RZ 3519 (# 93907) 07/07/2003
Computer Science 11 pages

Research Report

Efficient Programmable Middleboxes for Scaling Large Distributed Applications

Sean Rooney, Daniel Bauer, Paolo Scotton

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Some reports are available at http://domino.watson.ibm.com/library/Cyberdig.nsf/home.

Research
Almaden · Austin · Beijing · Delhi · Haifa · T.J. Watson · Tokyo · Zurich

Efficient Programmable Middleboxes for Scaling Large Distributed Applications

Sean Rooney, Daniel Bauer, Paolo Scotton
IBM Research
Zurich Research Laboratory
Säumerstrasse 4
8803 Rüschlikon, Switzerland

***Abstract*—**
A range of application classes is emerging in the Internet whose characteristics differ considerably from those of the point-to-point request/response protocols, which have dominated its recent past. Sensor applications, peer-to-peer systems, and massive multiplayer on-line games are multi-point applications that share both a need for the timely correlation of data generated by different participants in a session and a potentially unlimited number of session participants.

We propose the use of middleboxes residing in the network to help in the scaling of these applications. As these emerging applications are not readily subject to standardization, we believe that a programmable model, in which a given middlebox can be instrumented to support various applications either simultaneously or over time, is desirable. We describe our work to date in building such a middlebox that makes use of hardware assists in its data path to maintain high performance.

I. Introduction

Designing distributed applications that scale involves ensuring that the amount of data transmitted and processed grows in an acceptable fashion with the number of participants. Techniques that reduce the amount of data transmitted, while allowing applications the same level of information are desirable. For example, within the existing Internet, web-page caching coupled with HTTP request redirection enables Web sites to handle larger loads than would otherwise be attainable. Caching as a technique is least beneficial when information is quickly changing; at some rate of change it brings no benefit at all. Other techniques that can be used to reduce data are:

- *intelligent forwarding*, sending data only to some subset of participants;
- *aggregation*, combining data from different participants before forwarding it;
- *application-level filtering*, dropping packets that are no longer relevant, for example those out-of-date;
- *attenuation*, reducing the amount of data carried, based on the association between the sender and the receiver.

Currently none of these techniques is widely deployed in the Internet as they are not readily applicable to point-to-point request/response protocols.

Our contention is that emerging applications will require the deployment of these techniques in the network to allow them to scale to large numbers of participants. We identify three such application types: large sensor applications, peer-to-peer file-sharing systems, and massive multiplayer on-line games. Our work involves designing and building the devices capable of aiding the scalability of these applications through the use of the functions listed above.

The community has settled on the name middleboxes [1] for these intermediate devices, a middlebox service being loosely defined as one that requires application logic but is typically executed on a dedicated device in the network.

Middlebox services are limited by the fact that the application logic is instrumented either in an ASIC, thereby reducing the flexibility of the device, or in software, reducing the throughput it can sustain and therefore the location in the network at which it can be deployed. A Network Processor (NP) [2] is a processor that interacts with special hardware, for example, in the form of co-processors, to achieve faster packet-forwarding rates than a conventional processor could sustain. Originally conceived as a means of allowing equipment vendors to use the same hardware across a range of different network devices, NPs offer the possibility of allowing more complex application-specific functions to be executed in the network without degrading network performance on a general purpose configurable middlebox.

Currently, we are building such a general purpose middlebox, which we term a *booster box*. A booster box may run one or more application boosters – or *boosters* for short. The term booster box is inspired from [3], which described protocol boosters. Although our application boosters are different in nature from protocol boosters, they share the design principle that boosters, as much as

possible, should *transparently* improve performance.

We first describe our current implementation of the booster box, then we present our experience in using it to support application instances in each of the three target classes.

II. Architecture

The booster box is a one or two port general purpose computation platform with an interface that allows application-specific logic to configure the underlying packet forwarding engine. In our implementation this packet fowarding engine is an NP, but this is hidden from the application. In fact, often we try out functions using the packet forwarding capabilities of a Linux kernel before instrumenting equivalent capabilities on the NP. From an application point of view there is no functional distinction between these two forwarding engines. Multiple distinct application boosters may be executed in parallel. Arbitrary third parties are not allowed to run boosters so although booster boxes need to be protected against intruders, security concerns are no greater than those for other pieces of network equipment.

A. Operation Taxonomy

Carpenter in [4] gives a taxonomy of middleboxes and discusses their effect on the end-to-end principle that underlines the Internet. A range of various possible middleboxes function are discussed, a distinction being made between middleboxes that terminate connections and those that modify or divert packets on the fly. While booster boxes support both modes of operation, we recognize the problem of an invisible middlebox's failure having an adverse affect on network robustness. In consequence, from the application's point of view the booster box should be fully transparent or fully opaque:

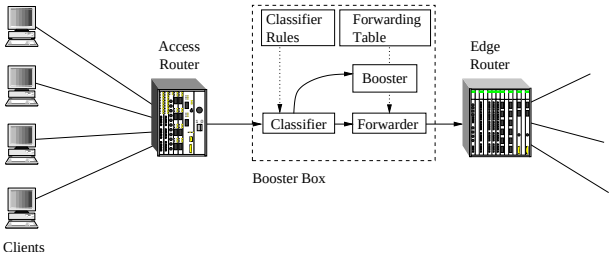

Fig. 1. Booster box operation in transparent mode

transparent mode: if the booster box is fully transparent neither client or server are aware of its existence. A booster box present on the path between client and server intercepts packets belonging to the application, processes and forwards them on as appropriate. If there are no booster boxes on the path or if a booster box fails, then packets reach the destination without being preprocessed. To operate in this mode a booster box is inserted in a link between two routers e.g. between an access router and an edge router. The traffic is sent through a classifier which identifies the packets to be boosted. Packets are classified by source, destination address and port, and protocol type — on-going work will extend the classification scheme to use arbitrary information in the payload. Such packets are diverted to a piece of logic implementing the booster functions while the rest of the traffic is forwarded unchanged. Our assumption is that only a small fraction of the overall traffic is handled by boosters. Figure 1 illustrates this principle.

opaque mode: if a booster box is opaque, then to the client it appears as another addressable end-point. In effect it is a network based server which may use NP support for network specific functions, for example, efficient application-layer multicast as described in Section III-C. A given booster box executing different boosters may be opaque to certain applications and transparent to others.

