HyperNet Games: Leveraging SDN Networks to Improve Multiplayer Online Games

Shufeng Huang
Laboratory for Advanced Networking
University of Kentucky
Lexington, KY 40506
Email: shufeng@netlab.uky.edu

James Griffioen
Laboratory for Advanced Networking
Univerity of Kentucky
Lexington, KY 40506
Email: griff@netlab.uky.edu

***Abstract*—Many of today's real-time multi-player online games are highly dependent on the communication network that connects players with the game and each other. Factors such as network latency and network throughput can significantly affect a user's gaming experience. For example, many online games use a (logically) centralized game server that acts as a rendezvous point for players to exchange real-time gaming events. The number and location of these servers in the network plays a critical role in the interactive response time of the game. Currently, gaming companies go to great lengths to find the best locations for the placement of their game servers. However, finding the best locations is challenging because the number of players and their locations is not known ahead of time. Consequently, game companies typically build a generic (static) game network that hopefully will provide acceptable performance to most users.**

With the growing popularity of cloud computing and software defined networking (SDN), it is now possible to build a custom game network on-the-fly; a game network tailored for a particular game and its current participants (players). In this paper, we introduce the concept of a *HyperNet Game* that is capable of dynamically creating and deploying a *software defined network* tailored to the needs of the game and the current set of participants in the game. We present an example multi-player hypernet game that we have developed that is able to dynamically deploy optimally placed game servers based on the current set of players. We present performance results that show greatly improved user response times compared to the currently deployed (static) game server network used today.

I. Introduction

Realtime multiplayer online games have become a prominent part of the gaming market. Today, nearly all console games come with an "online pass", that allows you to play the game online with other network players. Moreover, the rapidly expanding mobile platform game market (i.e., cell phone games) is experiencing a similar trend toward online multiplayer games. Realtime multiplayer games let you interact with a group of people, either acting as your teammates or your opponents, in a digital world. Fixed-duration games allow you to "play" with a group of players for the duration of a "game" – typically defined in units of a "set" or a "match". Examples of such games include *WarCraft [1]* and *Defense of the Ancients(DotA) [2]*. Continuously-running games, sometimes referred to as "massive multiplayer online games", allow users to continually join and leave an ongoing game where the set of players in massive and constantly changing. Example of such games include *World of WarCraft [3]* and even alternate realities such as *Second Life [4]*. While the focus of this particular paper is on fixed-duration games, similar techniques can be applied to address the dynamically changing membership of massive multiplayer games as well.

One key requirement of all real-time multiplayer games is consistency. That is, all players must share the same view of the digital world. To achieve consistency, game companies typically use a (logically) centralized game server to maintain a consistent view of the game across all players. The game server acts as a rendezvous point for all game events generated by the players (or by the game itself).

A key aspect of consistency in a multiplayer game is *delay*. That is, the time between a player performing some action and the time that action appears on all other players' screens. To minimize game delay, game companies carefully choose the network location for their game servers. In most cases, game servers are statically deployed before the game begins. However, finding the best locations for game servers is challenging because the number of players and their locations is not known ahead of time. Consequently, game companies typically build a generic game network that hopefully will provide acceptable performance to most users. One technique that companies use is to divide the Internet into geographical regions and deploy some number of game servers in each of the regions. The game then directs players to the game server in their region. For example, in WarCraft III [1], players are allowed to pick from four regions: "U.S. East", "U.S. West", "Asia" and "Europe". Deploying multiple servers in different regions has its apparent benefits: it is obvious that a player from California will get lower network delay (and thus, better gaming experience) if connected to a server in "U.S. West" region than in "U.S. East" region. If a sufficient number of game servers are deployed, all game players will be able to find a game server that is relatively close to them and will observe low network delay resulting in smooth game play. While this model has its advantages, it is far from optimal. First of all, in order to maintain the low delay between players, these systems often segregate players. Players connecting to different servers will not be able to see each other in the game, simply because they are using different servers. For example, a player in California will not be able to play with a player in New York because they will be directed to different servers ("U.S. West" for California and "U.S. East" for New York). If two

This work is supported in part by the National Science Foundation under grants CNS-0834243 and CNS-1111040

978-1-4799-0820-2/13/$31.00 ©2013 IEEE

players want to be in the same game, they have to choose the same server (in this case, either "U.S. East" or "U.S. West"), which favors one player over the other in terms of network latency. Secondly, because the "game network" is established (statically) before the game begins, it cannot adapt or optimize itself to the current set of players. If the current set of players are located near one another (say, in a university campus), they will be directed to connect to one of the pre-established Internet game servers, thereby causing all their traffic to go to the Internet when there are much more direct paths connecting them. The inability to dynamically create a game server and direct users to it, or to dynamically move a game server around based on changing game membership prevent the game from offering network performance optimized for the current set of players[1].

