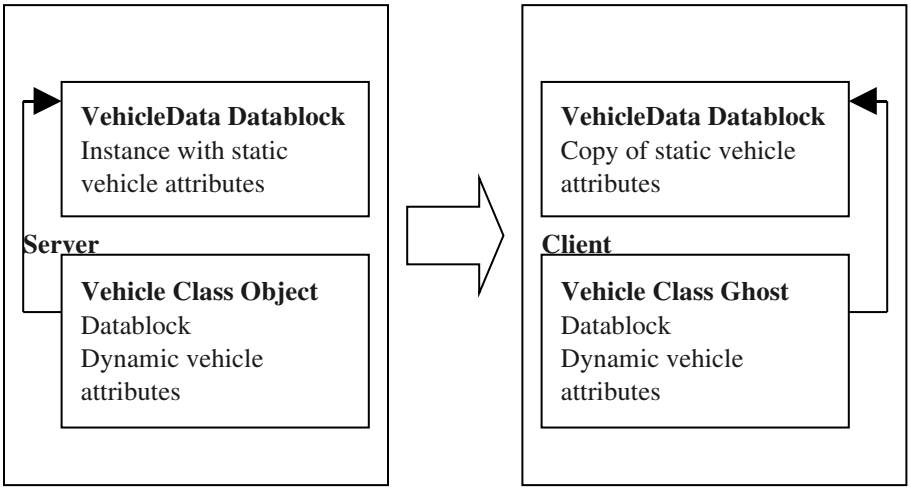

Figure 5: Vehicle Datablock relationship mirrored on client

The advantage of this separation of static attribute data and instance data is that Datablocks are normally only transmitted to the client once at connect time. When a Vehicle object is ghosted to the client (a process that can happen often) it sends only its datablock's ID along with other dynamic data relevant to the ghost. This results in huge savings in bandwidth, while still providing the flexibility of defining static attribute data on the server. As shown in Figure 5, the client will mirror the same relationship of Vehicle instance to VehicleData datablock.

Since datablocks are only transferred at times when the simulation is effectively stopped, more bandwidth is available and datablocks can contain more information than would normally be possible. This also means the Datablock manager imposes no extra overhead to the bitstream during the normal course of simulation.

Simulation Layer

The Simulation layer performs a number of functions of which only a few are relevant to a discussion of the networking model. These are the advancement of time, scoping for the Ghost manager, and client side prediction.

Advancement of simulation time is centered on the processing of moves. Moves are gathered every 32 milliseconds, and the simulation is advanced in 32 millisecond increments with a move for each increment or tick. All objects on the client and server are advanced in the same manner and if a Move manager is not controlling an object, then it is passed NULL moves.

Though client objects are advanced in fixed increments, the elapsed time between frames may not fall on even increments. To present a smooth view, especially on a machine with a high frame rate, time is always advanced to the beginning of the next interval, and ghost objects maintain backwards interpolation information which is used to "tween" frames between ticks. Objects on the server are never rendered so they do not have to perform this function.

Simulation objects must be scoped and prioritized for the ghost manager. Scoping is the process of determining which objects on the server are relevant to a particular client. In the simplest sense, this means all objects that are potentially visible to a client from that client's current control object. This form of scoping is performed using a spatial database maintained by the simulation. When an object is determined to be in scope, the object itself determines its own priority based on information such as current ghost state mask, distance from the scoping object, projected radius (using the scoping object's view frustrum parameters), relative velocity, animation state and interest modifiers (projectiles that are moving towards the client are more interesting than vehicles, vehicles are more interesting than items, etc.).

There are two forms of client side prediction in this model. First is the Control Object move prediction performed by the Move manager. This is by far the more complicated and provides the foundation for controlling and predicting objects such as players, vehicles, turrets and cameras. The other is prediction performed by non-controlled ghost objects such as projectiles, items, etc. This level of prediction is left entirely to the objects themselves and no supporting structure is provided.

An example of a prediction strategy is that provided by the Item class, one of the simpler objects. This object is driven entirely by the physics of the environment and does not process input moves. By simply simulating normally the ghost will correctly predict interaction with all static objects. If a dynamic object is hit on the server, then its ghost state mask is set to transfer its latest position and velocity. When the

client ghost receives an update from the server, it interpolates its current position smoothly to the server's updated position and continues from there. More complex strategies are used by the vehicle and player classes, which have to deal with predicting movement from other clients.

Summary

By classifying data according to its delivery requirements and organizing the stream managers to efficiently meet those requirements, the Tribes II networking model attempts to make optimal use of available bandwidth. In combination with dynamic object scoping and prioritization of partial object state updates, the TRIBES II engine allows smooth visualization of large virtual environments over low bandwidth connections, even when a substantial number of objects are visible to a client.

Though the model provides a solid foundation for network traffic, it is not sufficient to simply provide a fast reliable connection between hosts; there will always be bandwidth and latency issues to contend with. Not covered in this article are: choices made when deciding which object behavior should be managed by the server and which by the clients, which object state information to transfer, how to organize object partial states, how to prioritize object state changes relative to each other, which compression algorithm to use, and how to perform client side prediction of Control Objects. All of these can have a dramatic effect on perceived network performance.

This model was first implemented in Starsiege TRIBES, which shipped in December of '98. The updated implementation in TRIBES II has the following improvements:

1. *Reorganization and simplification of the Connection and Stream layers.*
2. *Better packet header compression for the Connection manager's notification protocol.*
3. *A reduction in the number of times moves are re-transmitted to avoid negative feedback loops due to increased packet size.*
4. *Better data compression by individual objects.*
5. *The addition of the String manager, which provides high order compression for repeatedly transmitted character strings (not discussed in this article).*

Several modifications to the model have been considered, and though they will not be implemented for TRIBES II, are worth exploring. These include:

1. *Variable production of packets by the Stream manager. Updates are currently produced at a constant rate dictated by the Stream manager's bandwidth settings. Packets could be produced as soon as data is available for transmission or data priority reaches a given threshold.*
2. *Asynchronous Connection processing on the server. All*

Connection managers are currently processed at the same time on the server in a single thread. The server architecture could be changed to support a primary simulation thread with a separate thread for each client. This would provide the Connection and Stream managers with lower latency communication with their remote host as well as provide better support for multi-processor hardware.

3. *The addition of new stream managers, including a multiplexing audio stream manager. TRIBES II will support voice using a simpler approach.*
4. *Dynamic allocation of server bandwidth. Bandwidth on the server is currently set at a fixed per client. The server could be modified to adjust its per client bandwidth based on the number of connected clients.*
5. *Determine a method of automatically and effectively calculating the available bandwidth on a connection. Bandwidth settings are currently set manually by a client.*

Acknowledgments

Though several of the ideas presented here are unique to TRIBES, others have been assimilated from other sources over countless years of reading and the authors, being too lazy to research their sources, would like to thank everyone in one sentence.