B. Design

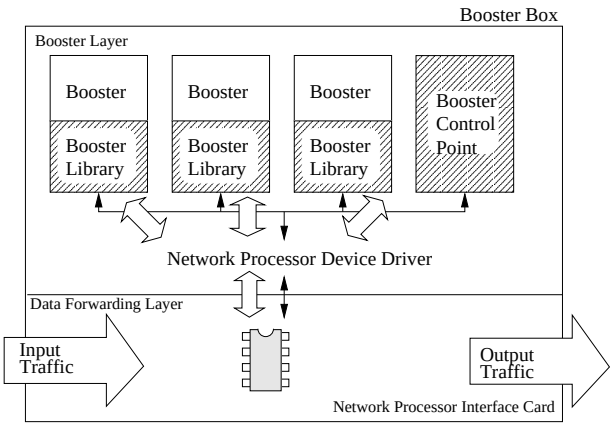

Fig. 2. Booster Box Architecture Overview

At an abstract level the booster box can be thought of as two distinct layers, as shown in Figure 2:

- a booster layer that contains the application-specific code as well as some control functions;
- a data-forwarding layer that contains the code executed upon the arrival of a packet in the data path.

This view is helpful for description and follows the well known control/data division of a network element. In practice, however, the division is not so crisp. When a packet matches a classifier rule in the data path an operation is performed on it. Whether this operation is actually performed by the network processor itself or redirected to the main processor, is implementation specific. It depends on the capabilities of the network processor,

the code currently instrumented on it and the complexity of the operation. Ideally the application program writer would supply an abstract description of the required operation and the appropriate code would be generated for the given environment. Building such a code generator would be extremely complex. Instead we specify an API between the two which minimally allows an application to *redirect* or *copy* classified packets to a user level process, but may contain other NP specific operations as well. This approach is some what ad hoc, but is adequate for our current needs.

At a practical level, boosters are Linux processes. The application specific code of a booster is linked to a booster library. This library contains the API through which the application specific code instruments the data layer. The API hides implementation details of the data layer and uses an event handling mechanism based on asynchronous operations. In addition, the booster library provides asynchronous versions of most socket library calls as well as a timer service.

A booster box controller process running in the booster layer coordinates the action of boosters. For example, before the booster library transmits a redirect request to the NP, it first communicates with the controller to ensure that the operation is allowed. This prevents boosters from inadvertently interfering with each other. The booster controller is also responsible for general management tasks, such as installing new code on the NP, loading the kernel modules that enable communication with it, etc.

The library offers interfaces to other general services that we have found to be useful, for example, a directory service in which the booster can store persistent state.

The data layer operates on individual packets as instructed by the booster layer. It consists of a classifier that examines all incoming packets. The default action is to forward incoming packets using a forwarding table provided by the booster box control process. Non-default actions consist of redirecting or copying packets. These actions are defined by the boosters and downloaded to the data layer by the booster library.

C. Implementation

Physically a booster box is a Linux PC with an IBM network processor board attached across the PCI bus. The NP board contains an IBM NP4GS3 processor that controls three gigabit ethernet ports, out of which two are used. The NP processes incoming packets without interacting with other components in the system. It can be seen as a programmable *router on a card* with an aggregated performance of over 2000 MIPS.

Communication with the Linux host computer takes place through virtual ethernet interfaces established by the NP device driver. Packets that are sent out on a virtual ethernet interface first traverse the kernel protocol stack and are then forwarded to the NP by the device driver, where they are transmitted over the corresponding physical port. Similarly, packets that arrive on a physical port with a local destination are made available on the virtual interfaces by the NP device driver. Figure 3 illustrates this process.

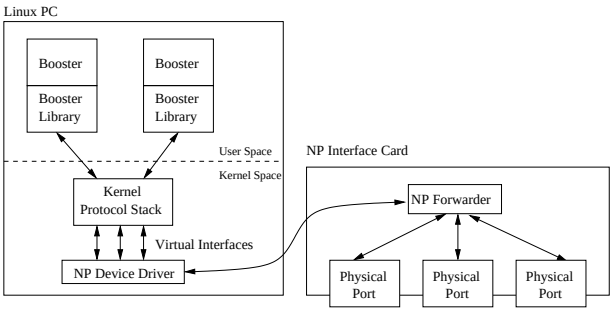

Fig. 3. Data Flow Implementation

One of the virtual interfaces is associated with the control point software on the NP. The NP service library provides a control API that allows, among other things, classifier rules and forwarding table entries to be maintained. This service library uses message-passing to communicate with the control point on the NP card. Messages are sent and received through the virtual control interface, to which they are forwarded by the NP device driver.

Boosters are not aware of how the network processor is integrated into the system or what type of network processor is used. If no NP is present, then the operations are carried out by the Linux kernel. From an application point of view, there is no difference in functionality, only in performance. Our performance measurements show that the NP forwards packets in $15\mu\text{s} \pm 3\mu\text{s}$ virtually independent of the offered load, the packet size and the number of classifier rules. In contrast, we found that Linux kernel forwarding on a Pentium IV, 1.6 Ghz processor has a latency in the range of one to several hundreds of microseconds, depending on the packet size and the offered load. In addition, the NP is able to forward packets at the line rate of 1 Gbit/s, whereas software based forwarding in Linux is able to sustain only an order of magnitude less.

