RZ 3441 (# 93732) 08/12/02
Computer Science 9 pages

Research Report

Building Infrastructure for Very Large Multi-Player Games

Sean Rooney, Daniel Bauer and Paolo Scotton

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

Building Infrastructure for Very Large Multi-Player Games

Sean Rooney
sro@zurich.ibm.com

Daniel Bauer
dnb@zurich.ibm.com

Paolo Scotton
psc@zurich.ibm.com

IBM Research
Zurich Research Laboratory
Säumerstrasse 4
8803 Rüschlikon, Switzerland

ABSTRACT

Massive multi-players games are interesting from a network research point of view as they are sensitive to loss and delay while at same time producing huge amounts of data. Their characteristics are very different from the request/response protocols that currently dominate the Internet. It is perhaps timely to consider the required infrastructure for future games that are one or two orders of magnitude larger than those currently available. Typically, existing large games use a central server model as peer-to-peer systems are assumed to be unscalable. However, central servers cannot scale beyond a certain point owing to I/O limitations. This paper proposes a hybrid solution between the peer-to-peer and central server models that, we claim, is more scalable than either alone.

1. INTRODUCTION

A game is a system in which some of the system states are considered by the participants to be more desirable than others and whose unique purpose is the placement of the system by the participants in these states. They achieve this through a set of well-defined operators. Different participants may have different desired states.

A networked game is one in which state changes are performed across the network. In practice two ways of achieving this are possible: each participant keeps a copy of the state locally, and all participants communicate the operations to be performed to all other participants who use this information to update their local state; one reference copy of the state is kept and all operations are transmitted to an actor which updates the state and distributes it to all participants. The first system is commonly called peer-to-peer while the second is a central server system.

The second method is more efficient in resource usage as the operations are transmitted only once and their execution is performed only once. However, a large burden is placed on the actor who maintains the reference state. Practical systems also need to consider that the infrastructure will introduce delay and loss into the transmission of operations and state updates.

In this paper we discuss the architecture of a networked game involving a million participants, each producing a maximum of one operation every second. These numbers are arbitrary and are chosen only in that they are more demanding than any existing multi-player game. For example, while the owners of the EverQuest role-playing game [19] claim that tens of thousands of people can participate simultaneously, a user is in fact restrained to choose one of about 40 servers on which to be located. As it is not possible to move between servers, each server is in fact independent. Therefore, EverQuest is better thought of as 40 independent instances of the same game, each of which handles about 2000 players. Bharambe et al. [5] report that current games have difficulties supporting more than 3000 – 6000 players.

First, we describe the infrastructure which would be needed to support such a game using a central server approach, then we identify weaknesses in this approach and describe how these can be addressed using a hybrid peer-to-peer/central server system. Finally we describe a prototype implementation of the basic component required to support this hybrid model.

2. EXISTING APPROACHES

The hypothetical game has 10^6 players, each of which is capable of sending an operation every second. The logic of this game is such that when a player performs an operation at time *t*, all operations performed at time *t-1* must have already been taken into account in order to calculate the new game state. This is slower than the inter-packet transmission time required by so-called First Person Shooter games (FPS), which is typically less than 100 ms [7].

Using a peer-to-peer model for supporting such a game would involve the transmission of 10^{12} operations per second; supposing each operation was encoded in 100 bytes — a typical client packet size for an FPS game [7] — then 10^{14} bytes of information would be sent across the network every second. This is unfeasible.

With a central server approach, the server receives 100 Mbytes of operations per second and has to transmit the new game state to each of the players. Suppose that the size of the total game state is directly proportional to the number of participants and that each participant's state is represented by 1 Kbyte, i.e. the total state of the game is 1 Gbytes. Sending the whole game state every second to each player is unfeasible; supposing further that only deltas were sent and only 1 byte of information per participant was modified each second this would still require 1 Mbyte of information to be sent to each participant per second.

The obvious conclusion is that in order to scale we cannot treat the game state as monolithic and that very large games must take advantage of the fact that a given participant is oblivious to most of the game sub-states. Many papers have already proposed this approach, for example [5, 8].

The fraction of the game state that interests a given participant is game specific. Let us assume — as is commonly the case for massive multi-player games — that the game has some concept of virtual location and that interest in the activities of the other participants is related to closeness of virtual locations. Say that a given participants has a maximum of a 100 other participants that are close enough for their activity to be of interest.

The server still receives the same number of operations, but needs to send only 100 bytes of state updates to each participant per second, i.e. 100 Mbytes per second. The server receives 100 Mbytes and transmits 100 Mbytes per second. Such figures are high but well within the boundary of currently available network access speeds.

The server sends and receives 10^6 distinct packets per second. These packets are generally carried using UDP; an examination of the loss-free rate at which a UDP server can receive from a client across a range of operating systems, CPUs, Network Interface Cards and buses [11] shows that on an unloaded server about 10^4 small packets per second are sustainable without loss. Assuming that a two-order-of-magnitude increase is possible with specialized hardware, then the server could send and receive at the required rate, but would have no time for any processing.