Emerging *Software-Defined Networking (SDN)* technology offers a radically new way to deploy networks. Sometimes referred to a *programmable networks*, SDN technology [5], [6] allows applications running on end systems to dynamically control the network topology, connectivity, routing, and possibly quality of service (QoS). Some SDN providers [7] even allow the users to "program the network" by dynamically loading and running their own application on network routers. In addition, emerging cloud computing service providers [8], [9] offer the potential to dynamically obtain significant processing and storage resources at a wide range of locations across the Internet. Together, SDN technology and cloud computing make it possible to *dynamically* stand up a server anywhere in the network and then create a highly optimized network that connects that server to the other network participants (game players). In other words, the emergence of SDN networks now makes it possible to create a custom game network, optimized for a specific game with a particular group of players and lasting for only the duration of the game. For example, a game network may place the game server in an optimal network location such that it provides the minimum average delay to and from all game players in the current game. The network connections between the server and all players may also be optimized such that they provide bandwidth guarantees to ensure a particular level of play is possible. The network routers in between the server and players may be configured such that they favor critical game packets (e.g., a shooting action) over non-critical packets (e.g., background texture) when faced with network congestion. Moreover, the ability to configure or program SDN networks facilitates the ability to create advanced networks that offer services not currently (or commonly) available today. For example, a game server network may be designed to send a single multicast flow instead of multiple unicast flows because the special purpose SDN network was set up to support multicast – a feature not widely available in the current Internet. As an other example, the game server might be designed to allow "server migration" so that it is possible for the SDN network to migrate the game server to a better network location on-the-fly when the game players change.

In this paper, we present a new model, called *HyperNets*, that enables users to easily deploy and use custom SDN networks that are designed for a specific purpose (e.g., real-time multiplayer games). Inspired by the concept of a virtual appliance [10], HyperNets bundle together the network topology configuration, software, and network services needed to create and deploy a custom SDN network.

In what follows, we briefly describe SDN networks and then present our new HyperNet architecture in Sections II and III, with a detailed description of the Network Hypervisor service. In Section IV, we describe the usage models of our architecture. We then present a SDN game network deployed via the use of our OpenArena Game HyperNet in Section V. Finally the paper concludes in Section VI.

II. SDNs and Cloud Computing

Virtualization of compute, storage, and network resources has radically changed the way in which we design and implement new services. Moreover it has significantly reduced the timescales on which we can deploy new services. For example, it is now possible to purchase virtual machines (and virtualized storage) from companies like Amazon (e.g., EC2 [8] and S3 [11]) and Microsoft (e.g., Azure [9]) as well as from and an ever-growing number of cloud providers. They can be allocated, initialized, and used on small timescales to support application-specific services. The number of resources used can be enlarged or shrunk over time as demand for the service changes.

Although virtualization of cloud resources has been in use for some time now, a relatively new addition to the virtualization scene is the virtualization of network resources (e.g., network routers and the links that connect them). It is now possible to dynamically allocate virtual routers and virtual links to create an application-specific network on-the-fly. This technology is often referred to as *software defined networking (SDN)* or *programmable networks*. Examples of SDN technology include OpenFlow switches [5] which allow an external program called an OpenFlow Controller to load application-specific forwarding rules into a set of OpenFow switches to dynamically control the paths taken by an appliction's data. Much like various cloud providers have arisen over time, we are beginning to see SDN providers emerge where applications can go to reserve SDN-enabled network resources. Many university campuses are becoming SDN-enabled, and corporations are increasingly looking to SDN as a way to offer new network-layer services. The GENI [12] research network offers users the ability to reserve a "slice" of the network consisting of (programmable) virtualized network routers and links. Companies such as Internet2 are beginning to offer support for the GENI API's as well as OpenFlow SDN. Other companies such as AT&T are also building SDN-enabled networks such as Shadownet [6] in order to be able to quickly roll-out new services.