Boosters use an API to access the booster library and to invoke asynchronous operations. The asynchronous operations' semantics allows applications to trigger an operation and process the results later. This model allows the high degree of parallelism needed by the boosters to avoid blocking on network operations. For example, an application might want to redirect packets and at the same time wait for control messages on a socket. Using the asynchronous redirect and read operation, both can be triggered and the results will be delivered as events. In the case of the redirect operation, redirection events will be

triggered until the application decides to stop the redirection.

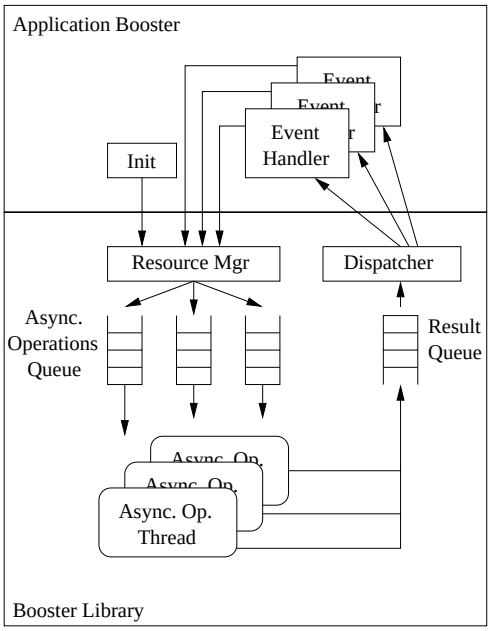

Fig. 4. Booster Library

An overview of the booster library is shown in Figure 4. Asynchronous operations are triggered by the application booster during the initialization as well as during the execution of event handlers. A resource manager resolves conflicts of operations that compete for the same resources. Competing operations are handled using a FIFO strategy and put into the same asynchronous queue. These queues are handled by a pool of threads that execute the actual operations. Results produced by asynchronous operations are put into a result queue from which they are dispatched to the event handlers of the application booster.

Figure 5 shows the steps in redirecting a packet. **(1)** the application registers a callback function with a redirect rule. **(2)** the library checks with the controller if the rule is allowed. **(3)** the rule is mapped to the format understood by the NP and **(4)** to the Linux kernel. **(5)** a packet arrives conforming to the rule. **(6)** it is directed to the Linux kernel by the NP and then to the application **(7)** by the Linux kernel. **(8)** the application callback function is executed.

III. Applications

In this section we describe our experience in using the architecture described in Section II to enhance different classes of applications, namely, sensor applications, peer-to-peer file sharing systems and massive multiplayer online game.

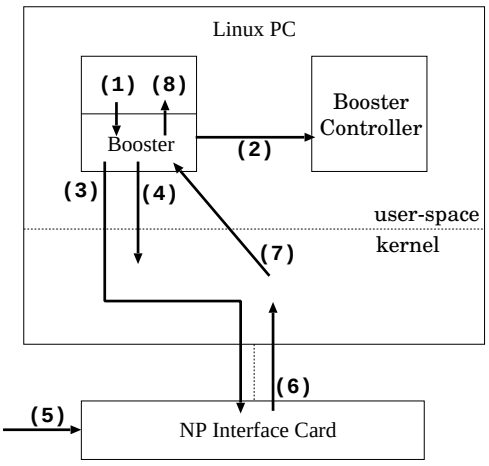

Fig. 5. Steps in Redirecting a Packet

A. Sensor Applications

The decreasing cost of processing and communication means that simple devices can be equipped with sensors capable of gathering information about their environment. These sensors transmit this data to agents typically resident on the devices themselves and which are able to process the raw data into useful information. Current work [5] is investigating cheap sensors dropped into a localized area and communicating via wireless technologies, e.g. bluetooth, to form a so called “sensor network.”

In certain circumstances the area of interest is too large to be handled by such local sensor networks and additional information, e.g. historic data, not available to the local sensors is needed to interpret the sensor data. In such cases the information can be forwarded to a central server.

Modern cars are equipped with both the means to monitor their own electronics and the ability to communicate with external agents. The set of protocols, services and data formats is termed Floating Car Data (FCD) [6]. In particular, vehicles can be used as probes to measure various aspects of the city, enabling new applications that process the data to generate useful information, for example dynamic urban traffic forecasts.

As pointed out by Estrin et al. [5], the disadvantage of using central servers in vehicle information systems is that they do not scale as the number of vehicles grows. For a high enough packet arrival rate, the server starts losing packets and the behavior of the application degrades. On the other hand, using a pure sensor network approach for urban traffic prediction is also not the ideal solution. The amount of information exchanged among cars will grow as a polynomial function of the number of cars. As the size of the city and the number of cars grow, the amount of information that can be exchanged will reach a limit causing the traffic prediction to be localized or inac-

curate.

In a sensor network, sensors aggregate received data before transmission, thereby reducing the total amount of data that needs to be transmitted, e.g. a sensor that receives a temperature reading from a set of neighbors may calculate the average temperature including its own reading before further transmission. The set of techniques used to reduce the amount of data or increase its accuracy, e.g. using probabilistic reasoning, is termed *data fusion* [7] within the sensor community.

The booster box is a platform on which functions analogous to those used in data fusion applications can be implemented and deployed in the Internet. Our proof-of-concept application is a prototype of the infrastructure needed for urban traffic prediction.