Distributing the processing over multiple servers is an obvious next step in scaling the game. Suppose that we have 100 servers and the load could be evenly balanced across them, then each server would only have to support the reception and transmission of 10^4 packets per second; we assume this leaves enough time to perform the processing in order to produce the game's new state.

However, each server has to receive the packets from participants who are in the same virtual location. Assume there are 10^3 distinct locations in the game each containing 10^3 players, then each server is responsible for 10 locations and must receive all the operations for those locations before it can calculate the new state.

It would be possible to place another server between the edge router and server farm which examines the virtual location of the emitter of a packet and forwards the packet to the server responsible for maintaining that location. This routing server would have to read and write 10^6 packets per second. As it performs no computation it seems more sensible to combine its action with that of the router in the form of an application-specific router, capable of looking at data carried in the payload of the packet to make a forwarding decision.

In summary, assuming a hundredfold increase in server I/O, approximately 100 servers each with no limitation on processing and interconnected to an almost loss-free Internet via a piece of dedicated application-specific hardware capable of "intelligent" forwarding at 1 Gbit per seconds could sustain a 10^6 person game in which each player sends 1 operation every second.

As a reality check lets us compare the required infrastructure for the example game with that of a real large networked application: the 1998 Winter Olympic Game's web site was supported by thirteen IBM SP/2 mainframes containing a total of 143 processors. On average it handled about 40 million requests per day, with a peak load of about 2000 requests per second. An upper bound of 30 seconds of end-to-end delay was set for users requesting a dynamic web page [6]. As the example game is required to handle two orders of magnitude more requests per second and process them in two orders of magnitude less time, it is to be expected that even with increases in processing power the infrastructure would be much larger than that of [6].

Even allowing for the idealized assumptions made, this solution is not very satisfactory. Firstly, it is centralized and thus vulnerable to failure. This could be handled by replicating the servers over a number of geographical sites, at the cost of additional servers and of keeping the replicated state on the servers synchronized.

More importantly, the solution is very dependent on the example we chose. If we decided that two events per second were needed, then the number of servers needed would double. On the other hand, if only a tenth of the anticipated number of participants played, our infrastructure would be largely over-dimensioned. A better solution is one that implicitly scales with the number of users, i.e. that as users join the total amount of processing, bandwidth and memory I/O increases.

The resources available to a peer-to-peer system increase with the number of peers. Although, as we have seen, these systems cannot scale to a large number of peers, we earlier assumed that at a given moment in the 10^6 person game there were fewer than 100 other participants with which a player was required to communicate. For our hypothetical game, each player would therefore only need to be able to receive 100 operations per second. The bandwidth requirements of each client would be 100 Kbytes/s, which is well in the range of available ADSL or cable-modem offerings. Assuming a reasonable amount of processing for each operation, e.g. comparable to that in current peer-to-peer games, normal desktop PCs should be quite capable of supporting the execution of the game logic. If it were possible to construct the massive game as a federated set of small peer-to-peer systems, then the game would not only scale with the number of players, but (best of all from the game provider's point of the view) the clients would be supplying both bandwidth and processing themselves.

In the next section we explain how this can be achieved.

3. A HYBRID APPROACH

The large game is partitioned into multiple peer-to-peer systems. These systems are federated in such a way that the user is unaware when crossing the boundary between them. The basic enabler of this federated system is an efficient and reliable means of communicating within the geographically distributed peer-to-peer groups. This is achieved using a network-based server capable of very fast packet forwarding, called a *Booster Box*. While game data is transmitted between clients without passing through a central server, control servers are used for out-of-band control information needed to configure and manage the system. All game data packets are carried using UDP, while control operations use TCP. An overview of this architecture is shown in Figure 1.

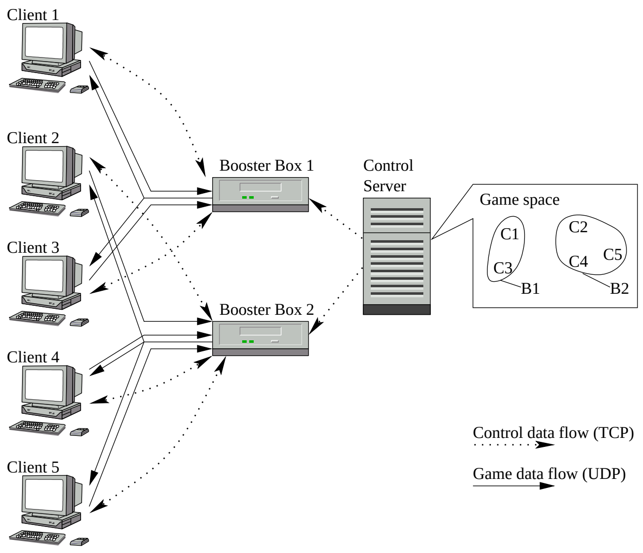

Figure 1: Architecture Overview

It is important to note that existing games cannot run as is on this hybrid system; existing commercial solutions [12] already recognize the need for new models for game development that explicitly take into account the scaling consideration required for very large games. However, our hybrid architecture, while imposing certain constraints on game developers, is game independent and can be used for many different games.