In short, it is now possible to dynamically construct (on demand) application-specific networks, complete with in-network processing and storage. As a result, one can now consider dynamically deploying a highly-optimized game network specifically designed for the needs of a particular game and its current set of users (players).

[1]While some games will allow users to deploy "local/private" servers, these servers must be setup/run by the players as opposed to the game operators, are less stable than game operated servers, are not easily moved or reconfigured in response to changing membership, and typically suffer form the "can see only local players" problem.

III. The HyperNet Architecture

Consider the problem of dynamically deploying a centralized game server that attempts to minimize the network latency (delay) of all players (i.e., a player-specific game server created just for that set of players). Given a set of players, the goal is to find a location in the network where the game server should be instantiated to minimize delay. Once instantiated, the game needs to set up optimized network paths from each of the players to the newly instantiated game server.

While SDN-enabled networks have the potential to completely change the way players and games communicate, setting up an application-specific SDN network (such as the player-specific minimum-latency game server network described above) is by no means easy. What is needed is a higher-level abstraction/service that helps game developers create application-specific SDN networks while hiding the underlying details of the SDN network from the game developer.

To address this need, we have developed a new abstraction called a *hypernet*. One can think of a *hypernet* as a downloadable program that can be "run" to dynamically create an application-specific SDN network. The hypernet includes the logic needed to select the resources needed by the application-specific network. It also contains the software to be run on the various nodes in the SDN network (e.g., the game-server code that must be started on the – dynamically discovered – centralized node).

The concept of a hypernet is modelled after that of a *virtual appliance* [10]. Much like a virtual appliance specifies the configuration of a virtual machine and includes precisely the software needed to implement a particular application or service, a hypernet is a complete software package that encapsulates all the expertise necessary to create an SDN network and deploy the software to be run on the SDN. For example, a hypernet designed for a particular game would contain a program to determine the best network topology given the current player set, the software protocol stacks to run on the end systems, the code and configuration files needed to run the game server on the server node, the specific router processings to use (e.g., routing tables, multicast, priority queues, etc), and the rules to load those configurations and software onto the corresponding nodes. Because the hypernet encapsulates all the expertise and code, users simply need to "run" the hypernet to create a highly-optimized network for their game.

Just like a virtual appliance runs on a virtual machine hypervisor [13], a hypernet runs on a *network hypervisor*. Figure 1 shows the proposed hypernet architecture. Hypernets use a set of API calls provided by the network hypervisor to setup and create the SDN network. Based on the API calls made by the hypernet, the network hypervisor performs the task of obtaining network resources from SDN providers, connecting resources together to form the topology required by the hypernet, loading the necessary software and/or configuration files onto network nodes, and then adapting the topology over time as the participants come and go. Thus, the network hypervisor acts as a broker between the various hypernets ("running" on the network hypervisor) and the SDN providers. Note that a network hypervisor can be used to support other types of specialized SDN networks in addition to real-time multiplayer games. In fact, at present we have designed/implemented a variety of different hypernets as illustrated in Fig 1.

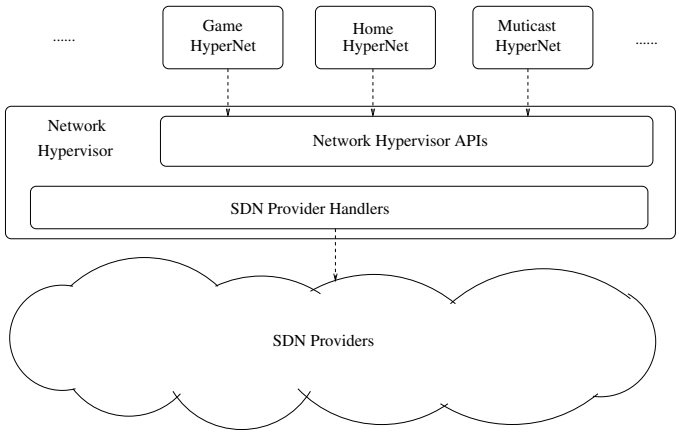

Fig. 1: The HyperNet Architecture

A. The Network Hypervisor API

The network hypervisor supports a set of API calls specifically designed to make it easy to create application-specific SDN networks. In particular, the API calls allow a hypernet to easily (1) specify the topology, (2) program/run code on SDN routers, and (3) enable participants to join the SDN. For a detailed description of all hypervisor API calls, please refer to [14]. For the purpose of this paper, we will focus on the following four API calls: *findPR(), findCentralNode(), loadApp(), join()*.