The prediction model is an extension of that used by Peytchev et al. [8]. Their model uses two state variables: queue length at the road's exit traffic light and traffic density in the road. These are modeled as a set of discrete-time linear difference equations, such that the value at time k of the state variable only depends on the value at time $k-1$. The values of the state variable for a given road are affected by only those roads with which it shares a junction. The model requires knowledge of the discharge rates at each traffic light and of the probability of car turning into each outgoing link at an intersection. The results of one iteration can be applied in the next in order to obtain predictions an arbitrary number of time cycles into the future — albeit with decreasing accuracy.

We extended the model such that we could obtain predictions of the traversal time across roads. Knowing the likely queue length and traffic density at some number of traffic cycles in the future, the likely traverse time is calculated. Traverse time has two components. The first is the time it takes to drive through the road calculated from the average speed and length of the road. The second component is the time the car will be stuck in the queue before it gets discharged. We measure the average speed of a car on a road using the information transmitted from the cars.

Figure 6 shows the various components in the experimental setup. We used a city simulator to simulate the behavior of 1000 cars moving around a small city. Packet generators read data from the output of the simulation and generate UDP packets containing the position and velocity of an individual car and using the same format as that sent by the cars themselves. In addition, traffic lights also communicate their state changes. Different packet generators are assigned to different parts of the city. These packets are intercepted by the booster boxes, in the way described in Section I, and their information is aggregated in a format suitable for the prediction model. At every traffic light state change on a given road, the booster box communicates the queue length and the traffic density to

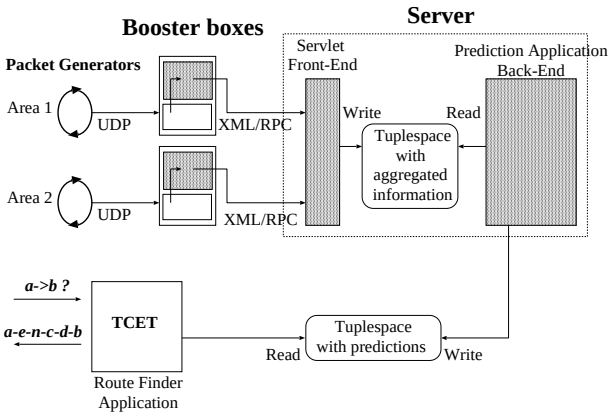

Fig. 6. Aggregating Sensor Information

the server. In the case that packets are not intercepted by a booster box, the server is also capable of accepting and combining the raw data from a car with the aggregated information.

In addition to aggregating information at the booster box, we format the information such that the packets forwarded to the server use XML/RPC over a long lived TCP connection. This has two advantages, first, it allows us to use standard server technology to implement the backend server, e.g. Apache httpd associated with an IBM WebSphere servlet engine, and second, the processed information can be transferred reliably over the network in a TCP friendly way.

The server calculates the prediction using the model described, and outputs to a tuplespace. The tuplespace plays the role of a broadcast medium. In a real application the broadcast medium can be for example Digital Audio Broadcast (DAB). A routing application running on the in-car computer platform — the TCET in Figure 6 — periodical reads from the tuplespace to calculate shortest time paths across the city.

In our experimental setup, each of the two data generators was connected to the server through a booster box. Each booster box was implemented on a Linux machine equipped with a Pentium II 233 MHz processor and 64 MB of RAM. The server also ran Linux but with a 1500 MHz processor and 1024 MB of RAM. The network used to interconnect the devices was a 100 Mbit ethernet LAN.

Figure 7 shows how a server scales with and without booster boxes. The graph plots the number of UDP packets received as a function of the number of packets sent. The measurement was performed at the server when there is no booster box and on the booster boxes themselves when they are used.

The experiment without booster boxes started to lose some packets at a load of less than 2,000 p/s and had losses of more than 50% at 5,000 p/s. The experiments

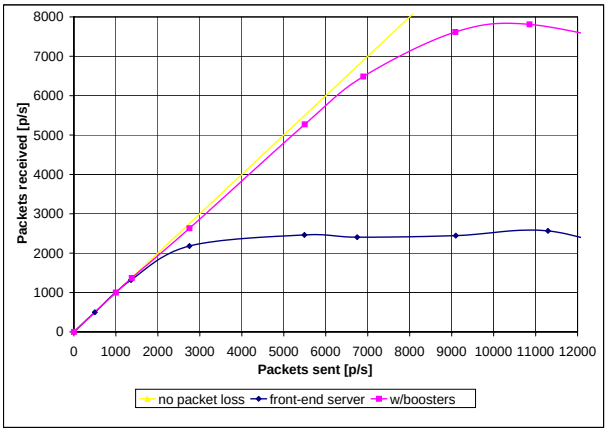

Fig. 7. System Performance and Packet Loss.

with booster boxes showed no significant loss until the load was higher than 6,000 p/s, and at 11,000 p/s the loss was about 25%. At the same rate the scenario without booster boxes had nearly 90% loss.

The machines used in the test environment are much less powerful than those that would be used in a real system; therefore the absolute figures for load handled are not truly representative. Of more interest is the gain achieved by using the aggregation function on booster boxes. Note that these booster boxes were eight times slower than the server. The results showed that in the test environment the use of the booster boxes reduces the amount of packets the server has to handle by a factor of 10. Although the amount of data transferred is the same, better performance is observed as the bottleneck is the number of packets that can be handled, rather than the throughput. A detailed analysis of the experiments and the results can be found in [9].

B. Peer-to-Peer File Sharing Applications

Peer-to-Peer file sharing applications such as Gnutella [10] build an overlay network of peer nodes across the physical network and then flood requests over this overlay. This approach has a well known scalability problem [11], due to the incongruity of the physical and overlay network topologies, leading to inefficient request forwarding, and the inherently unscalablility of flooding.