3.1 Game Packet Forwarding

3.1.1 *Multicasting*

A booster box maintains the list of clients resident at a given virtual location. The client sends data packets to the booster box which then sends them to all co-located clients. The client does not know the addresses of the other clients, nor does it communicate with them directly. This reduces the communication overhead for the clients significantly and matches with the asymmetric bandwidth resources of ADSL and cable modems in that the upstream direction, which is used for sending, provides less capacity than the downstream direction, which is used for receiving.

The booster box implements the multicast function without requiring the network to support IP multicast. The clients send unicast IP packets to the addressable booster box, which in turn generates many unicast IP packets to send to the members of the peer group.

Consider again the hypothetical game described in Section 2, suppose the maximum number of clients that a peer-to-peer system can support is 100, then 10^4 distinct virtual locations/multicast domains are required to support a 10^6 person game. The controlling booster box receives 1 operation each second from each client in a given virtual location. As the booster box multicasts all operations in each virtual location, the booster box forwards 100 packets to all 100 clients, i.e 10^4 packets per second. As we have seen in Section 2 this is about the rate at which current commodity systems can handle UDP packets without loss. However, it would require 1 booster per box per virtual location, i.e. 10^4 booster boxes. Such a system would be both costly and difficult to manage.

Instead we implement the multicast function in programmable hardware [15]. This allows forwarding rates of more than 10^6 packets per second, allowing a single booster box to handle 100 virtual locations, and thereby reducing the number of booster boxes to more manageable levels. A booster box may handle multiple distinct games, in which case the number of clients per game is reduced accordingly.

Booster boxes are typically distributed throughout the network; as there is no correlation between network and virtual location, the round-trip Time (RTT) between a client and its controlling booster box and the RTT between client and a central server are likely to be comparable. However, as the booster box performs no computation on the data path, the RTT between two clients communicating via a booster box is lower than if they were communicating via a server.

3.1.2 *Handling Packet Loss*

Loss leads to incoherent states between clients, and at a certain level of loss games become unplayable. The use of reliable transport protocols such as TCP involves a larger latency as the sender after sending a window of data must wait for an acknowledgment before sending the next. Allowing for 200 ms RTT, then supposing each operation is acknowledged, a client can only send two operations per second. If the size of TCP's receive window is larger than that of a game packet, multiple packets can be acknowledged with the same ACK, but loss can lead to a received packet not being expedited to the user process until the preceding packet has been retransmitted. For these reasons UDP is generally preferred over TCP in games. However, the application developer must handle retransmission at the application level.

The booster box adds support for application-layer retransmissions. When a packet arrives at a booster box, the forwarding mechanism adds a packet sequence number. The sequence number increases monotonically. A receiver can determine whether it has not received a packet by examining whether any sequence numbers are missing in the stream of packets it receives. Noncontiguous arrival may be caused by out-of-order delivery as well as loss. The application must wait some window of time before deciding that a packet has been lost and should be retransmitted; how long it waits is application specific. Besides allowing retransmissions, the sequence numbers also define a causal order on the packets. This allows the clients to keep a consistent state by processing incoming packets in the same sequence.

A client requests retransmission of the packet from the booster box, rather than from the initial sender. The client does not know the address of the sender, and the burden for handling retransmissions is placed solely on the booster box. The booster box can only keep a finite number of already transmitted packet in memory, so packet retransmission is not possible for old packets. As game-related packets are in any case only valid for a short period of time, this is not a drawback.

Periodically the booster box sends a "heart beat" packet to all clients in a location, repeating the sequence number sent last and the lowest (oldest) available one it has in memory. In this way clients can distinguish inactivity from high levels of loss and can determine whether retransmission is possible.