The *findPR()* call finds a nearby SDN router based on the network location of a participant (i.e., its IP address). To deploy an SDN network that is tailored for its participants, the first step is to attach the participant to the "closest" point in the SDN network. In the case of a game SDN network, findPR() API call identifies a "connection point" to the SDN network for each player. Rather than having players use IP to reach the game server, findPR() allows players to send their data over the optimized SDN network to the game server. Only the "tunnel" between the player and its "connection point" goes over IP where the path and QoS cannot be controlled. The remainder of the path between a player and the game server traverses the SDN network which can be optimized and provisioned to provide optimal performance.

The *findCentralNode()* API call finds the SDN node that is centrally located relative to a set of participants (i.e., players). This is particularly useful for identifying the best location to run the game server. The central node can be chosen based on the number of hops or the network performance from the given nodes. For example, in the case of a game network, we can find a central node that provides the minimum average network delay for all players. To implement this API call, the Network Hypervisor must obtain information about the underlying SDN provider's network. For example, to find a central node based on number of hops, the physical topology information is needed. To find the best central node based on network performance, a probing mechanism needs to be provided by the SDN provider to determine the network

distance between two of its managed nodes. In our case, we use "ping" probes to determine the Round-Trip Time (and thus, the network delay) between a potential central node and the current players. In order to avoid sending out probing messages to and from all available SDN nodes, we divide the available resources in the SDN provider network into a small number of "aggregates", and assume that all nodes in the same aggregate share the same network location and have similar networking performance. This assumption maps well onto the existing cloud service providers. In fact, both Amazon's EC2 system [8] and Windows Azure Cloud [15] distribute their cloud resources in multiple data centers. We then reserve one node from each aggregate that "represents" the network location for that aggregate. This way, we shrink the number of probe messages from millions (the number of nodes available from the SDN providers) to dozens (the number of aggregates).

The *loadApp()* API call loads an application and a configuration file onto a specified node. It also accepts a command to execute the application. Using this API call, we can configure and run the game server on the central node as well as load and run software on any other router in the SDN network.

The *join()* API call creates an IP tunnel between a participant (game player) and its associated SDN "entry node", assigns an SDN network address to the participant, and configures the routing tables properly to ensure that the participant is able to talk with other nodes in the SDN network. After a game player "joins" a game SDN network, the game client is able to talk with the game server via the IP tunnel, enabling the player to begin playing the game.

IV. HyperNet-based Games

Given the ability to easily create an SDN network optimized for a particular game and its participants, there are two ways to go about offering a hypernet-based game: (1) build a game-specific SDN network and run the existing game (unmodified) over that network, or (2) modify the game's source code to call the network hypervisor APIs itself, thereby allowing it to more effectively leverage the SDN network.

Consider a game server that must distribute game events to all players in order to maintain consistency. Because multicast is not supported in the Internet today, existing game servers typically send unicast messages to all the players to keep them up-to-date. Using the first approach, one could create a hypernet that identifies a central node and runs the (unmodified) game server on that node, sending unicast message to all players. As we will see later, this can greatly reduce the delay experienced by players even though the game server itself has not been modified. Only its location has been changed (i.e., optimally placed).

Under the second approach, however, one could modify the game server code to also take advantage of the programmability of SDNs. For example, in addition to finding the central node, one could modify the game server code to deploy multicast network services across the newly created SDN network and then could multicast game events to players, thereby greatly reducing the network traffic. However, this requires changes to the game server to send out multicast traffic as opposed to the multiple unicast it currently uses. Although we do not have space in this paper to describe how one can create a multicast SDN network, we refer the interested reader to [14] which describes how to create a wide area multicast network between participants (something that – for various reasons – is not possible in today's Internet).