Solutions to these problems have been proposed. For example, Ratnasamy et al. [12] propose the use of an Internet coordinate derived from the distance to well known beacons to determine network closeness, while others [11], [13] propose dividing the responsibility for storing files across the set of peers based on file names, such that a peer can forward requests in a more guided way. Within the Gnutella community the scaling problem has led to the introduction of the concept of an ultra-peer in which one of the peers in an area is denoted an ultra-peer, and a two level hierarchy is created such that leaf-peers connect to ultra-peers and ultra-peers connect amongst themselves. The ultra-peers gather information about attached leaf-peers, e.g. asking them for the entire set of files they have, and exchange this information with other ultra-peers in order to allow more guided file query requests.

The disadvantage of these techniques is that they require the development and deployment of new and more sophisticated peer-to-peer file sharing systems. It is debatable whether the fact that these systems make better use of network resources than existing ones is a sufficient reason for their wide scale deployment. A better solution is one in which existing systems are made to conform to better practice without having to change the software running on clients' end-stations. This can be achieved if the network can intercept and manipulate file sharing operations.

The basic principle of our approach is to place booster boxes at the ingress/egress links of a cluster, e.g. an administrative domain. The booster box intercepts connection request messages issued by peers and "convinces" those peers to connect to itself. The booster then becomes the gateway with which all external clients must peer and through which all intra-cluster requests for files must be carried. If multiple booster boxes are scattered throughout the network, they can uniquely or preferentially peer with each other, creating a two level overlay. As the booster to booster communication can use proprietary protocols for exchanging information, they can use techniques such as those described in [13]. Better scalability of the peer-to-peer file sharing systems is achieved because the topology of the overlay can be controlled and requests can be more intelligently forwarded, without having to change the file sharing system software running on the end-systems.

Our proof of concept uses the Gnutella file sharing system. At the start of a Gnutella session, each Gnutella client attempts to peer with some subset of Gnutella peers learned about by other means, e.g. obtained from a well known server. The Gnutella client opens a TCP connection to the remote client and then sends a *Gnutella Connect* message to ensure that the remote client is running Gnutella; if it is, the remote client sends a *Gnutella OK* message. A Gnutella client can probe the network for additional peers by sending Gnutella *ping* messages to existing peers, and these in turn will forward the *ping* message to all their peers; this continues for some number of hops across the overlay. A client receiving a *ping* and having an interest in peering replies with a *pong* message containing the address and port of itself, which is carried back across the overlay.

We implemented a booster that intercepts Gnutella traf-

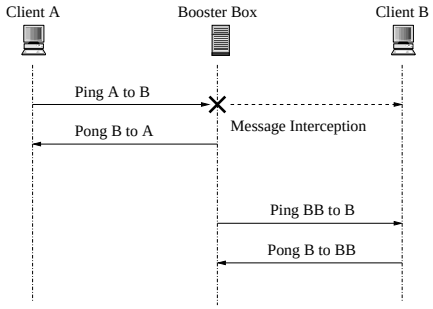

Fig. 8. Interception of Gnutella Control Traffi c

fic identified by the well known Gnutella port [1]. This booster intercepts and accepts the Gnutella TCP connection request from client's in its local domain and acts as it were the designated remote Gnutella client. The local client may send *ping* messages across this connection, in which case the booster may simply drop them or reply with *pong*s containing the addresses of other clients in the local domain with which it has already connected. Figure 8 illustrates the message exchanges. When *Client A* issues a *ping* to *Client B* the message is intercepted by the booster box, which replies with a *pong* as if it were issued by *Client B*. Depending on the logic used to form the overlay network, the booster box may or may not connect to *Client B*.

The result is that if the booster box is placed after the unique access router of a given domain, then all hosts within that domain can peer only with other hosts in that domain or with the booster. The booster itself can peer with other Gnutella clients outside the domain in the conventional way. The booster makes the topology of the overlay network more congruent with that of the physical one. Figure 9 depicts an example of this principle. It shows how booster boxes inserted in the three clusters minimize the connection on the links between clusters. The booster waits until it has received a given Gnutella *query* message multiple times before forwarding it outside the domain boundary, ensuring that a first attempt is made to find the file locally before searching for it remotely. More advanced techniques can be envisioned, e.g. creating indexes of hosts with given files, caching commonly requested files etc.

The novelty in this approach is not the better file sharing techniques per-se, but rather the ability to efficiently extract identifiable peer-to-peer messages from gigabit streams and then apply those techniques to them. An important benefit deriving from this approach is the dramatic reduction of the traffic on clusters' ingress/egress links.

[1] If Gnutella clients use ports other than the standard one then identifi cation of a Gnutella request can be achieved by looking at the payload. This is possible using a programmable NP.

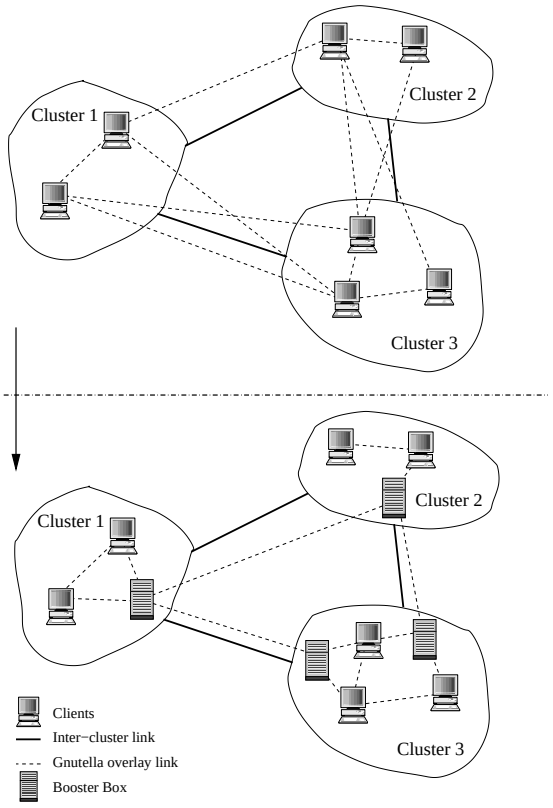

Fig. 9. Clustering the Gnutella overlay network

C. Massive Multi-Player On-line Games

The term Massive Multi-Player On-Line Game (MMPOG) has come to denote games with a large number of participants played over the Internet, for example the EverQuest role-playing game [14]. Current MMPOGs use a central server approach as peer-to-peer systems cannot scale to the required number of players and centralizing the logic in a single game server farm eases the problem of administration. The latter is important as the game provider charges clients for use of the game so security and resilience are essential.