The booster box behaves similarly to the *master* component of the "Multicast Transport Protocol" (MTP) de-

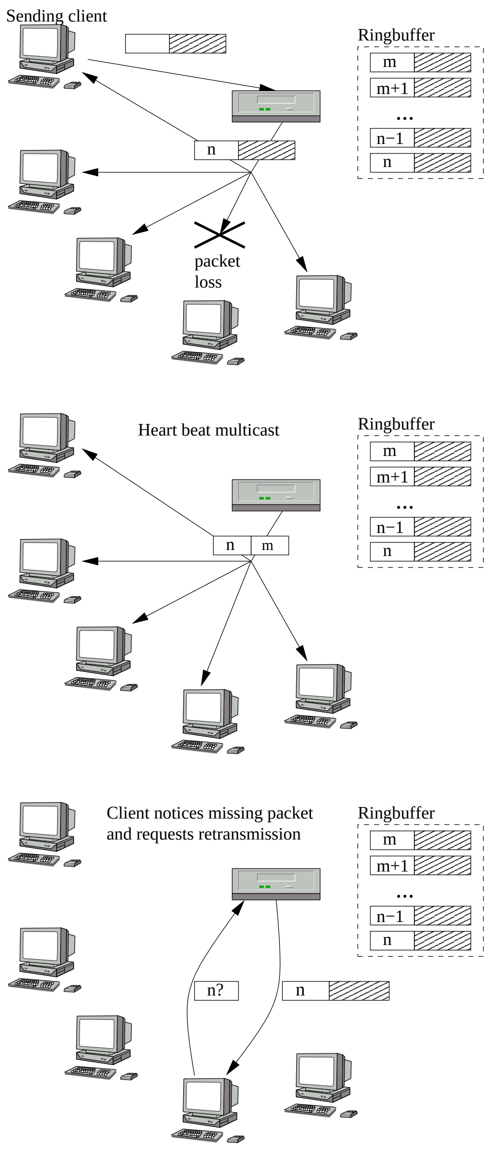

Figure 2: Handling Packet Loss

scribed in [3]. However, MTP offers a richer set of functions, in particular a rate-based flow-control mechanism that is implemented using a token-passing scheme. The token-passing scheme limits the amount of parallelism and also significantly adds to the delay and therefore cannot be used for networked games.

Levine et al. [12] describes a method of application-layer transmission between a central game server and client. The application-layer protocol allows senders to distinguish between packets that should be reliably and unreliably sent, keeping a copy of the reliable ones in memory in case of the need for retransmission. The problem with applying such a model of retransmission in a central server system is that unless it can adapt to the current network state, then in times of loss clients' requests for retransmission may become correlated; requests for retransmission may cause more loss, which causes even more retransmission. There is less danger of this form of positive feedback forming in the architecture described here as the clients do not all communicate with the same server, and so it is less likely that loss will be tightly correlated. However, it is still possible that large numbers of clients will simultaneously request retransmission if the booster box, or the network close to it, is subject to disruption. The solution is to ensure that retransmission requests never exceed a certain threshold by reducing the number of advertised packets in the heart-beat message. Because clients only ask for retransmission of packets in this window, as the window becomes smaller the total number of retransmissions decreases. If the window is empty, then no retransmissions will be sent. In effect, we trade off reliable delivery of packets against the possibility of retransmission causing failure.

3.1.3 *Maintaining Location State*

There are three types of state:

static state: e.g. maps, which either never change or change only infrequently;

client-specific state: e.g. the client's character, which changes during a game session, and which is persistent between sessions;

location state: e.g. which clients are currently present at a location; this state is transient and constantly changing.

The static state is kept on the client only. The client and control server collaborate to maintain and update the client-specific state, but the booster boxes manage the location state.

The state-maintaining function of the booster box is distinct from that of its packet-forwarding one, because:

- packet forwarding is application independent, while state maintenance requires running parts of the game application;
- packet forwarding is simple but must be performed very efficiently. An incoming packet is processed to update the game state and at the same time it is multicast to all clients at the same location. This is performed in parallel.

Consequently in the prototype implementation described in Section 4, these two distinct functions are actually performed at different layers in the booster box architecture and on distinct processors.

3.2 Game Control Operations

3.2.1 *Booster Box/Control Server Interaction*

The control server is responsible for dividing the virtual space into separate locations and assigning these locations to booster boxes. As the number of booster boxes in the system is quite low, e.g. in the range of a few hundred, it is possible to maintain a mapping table on a single server. For redundancy reasons, multiple servers might still be used, but are not strictly required for scalability.

The control server can be viewed as a central application-level router that computes and maintains the table containing the mapping between virtual locations and booster box IP addresses. This is not a one-to-one mapping, as a single booster box is expected to handle many virtual locations, and each virtual location handled by a booster box is identified by a corresponding TCP/UDP port. In addition, a location can be configured with one or more backup booster boxes which take over if the primary booster box fails. We assume that at least one backup booster box is available for each virtual location.

When a booster box is added, it connects to the control server and registers itself. The server then computes a new mapping of virtual locations to booster boxes and distributes the resulting location table to all booster boxes, which in turn forward it to the attached clients. Note that the size of virtual locations itself does not change, just the mapping of locations to booster boxes. This ensures that the view of clients does not change, even if their location gets remapped to a different booster box. Also, remapping is done in the least disruptive manner possible. Assuming that a booster box is added to a system where n virtual locations and b booster boxes exist, then exactly $\lceil n/(b+1) \rceil$ virtual locations are remapped to the new booster box.