V. An OpenArena Hypernet Game Example

To showcase our hypernet architecture, we created a hypernet game that automatically deploys a custom SDN network for the OpenArena game [16]. OpenArena is an open source multiplayer online game. The game has several publically accessible game servers to which participants can connect. However, being an open source game, it also allows participants to compile and run their own game servers. Consequently, it is possible for a set of players to identify the best location for a game server (e.g., a location with the lowest delay for all participants), and then run their own game server at that location.

The goal of our OpenArena hypernet game was to automatically select the best location for a game server, run the game server at that location, and then set up and SDN network to connect the game server to all the participants. For our experiment, we used ProtoGENI [17] as the underlying SDN provider. ProtoGENI is an implementation of GENI that fully supports the GENI AM API [18]. Just like other existing cloud service providers [8], [9], networking resources in ProtoGENI (e.g., programmable routers and PCs) are distributed in multiple aggregates. Moreover, ProtoGENI allows its users to provision the network connections between its managed nodes [2], which makes it the perfect SDN provider for our game SDN network.

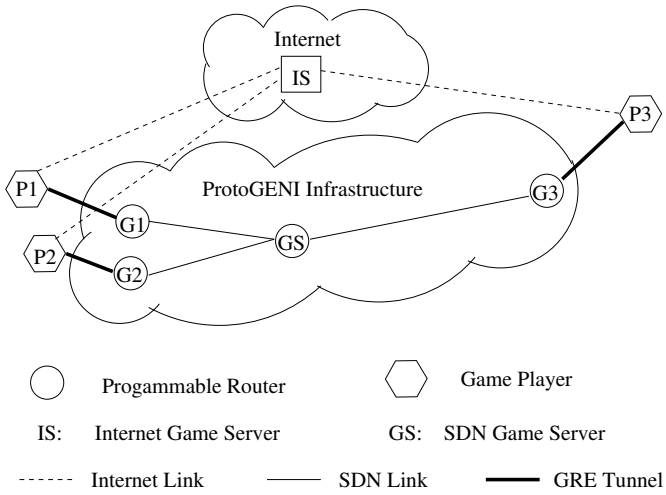

Fig. 2: Three OpenArena players connected either by (1) the Internet, or (2) our custom SDN network over ProtoGENI.

For our particular example, depicted in Figure 2, we wanted to create a custom SDN network for three OpenArena game players: P1, P2 and P3. The first two player, P1 and P2, were located in Kentucky, while P3 was located in Utah.

[2]Users of ProtoGENI can define the bandwidth of connections within an aggregate. For connections between two different aggregates, they use "stitching" technique to provision bandwidth.

Algorithm 1 Multiplayer Gaming HyperNet

```
//get configuration and IP of each player
myHyperNet = readConfig()
regHyperNet(myHyperNet) //register HyperNet
for every player p in myHyperNet do
   gateway = findPR(p)
   gatewayList.add(gateway)
end for
//find the central node to load game server
gameServer = findCentralNode(gatewayList)
for every gateway g in gatewayList do
   addTunnel(g, gameServer) //create a virtual link
end for
//load OpenArena game server software onto central node
loadApp(gameServer, OpenArenaServerApp)
build() //deploy the topology
//configure and start the server
loadApp(gameServer, OpenArenaServerScript)
//join all the players
while true do
   joinRequest = checkJoin() //get join request
   tunnelInfo = createTunnelInfo(joinRequest)
   //setup tunnel between gateway and player
   addGW(tunnelInfo)
end while
```

Algorithm 1 shows the pseudo-code for our OpenArena hypernet game. The hypernet first reads the configuration file for the SDN network, including the name of the SDN network, the password needed to join the network, and the network location information of the (anticipated) participants, in our case, the IP addresses of the players. The hypernet then finds a nearby "gateway" programmable router for each of the players via the "findPR()" API call. Next, it finds a central node that provides the minimum average delay to all the gateways via the "findCentralNode()" API call and creates a virtual channel from each gateway to the central node. The hypernet then loads the OpenArena server application to the central node via the "loadApp()" API call. Finally the hypernet deploys the SDN network using the "build()" API call.