The number of simultaneous participants a game can host is dependent on the amount of resources available on the server farm, in particular the server I/0, network bandwidth, processing cycles and memory. How exactly these grow with number of participants is game dependent, but as the games involve interactions between participants this is almost certainly not less than linear. In consequence to increase the number of simultaneous participants by a factor of ten, at least ten time more resources are needed.

As the infrastructure must be built before the popularity of the game is known, this represents a large risk for the game provider. The game may be more popular than expected leading to congested infrastructure and perhaps

even game failure, or the game is less popular and the infrastructure is massively over-provisioned.

Better server farm scalability could be achieved if some of the computation could be delegated to the participant's computer. The total available computation to the game would increase as with the number of participants. Peer-to-peer games have this property, however, they are only applicable to a small number of participants — as the number of packets sent grows as a square of the number of participants — or require communication over a broadcast medium, e.g. a LAN. It is also difficult to envisage how a pure peer-to-peer system could be made reliable and secure enough.

Our approach is to use a hybrid of the peer-to-peer and central server model to obtain scalable but reliable MMPOGs. We achieve this by:

- dividing a large game into multiple federated small games each of a size that can be handled using a peer-to-peer model;
- using fast dedicated multicast reflectors which handle the communication for one or more of the small games;
- separating the control and data planes, such that the multicast reflectors are responsible for the fast data forwarding, while servers continue to perform the control and management functions.

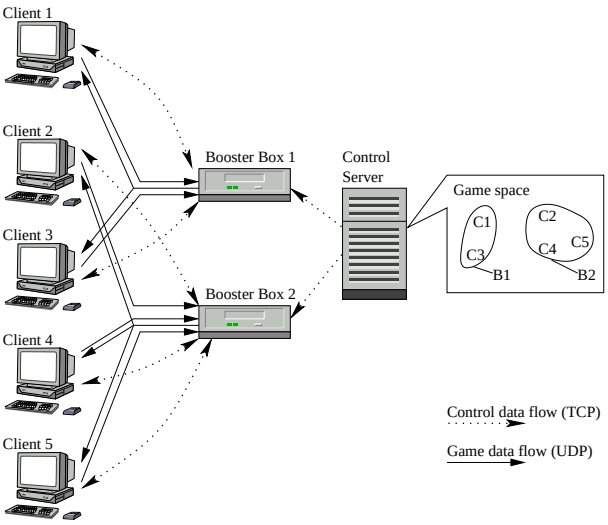

Fig. 10. Hyrbid Game Architecture

At registration, clients obtain the virtual location / booster box mapping from servers. They add and remove themselves from multicast groups as they move within the game. For reasons of space we limit ourselves in this paper to a description of the data path. The architecture is shown in Figure 10.

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. Booster boxes may be all located at the providers server farm, or scattered throughout the network. While the first is easier to install and manage, distributing booster boxes throughout the network allows better scalability — as all the data traffic is not funneled to the same network access point — and is more resilient.

The booster box implements the multicast reflector 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. The booster box in effect acts as a LAN for the virtual location. We implement this multicast function using the multicast support offered by the NP.

UDP is generally preferred over TCP in games, however, the application developer must handle retransmission at the application level. As well as supporting the basic communication within the virtual location, the booster box also offers support to the application for retransmission in the case of loss. The application uses application-specific knowledge to determine if retransmission is desirable or not.

The booster box multicasts the game packet to all players resident at the virtual location, including the sender of the packet. The application can determine if the packet were received by the booster box by the fact that the same packet is received back from it. This packet in effect acts as the equivalent of a TCP ACK and allows the protocol stack on the client's computer to determine the Round Trip Time (RTT) between client and booster box. A client can distinguish the packets it sent from others through a 32 bit packet identifier used in the header. The client chooses a random number as packet identifier when sending a packet and maintains a list of the recently sent packets. Clients also maintain the list of recently received packet in order to recognize duplicates.

The application decides whether a packet should be delivered reliable or not, if it is then after no acknowledgment has been received within the RTT Time Out (RTO) — as determined using the same algorithm as TCP — it is retransmitted. The application also sets an upper bound on how long to try retransmitting a packet. A packet that an application specified should be sent reliably is sent to the booster box with a reliable delivery flag set. When such a packet arrives at a booster box, the forwarding mechanism adds a packet sequence number and the packet is stored in a buffer on the booster box. As the sequence number increases monotonically, a receiver can determine whether it has not received a reliable 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 or even if it requests retransmission at all is application specific. A retransmitted packet is identified as such by a flag in the header. 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. Booster boxes add the range of currently retransmitable packets to each packet sent to the client. As a client cannot distinguish inactivity from high levels of loss, if no packet has been sent within a virtual location for a certain period a "heart beat" packet is sent to all clients in a location, repeating the sequence number sent last and the lowest (oldest) available one it has in memory. The heart beat carries no data and no packet identifier.