3.2.2 *Client/Control Server Interaction*

When the game software is started on a client, it communicates with the control server. The cient must authenticate itself to the control server, for example using the Kerberos [1] security mechanism. Authentication is necessary to ensure that only paying clients participate in the game. The control server supplies a ticket to the client, which the client uses to authenticate itself with booster box. These tickets have an expiry date, which limits the time that the client can participate in the game without having to be reauthenicated by the server.

After a successful authentication and authorization, the client receives the location table and the client's persistent state. It then terminates its connection with the control server and communicates with booster boxes only. It forwards game-related traffic to the booster box that controls the virtual location in which the client is resident. As the player moves between virtual locations, the game software changes the booster box destination.

When a client terminates a game session the new client-specific state is stored on the control server.

3.2.3 *Client/Booster Box Interaction*

As well as acting as a multicast reflector for client packets the booster box supports a set of control operations with which application software can use to communicate with it directly:

add request , which adds the client's address to the list of addresses in the multicast domain.

remove request , which remove its address.

stateUpdate request , which requests the entire local game state for a given virtual location.

locationTableChanged event , which informs the clients about a new location table.

Each time a client changes virtual location, the client obtains the state of the new virtual locations using the *stateUpdate* command from the booster box controlling that location. In order to ensure a quick transition between virtual locations the application software writer may choose to be a member not only of the virtual location at which the client is resident but also of all those around it. Assuming each virtual location is a hexagon, this increases by six the amount of processing and bandwidth required of the client, as well as reducing the number of distinct clients a booster box can support.

Booster boxes are typically distributed throughout the network; as there is no correlation between network and virtual location, the RTT between a client and its controlling booster box and the RTT between client and a central server are likely to be comparable. However, as the booster box performs no computation on the data path, the RTT between two clients communicating via a booster box is lower than if they were communicating via a server.

3.3 Non-Uniform Client Distribution

It is highly unlikely that clients are evenly distributed across the entire virtual space. It is more likely that at given times certain locations are highly popular while others are almost unpopulated. This is a potential problem for the architecture described here as beyond a certain threshold of participants at a location, clients become saturated by operations and the controller booster boxes cannot keep up.

One possibility would be to do nothing at all and hope that the degradation at a given location will cause players to flee to locations with better performance (i.e. with fewer players). However, such a system would be subject to oscillations as a large number of players moved from one location to avoid saturation only to saturate the new location.

The preferred solution is to dynamically adapt the number of virtual locations that a booster box handles. If the load on a booster box crosses a predefined threshold, it sends an overload request to the control server. The control server in turn computes a new location table that shifts a small number of virtual locations from the overloaded booster box to some less loaded booster boxes. The original booster box transfers the set of client IP addresses to the server, which then passes it to the new booster boxes.

The new location table is distributed to all the booster boxes, which in turn distribute to the clients in their locations. Packets that are sent to the previous booster box will for a time be forwarded to the new booster box, with a redirect message being sent back to the client telling it that it should update its location table.

3.4 Handling Failure

In very large distributed systems failure, of part of the system is the norm rather than the exception. The basic model of handling failure is to soft state for control, i.e. state that needed to be periodically refreshed in order to persist.

Client's periodically use the *add* operation to ensure their continued presence in a given location. If a client has not renewed its presence after a certain time, the booster box assumes the client has gone and removes it from the list of residents of the virtual location.

If a client has not received any packets from the booster box for a period, it can assume that either the booster box failed or that there no longer is a network connecting it to the booster box. It sends a *ping* message to the back-up booster box in its location table. If the back-up booster box receives the *ping* it replies, as well as forwarding the ping to the primary booster box, which if it receives it will also reply through the back-up. If the client receives no replies, it assumes there is severe network disruption and tries to terminate cleanly. If it receives a reply from the back-up booster box and the primary booster box, it assumes that there is a small amount of network disruption and will communicate via the back-up to the primary. If only the back-up replies, then it assumes the primary booster box has failed, and switches to the back-up booster box. This simple solution does not work reliably in the case of network partitions where primary and backup booster boxes get separated. In this case, primary and backup booster boxes will each handle a part of the clients. The state information has to be resynchronized once the network partition has been repaired. This is a topic for future research.

4. IMPLEMENTATION

In this section we describe how we have implemented parts of the architecture described in Section 3.

A booster box is a general-purpose computation platform capable of forwarding packets at very high speed. A piece of application-specific code, called a *booster*, is executed on the booster box. The booster box provides interfaces that allow the booster to observe and manipulate data streams, and to participate in control and management protocol operations. Boosters may cache, aggregate, filter, and reroute packets in an application-specific way. Boosters are not limited to gaming applications, and are useful in other contexts as explained in Section 5.