At this point, the game SDN network is up and running. Next, game players use the "join()" API call to join the game. The hypernet gets the join requests from players, figures out the configuration for the corresponding gateways (IP addresses that need to be assigned, and the routing table entries that need to be added, etc). It then creates a GRE tunnel between each player and its assigned gateway. Recall that ProtoGENI also allows its users to provision virtual links between its managed nodes. In our case, the hypernet reserves 100Mbps bandwidth for all "SDN Links" shown in Figure 2, which is more than sufficient for the maximum sending rate of 200Kbps. Note here that the hypernet is fully expandable to allow dynamic game joiners, i.e., by finding and assigning more gateways to new requesting game players.

We implemented our OpenArena hypernet game and compared it with the standard OpenArena game and existing public game servers. Using the network hypervisor API calls, it only took about 120 lines of Java code to write our OpenArena hypernet game. We then ran our OpenArena hypernet game on the network hypervisor to create a custom SDN network specifically designed for our three players. As shown in Figure 2, we were able to find an optimal game server (GS) in ProtoGENI that provided an average round trip time of 22 ms. In comparison, the best public Internet Server ("IS" in the figure) provided an average round trip time of 212 ms, showing that custom placement of the game server can provide an order of magnitude improvement in the game's response time.

VI. Conclusion

In this paper we introduced HyperNet Games, an abstraction that enables one to dynamically deploy a multi-player game network specifically designed for a set of players and the game by leveraging the emerging SDN techniques and cloud computing resources. The proposed HyperNet architecture provides a platform for game developers to easily develop a game HyperNet and for game players to easily deploy a game SDN network. By using our OpenArena game HyperNet, an average game player is able to deploy a OpenArena game SDN network which provides much better gaming experience than the best static game server that can be found from the Internet.

References

[1] "WarCraft." [Online]. Available: http://us.blizzard.com/games/war3/

[2] "Defense of the Ancients." [Online]. Available: http://www.playdota.com/

[3] "World of WarCraft." [Online]. Available: http://us.battle.net/wow/

[4] "Second Life." [Online]. Available: http://secondlife.com/

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, 2008.

[6] X. Chen, Z. Morley, M. Jacobus, and V. Merwe, "ShadowNet: A Platform for Rapid and Safe Network Evolution," in *Proceedings of the 2009 conference on USENIX Annual technical conference*, ser. USENIX'09. Berkeley, CA, USA: USENIX Association, 2009.

[7] L. Peterson, S. Sevinc, J. Lepreau, R. Ricci, J. Wroclawski, T. Faber, S. Schwab, and S. Baker, "Slice-Based Facility Architecture," 2009. [Online]. Available: http://www.cs.princeton.edu/~llp/arch_abridged.pdf

[8] *Programming Amazon EC2*. O'Reilly Media, 2011. [Online]. Available: http://aws.amazon.com/ec2/

[9] D. Chappell, "Introducing Windows Azure," 2009. [Online]. Available: http://www.windowsazure.com/en-us/

[10] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum, "Virtual Appliances for Deploying and Maintaining Software," in *Proceedings of the 17th USENIX conference on System administration*, ser. LISA '03. Berkeley, CA, USA: USENIX Association, 2003.

[11] "Amazon Simple Storage Service." [Online]. Available: http://aws.amazon.com/s3/

[12] G. P. Office, "Global Environment for Network Innovations - System Overview," 2008. [Online]. Available: http://www.cra.org/ccc/files/docs/GENISysOvrvw092908.pdf

[13] VMware, "A Performance Comparison of Hyperivors," 2007. [Online]. Available: http://en.wikipedia.org/wiki/Hypervisor

[14] S. Huang, J. Griffioen, and K. L. Calvert, "Fast-tracking GENI Experiments using HyperNets," *GENI Research and Education Experiment Workshop*, March 2013.

[15] "Windows Azure DataCenter Locations." [Online]. Available: http://goo.gl/maps/dGtgJ

[16] "OpenArena." [Online]. Available: http://openarena.ws/

[17] "ProtoGENI." [Online]. Available: http://www.protogeni.net/

[18] T. Mitchell, "GENI Aggregate Manager API," 2010. [Online]. Available: http://groups.geni.net/geni/wiki/GeniApi