Games because of their logic are implicitly rate-based, i.e. the packet sending rate will not grow in an unbound way, but in general is limited by the speed of human interaction, as such we do not see the need for any flow control in the basic data sending rate — although an application could instrument such an mechanism on top of the infrastructure if it thought it necessary. Retransmissions are not restricted by human interaction and as such require special consideration. It is possible that requests for retransmission become correlated, for example if the booster box, or the network close to it, is subject to disruption. There is a danger that the requests for retransmission may themselves cause more loss, which causes even more retransmission. TCP congestion control probes the effective throughput of the network by adjusting the sending rate as a function of the acknowledgments of successfully transmitted packets. However, slow-start is not appropriate when the timeliness of delivery is essential, instead we allow a client to ask for retransmission of any missing packets, but require the reception of all missing packets or the expiration of a timeout before further request can be made. The timeout is initially set to the RTO, but is exponentially increased at expiration. In this way, we avoid the problem of positive feedback at the cost of reliable delivery.

Figure 11 summarizes the fields of the booster header encapsulation. The use of the various fields have been explained in the previous description; the RTO is used by the booster box in the calculation of the heart beat interval, as it is also received by all other clients, it may be used by application logic.

The booster box behaves similarly to the *master* component of the "Multicast Transport Protocol" (MTP) described in [15]. However, MTP offers a richer set of functions, in particular a rate-based flow-control mech-

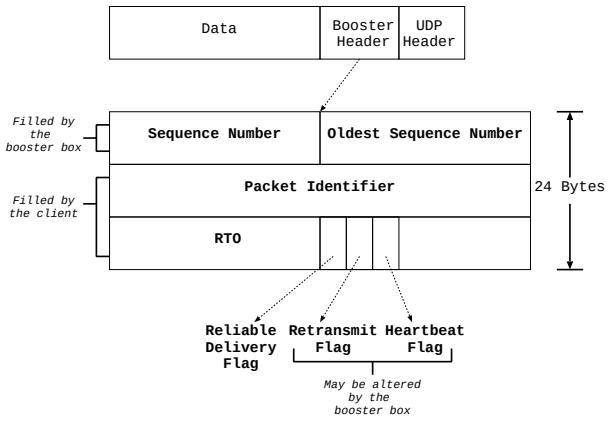

Fig. 11. The Booster Data Path Encapsulation

anism 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. [16] 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; congestion avoidance is not considered.

Figure 12 shows some of the most important data path operation scenarios. The evolution of the client's game state is denoted by *Si*. In the first case the client sets the packet identifier *a* and the booster box adds the sequence number *1*. In the second case the packet is lost on the path from the sender to the booster box, the client resends the packet after the timeout. In the third the packet is lost from the booster box to the sender, the sender retransmits the packet, with the same identifier *c* and the booster adds a new sequence number, all the clients receive the new packet, but recognize it as being a duplicate due to the packet identifier and ignore it. The sender may also ask for the retransmission of the packet from the buffer due to the fact it is missing sequence number *3*, but will ignore it when it arrives. In the last case a packet is lost on the way to the receiver. In the scenario shown, the next packet sent is a heartbeat packet, although a data packet would serve the same purpose, the receiver realizes it is missing packet *5* and asks for its retransmission.

In summary, the booster box offers a means for reliable delivery of packets within some time period, if both sender and receiver think it is worthwhile.

IV. Related Work

Packaging multiple configurable middlebox functions in the same network element allows ISPs to reduce the

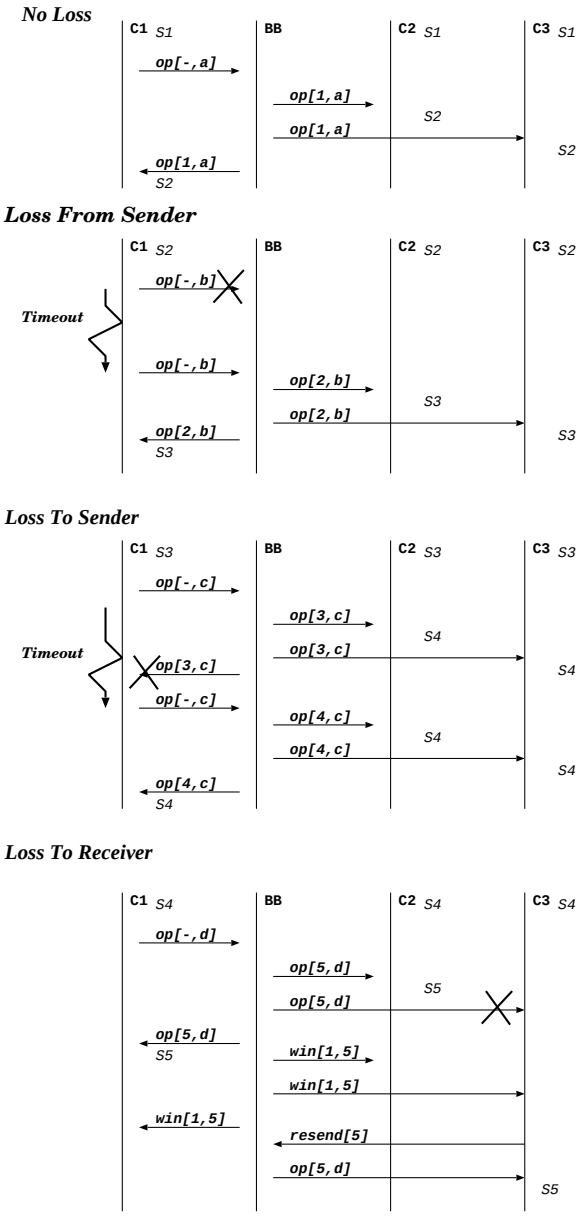

Fig. 12. The Data Path Operations

number of distinct middleboxes and thereby decrease costs. An example of a commercial configurable middlebox is CoSine's IPSX switch [17]. The IPSX is a middlebox divided such that each subscriber is allocated a separate virtual router upon which a range of distinct middlebox function, e.g. NAT, content filtering, IPSec etc, can be applied. The IPSX shows the importance of offering higher level services in the network, however its focus is on the efficient consolidations of existing functions, for example using ASICs, rather than the deployment of entirely new ones.