We envisage that booster boxes will be deployed at the edges of an ISP network, attached directly to the IPS's access routers. These booster boxes might be considered as members of the broad range of appliances that the IETF terms "middleboxes"[20]. The ISP will allow privileged third parties to instrument their booster boxes in a way specific to the applications they wish to support.

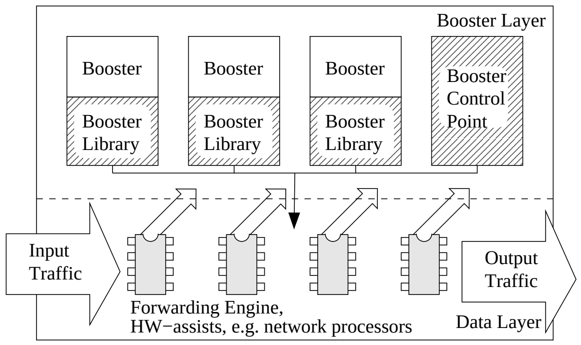

Figure 3: Booster- and data-layer overview

Booster boxes are "conventional" computers with a Linux operating system and equipped with a IBM-NP4GS3 network processor card [15]. Network processors allow booster boxes to process and forward packets at line speed. As depicted in Figure 3, the booster box architecture is divided into a *booster layer*, in which the high-level logic resides, and a *data layer*, which actually does the packet forwarding.

4.1 Data Layer

The main role of this layer is to embody forwarding-related functions. Typically such functions consist of redirecting packets belonging to a given flow (based on addresses and ports), modifying headers, computing checksums, duplicating packets etc. The data layer must be able to handle these operations at speeds equivalent to the port of a residential access router, i.e. in the range of 155 Mbit/s to 1 Gbit/s. A pure software solution is not able to handle such line speeds, whereas a pure hardware solution does not offer sufficient flexibility. Our approach is to use network processors for implementing the data layer of the booster box. Although the term network processor (NP) covers a wide variety of processors with different capabilities and designed for different markets — a good overview can be found in [17] — the simplest way to think of a NP is as a general-purpose processor with access to many network-specific co-processors, performing tasks such as checksum generation, table look-up, and header comparison. Arbitrary network-forwarding code can be written in a high-level language such as C (augmented with pragmas for co-processor invocation) compiled and loaded into such a processor. The NP therefore is a mid-point between a pure hardware and pure software solution.

Two functions are required in the data layer in order to support the architecture described in Section 3: receiving game and control packets sent by the client, and multicasting game packets to all the clients belonging to a virtual location.

Booster boxes are built on a Linux kernel with a TCP/IP stack and are, therefore, network entities addressable with an IP address. The data-layer acts here as a redirection mechanisms for all the incoming packets. TCP packets are directly forwarded to the TCP/IP stack and delivered to the appropriate booster based on the TCP port. A booster in charge of a given virtual location is identified by the TCP or UDP port.

Game packets (UDP packets) are treated differently. These packets have to be forwarded to the booster in charge of the virtual location and as well as being multicast to all the clients in that location. In addition, a sequence number has to be inserted to enable loss detection and retransmission. The use of a network processor is particularly suited implementing this process. When such a packet is received an increasing sequence number is inserted between the payload and the UDP header. After the header fields are updated (checksum and length), the packet is forwarded to the Linux protocol stack and delivered to the appropriate booster. A copy of the packet is kept in the network processor for multicasting. The UDP port is extracted and used to look up a table associating the UDP port to the list of clients belonging to this virtual location. The copy of the packet is then forwarded to all the clients after the appropriate UDP and IP headers have been computed. All these operations are executed at line speed. Figure 4 gives a schematic overview of this process.

The data layer caches game packets for retransmission in case of loss. Game packets corresponding to a given virtual location are stored in a circular buffer associated with the location. This buffer keeps the last N packets. The booster box generates heart-beat packets containing the lowest and the highest sequence number of the packets in this buffer. When a client detects a mismatch in the sequence number it sends a retransmit message to the booster box. The data layer of the booster box will then resend the missing packet to the client provided it still exists in memory. Additional logic for adjusting the heart-beat messages, as explained in

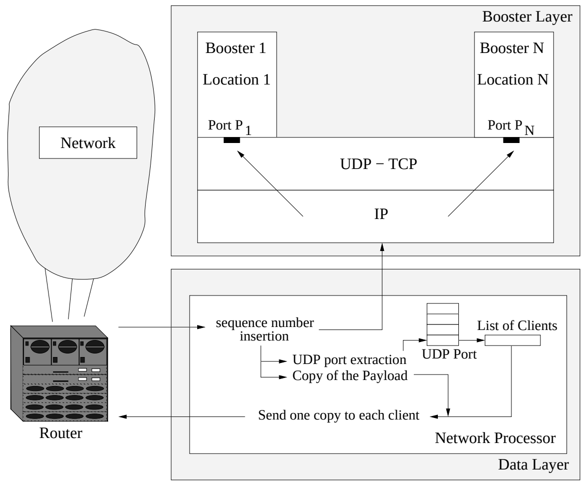

Figure 4: Processing of incoming game packets

Section 3, in order prevent retransmission itself from causing loss, is also implemented on the NP.

4.2 Booster Layer

The booster layer consists of a set of boosters and a control point that is common to all boosters. Each booster is a combination of application-specific logic and generic booster library functions. The application-specific code is trusted, being either written by the ISPs themselves or supplied by trusted third parties. The library contains the API through which the boosters control the data-layer operations. The boosters execute independently; however, certain operations need to be coordinated. The booster control point coordinates the boosters, and manages any data that is common to all of them. For example, the booster control point coordinates the assignment of port numbers to boosters and communicates them to the central control server.

A booster for the architecture described in Section 3 needs to implement the four operations: *add*, *remove*, *stateUpdate* and *locationTableChanged* described in Section 3.2.3. The first two operations manipulate the multicast-routing table in the network processor.

The booster receives a copy of each game UDP packet handled by the network processor. This enables the booster to track the state of the game in its virtual location. It keeps the up-to-date state in local storage. The state contains the sequence number of the packet that last modified the state.

When a new client enters a virtual location, it opens a TCP socket to the associated booster and registers with the *add* primitive. Subsequently the client issues a *stateUpdate* primitive, causing the booster to transmit an up-to-date state of the game in the virtual location to the client. The booster adds the client's IP address in the entry of the multicast-routing table. At this point the clients receives all the game packets associated with the virtual location. As the state contains the reference sequence number, the client can ask for retransmission of any packets it might have missed during the initialization phase. Therefore this approach guarantees that the client can synchronize.

Periodically, a client refreshes the soft-state by sending an *add* request to the booster. If the booster does not receive an *add* from a given client after a certain period of time, it removes the client from the location.

5. OTHER APPLICATIONS

Booster boxes are general-purpose computation platforms and as such can be used for enhancing the performance of other non game applications. We briefly mention the nature of these applications.

5.1 Peer-to-Peer File Sharing

Gnutella [10] is an example of a file-sharing protocol that does not require a central server. Such approaches are very resilient to failure and have other advantages such as the number of locations at which a file can be found is proportional to the file's popularity. However, they are not very scalable, as a single search operation results in the flooding of a large number of request messages.

Current work investigates making peer-to-peer file-sharing protocols more scalable by using boosters. Boosters can reduce the amount of traffic by caching, aggregating requests, and performing intelligent forwarding.

5.2 Floating Car Data

Floating Car Data (FCD) [2] is a label for a set of different initiatives that involve gathering sensor information from cars, processing it, and using the result to generate useful information such as traffic-flow predictions. Sensor devices are placed in the cars to measure parameters such as speed or location information. The resulting data is transmitted in real time to back-end servers, first using General Packet Radio Service (GPRS) and later third-generation wireless communication and/or wireless LAN technologies. Potentially a huge amount of data can be produced by each car. Current work looks at how boosters can aggregate such information, e.g. by sending a single message saying that 50 cars are traveling at 40 km/h, rather than sending 50 separate messages.

5.3 Overlays Using Measurement-Based Routing

Previous work has shown that routing inefficiency in the public Internet results in the overuse of certain links while others remain idle. Booster boxes can form an overlay over the public Internet to allow certain packets to be carried over paths other than those that would be chosen by normal IP routing. The booster boxes measure loss and delay between their peers in the overlay in order to detect and avoid congestion. This work is explained in some detail in [4].

6. RELATED WORK

A good overview of the problems arising with the development of networked multi-player computer games can be found in [18]. Funkhouser [9] proposed placing "Message Servers" in the network to enhance server scalability. Each of these entities is in charge of a number of clients and manages message communications on their behalf. In addition Message Servers can perform specific processing on the information. Similar approaches have been proposed in [8, 13]. Using booster boxes removes the need for transferring game packets through a central server. The control function of the booster boxes in some ways can be considered as an intelligent proxy for the control server, e.g. client add and remove themselves from a location by communicating with the booster box rather than with the server.

Bharambe etal. [5] attempt to ensure that game players receive only relevant events using a publish/subscribe model. It specifies a routing mechanism whereby nodes are arranged in attribute chains, such that each node is responsible for keeping track of other nodes interested in a given attribute range, e.g. virtual location. The system described in [5] permits a more scalable peer-to-peer system, and has the advantage of not requiring any infrastructure other than the clients themselves; however this in turn limits the number of clients it can support.

The attempt described to make peer-to-peer games more scalable has analogies with that being done in the field of file sharing, e.g. [16]. In this work different peers are allocated different parts of the file name space to manage. Active research topics involve ensuring that the files are balanced across the set of available peers and that a given file can be quickly found without requiring requests to be flooded. However, other than the peers agreeing on how the name space should be structured and divided, there is no need for coordination between peers in file-sharing systems. This clearly is not the case for games.