The model of computation whereby a client owns the computing resource on which they run their processes has been the dominant one for more than two decades. Recent work has challenged this model. Within the field of outsourcing, instead of a client purchasing the entire set of servers on which their software is run by the outsourcing agent, economies of scale can be achieved by allowing the same set of servers to be shared between clients, for example [18]. While in the field of scientific computing the Grid [19] is an initiative to formalize this pooling of resources by creating virtual organization abstractions, which are formed from a grid of pooled computers and exist for the duration of an activity. Current OS research is investigating the means by which the same network based server can be securely and efficiently shared between many different clients e.g. [20], [21] — typically through the creation of virtual machines — and how distributed virtual machines can collaborate together to form service specific networks [22]. Our work focuses on how application logic can make use of network specific hardware in order to better scale certain types of large distributed applications. That said, if resource control mechanisms were required between boosters then techniques such as those described in [20] could be applied.

Srisuresh et al. [1] propose a framework for allowing middleboxes to be configured by trusted third parties. In this framework the application specific code is instrumented as a *midcom agent*, either running on the middlebox itself or communicating with it across the network. It is similar in spirit to the work described here, but limits itself to control path functions. For example, in the creation of an IP telephony session, the SIP messages are handled by the midcom agent, whereas the RTP/RTCP messages are handled by the middlebox itself.

V. Conclusion

The popularity of hypertext documents led to the need for specific network infrastructure elements such as HTML caches and URL-based switches. We contend that a range of new, large, and distributed applications will have a similar impact on the Internet and will require similar dedicated support. This paper has outlined some initial work on prototyping a middlebox capable of assisting in the scaling of these applications. As these applications are rather diverse, and because of their novelty difficult to standardize, we have chosen a programmable approach, in which different application-specific pieces of logic can be instrumented on the same platform.

References

[1] P. Srisuresh, J. Kuthan, J. Rosenberg, A. Molitor, and A. Rayhan, "Middlebox communication architecture and framework," Request for Comments 3303, IETF Network Working Group, Aug. 2002.

[2] Niraj Shah, "Understanding Network Processors," Tech. Rep., University of California at Berkeley, Sept. 2001, http://www-cad.eecs.berkeley.edu/~niraj/web/research/network_processors.htm.

[3] D.C. Feldmeier, A.J. McAuley, J.M. Smith, D. Bakin, W.S. Marcus, and T. Raleigh, "Protocol Boosters," *IEEE JSAC, Special Issue on Protocol Architectures for the 21st Century*, vol. 16, no. 3, pp. 437–444, Apr. 1998.

[4] Brian Carpenter, "Middleboxes: Taxonomy and issues," Request for Comments 3234, Internet Engineering Task Force, Feb. 2002.

[5] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar., "Next Century Challanges: Scalable Coordination in Sensor Networks," in *Mobicom, Seattle, USA*, 1999.

[6] ISO, "Transport information and control systems reference model architecture(s) for the tics sector, iso/tr 14813-1:1999," Tech. Rep., ISO, Standards of TC 204, 1999.

[7] Lawrence Klein, "Sensor and Data Fusion Concepts and Applications," in *SPIE Optical Engineering Press TT14*, 1993.

[8] E. Peytchev, A. Bargiela, and R. Gessing, "A Predictive Macroscopic City Traffi c Flows Simulation Model," in *In Proceedings of European Simulation Symposium ESS'96, Genoa, 2, pp. 38-42.*, 1996.

[9] Annie Chen, Navendu Jain, Angelo Perinola, Tadeusz Pietraszek, Sean Rooney, and Paolo Scotton, "Experience in Building Scalable Real-Time Telematics Applications," *Available as IBM ZRL Research Report*, Dec 2002.

[10] "The Gnutella Protocol Specifi cation v0.4," `http://www.clip2.com\slash{}GnutellaProtocol04.pdf`, Document Revision 1.2.

[11] Antony Rowstron and Peter Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Nov. 2001, pp. 329–350.

[12] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker, "Topologically-Aware Overlay Construction and Server Selection," in *IEEE Infocom*, June 2002.

[13] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker, "A Scalable Content Addressable Network," in *Proc. ACM SIGCOMM*, Aug. 2001, pp. 161–172.

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

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

[16] David Levine, Bart Whitebook, and Mark Conway Wirt, *A Massively Multiplayer Manifesto*, Butterfly.net, Inc., 123 East German St. Shepherdstown WV 25554, May 2002, Version 1.1.

[17] CoSine Communications, "IPSX Service Processing Switch Family," Data sheet, Cosine Communications Inc, 2002.

[18] Sean Rooney and Anthony Bussani, "Client Delegated Control within an ASP Infrastructure," *Journal of Communications and Networks*, vol. 3, no. 1, Mar. 2001.

[19] Ian Foster, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Lecture Notes in Computer Science*, vol. 2150, 2001.

[20] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble, "Scale and Performance in the Denali Isolation Kernel," in *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, Dec. 2002.

[21] K. Fraser, S. Hand, T. Harris, I. Leslie, and I. Pratt, "The XenoServer Computing Infrastructure, a project overview," *Cambridge University*, 2001.

[22] L. Peterson, D. Culler, T. Anderson, and T. Roscoe, "A Blueprint for Introducing Disruptive Technology into the Internet," in *In Proceedings of the 1st Workshop on Hot Topics in Networks (HotNets-I), Princeton, New Jersey, USA, October 2002.*, 2002.