The work described in the overview paper of military simulation [14] is simular in nature to the architecture described here. For example, in the Navel Postgraduate School Net (NPSNET), the space is divided into cells in which information is multicast. One member of the cell is responsible for adding and removing others as well as giving a new member the current state of the cell. The difference between military simulations and games is rather of context than form; simulations are performed in a highly controlled network where the participants in the simulations and their activities are known in advance. Consequently the simulation designers can design the infrastructure with this in mind. There are more unknowns in running commercial games over the public Internet; for example, no player can be trusted to be a group leader as in NPSNET.

7. CONCLUSION

By dividing very large multi-players games into a federation of many peer-to-peer systems, we allow the resources required to support the game to scale with the number of participants. This federation is enabled by separating control from data operations, such that control is handled by conventional central servers, while the actual data forwarding is performed by network-based servers with special hardware support for fast packet forwarding. The latter is made possible by the recent availability of highly configurable network processors.

Our work to date has involved prototyping various parts of the architecture in order to gain a better understanding of the system behaviour. In this paper, we have tried to give an overview of how an entire system could be made to work. However, determing its overall feasability would require much larger-scale experiments than we have currently been able to perform.

8. REFERENCES

[1] Kerberos: The network authentication protocol. http://web.mit.edu/kerberos/www.

[2] AMI-C. *Use Cases Release 1 SPEC 1003*. Automobile Multimedia Interface Consortium, 2001. http://www.ami-c.org.

[3] S. Armstrong, A. Freier, and K. Marzullo. Multicast transport protocol. Request for Comments 1301, Internet Engineering Task Force, Feb. 1992.

[4] D. Bauer, S. Rooney, P. Scotton, S. Buchegger, and I. Iliadis. The Performance of Measurement-Based Overlay Networks. Technical Report RZ3415, IBM Research, 2002. http://www.research.ibm.com/resources/paper_search.shtml.

[5] A. Bharambe, S. Rio, and S. Seshan. Mercury: A Scalable Publish-Subscribe System for Internet Games. In *NetGames 2002 – First Workshop on Network and System Support for Games*, Braunschweig, Germany, Apr. 2002.

[6] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites. In *Proceedings of ACM/IEEE Supercomputing '98 (SC98)*, Orlando, FL, Nov. 1998.

[7] J. Farber. Network Game Traffic Modeling. In *NetGames 2002 – First Workshop on Network and System Support for Games*, Braunschweig, Germany, Apr. 2002.

[8] S. Fiedler, M. Wallner, and M. Weber. A Communication Architecture for Massive Multiplayer Games. In *NetGames 2002 – First Workshop on Network and System Support for Games*, Braunschweig, Germany, Apr. 2002.

[9] T. Funkhouser. Network Services for Multi-User Virutal Environments. In *IEEE Network Realities*, Boston, MA, Oct. 1995.

[10] The Gnutella Protocol Specification v0.4. http://www.clip2.com/GnutellaProtocol04.pdf. Document Revision 1.2.

[11] Hewlett-Packard. *NetPerf: A Network Performance Benchmark*. Information Networks Division, 1995. Revision 2.0.

[12] D. Levine, B. Whitebook, and M. C. Wirt. *A Massively Multiplayer Manifesto*. Butterfly.net, Inc., 123 East German St. Shepherdstown WV 25554, May 2002. Version 1.1.

[13] M. Mauve, S. Fischer, and J. Widmer. A Generic Proxy System for Networked Computer Games. In *NetGames 2002 – First Workshop on Network and System Support for Games*, Braunschweig, Germany, Apr. 2002.

[14] K. Morse. Interest management in large-scale distributed simulations. Technical report, Dept of Information and Computer Science, University of California, Irvine, 1996.

[15] IBM PowerNP NP4GS3. http://www-3.ibm.com/chips/products/wired/products/np4gs3.html.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In *Proceedings of ACM SIGCOMM*, pages 161–172, Aug. 2001.

[17] N. Shah. Understanding Network Processors. Technical report, University of California at Berkeley, Sept. 2001. http://www-cad.eecs.berkeley.edu/~niraj/web/research/network_processors.htm.

[18] J. Smed, T. Kaukoranta, and H. Hakonen. Aspects of Networking in Multiplayer Computer Games. In L. W. Sing, W. H. Man, and W. Wai, editors, *Proceedings of*

International Conference on Application and Development of Computer Games in the 21st Century, pages 74–81, Hong Kong SAR, China, Nov. 2001.

[19] Sony Online Entertainment Inc. EverQuest. http://www.everquest.com, 2002.

[20] P. Srisuresh, J. Kuthan, J. Rosenberg, A. Molitor, and A. Rayhan. Middlebox Communication Architecture and Framework. Internet Draft, Internet Engineering Task Force, Dec. 2001.