applied sciences

MDPI

Article

Transparent Redirections in Content Delivery Networks

Tomáš Boros *, Rastislav Bencel and Ivan Kotuliak

Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Ilkovičova 2, 842 16 Bratislava, Slovakia; rastislav.bencel@stuba.sk (R.B.); ivan.kotuliak@stuba.sk (I.K.)
* Correspondence: tomas.boros@stuba.sk

Received: 9 October 2019; Accepted: 6 December 2019; Published: 11 December 2019

Featured Application: This paper presents a solution of a TCP session handoff mechanism using Software Defined Networking, which can be used as a transparent redirection mechanism in Content Delivery Networks.

Abstract: In today's age of digital media and social networks, delivering content to a massive scale of recipients became one of the main challenges. Load Balancers and Content Delivery Networks are used to distribute content to multiple locations, which increase the scalability of services and decrease the load on content origins. Both technologies rely on redirections. Redirections have not received a significant amount of attention in the recent years; however, they do impose some limitations. In this article, we propose a transparent redirection mechanism, which exploits the versatility of Software Defined Networking. The redirection method is achieved by handing off existing TCP sessions without any required modifications to the endpoints. This article demonstrates how the proposed redirection mechanism can be adopted in Content Delivery Networks and Load Balancing scenarios. The performance of the solution is thoroughly tested and compared to existing legacy solutions.

Keywords: Software Defined Networking; TCP session handoff; Content Delivery Networks; redirections

1. Introduction

Nowadays people are almost always online on their mobile devices, which enables them to share to moment (content) via social networks just using a couple of taps of fingers. Data shared on social media must be effectively delivered and distributed to the required users in a matter of seconds. Users no longer keep their data on their portable devices, they are backing up photos, videos and other content to cloud providers which are shared between multiple devices and locations. Portable data storage is no longer used as they are not considered convenient anymore.

Content Delivery Servers (CDNs) allow content providers to effectively deliver their content to consumers on a massive scale. CDNs are systems for the efficient delivery of digital objects (e.g., videos, photos) and multimedia streams (e.g., live television streams). Typically, a CDN consists of several (hundreds) servers that deliver the digital objects and a management/control system. The management/ control system takes care of the content distribution, request routing, reporting, metadata and other aspects that make the system work [1]. It is needless to say that is not a new technology. It has been around for over 20 years, but redirection mechanisms have not received a significant amount of attention in recent years. Software Defined Networking (SDN) is a new approach towards networking, which decouples the data plane from the control plane. This approach opens new possibilities to enhance redirections used in CDNs. This article presents a transparent SDN based redirection mechanism, which is an extended version of our paper [2] published in 2019 at the 42nd International Conference on Telecommunications and Signal Processing (TSP). This article additionally introduces a transparent load balancing use case.

2. State of the Art

2.1. Protocols and Caching

In the early 90s 44% of Internet traffic originated from FTP (File Transfer Protocol) requests, which later faded away and HTTP (Hypertext Transfer Protocol) communication took the leading position [3,4]. HTTP is encapsulated into the Transmission Control Protocol (TCP), which is a connection-oriented protocol and provides reliable data transfer between the endpoints as the chunks sent are acknowledged by the receiving side. TCP was first introduced in 1981 [5]. The protocol is responsible for breaking the message into TCP segments and reassembling them at the receiving side, capable of data reordering in case of out of order delivery. TCP assigns a sequence number to each byte transmitted and expects a positive acknowledgement (ACK) from the receiving layer. If the ACK is not received within a timeout interval, data is retransmitted. Sequence numbers ensure that the data stream is passed to the application layer in the right order eliminating duplicate TCP segments. A fundamental notion in the design is that every octet of data sent over a TCP connection has a sequence number. This space ranges from 0 to $2^{32} - 1$. Since this space is finite, all arithmetic dealing with sequence numbers must be performed modulo 2^{32}. This unsigned arithmetic preserves the relationship of sequence numbers as the cycle from $2^{32} - 1$ to 0 again [5].

Over the years of evolution of HTTP (versions 0.9, 1.0, 1.1, 2 and 3 currently under development and standardization), the protocol was extended to allow caching at various locations to eliminate the need to reach out to the origin server or eliminate the need to send full responses to requests. This mechanism saves resources on the network and on the origin server too. The list of cache control mechanisms and various headers are found in RFC 7232 and RFC 7234 [6,7]. With several cache nodes available, the content could be distributed to various geographical locations to lower the time required to deliver the content to the endpoints. Effective content delivery can be achieved by redirecting user requests to the closest cache server based on the user's location. In the following section, the article gives insight into various redirection mechanisms. Due to the popularity of content distribution via HTTP, this article will focus on this application layer protocol in version 1.1 in an unencrypted form.

2.2. CDN Redirections

Redirection can be achieved in multiple ways. Barbir et al. in RFC3568 [8] list and compare the known CDN request routing mechanism:

- DNS (Domain Name System) based request routing
- Application layer request routing
- Transport layer request routing

2.2.1. DNS Based Request Routing

DNS bases request routing is common in CDNs due to the ubiquity of the DNS system [9]. The DNS server is responsible for translating domain names to IP addresses. DNS maintains a large distributed database, which means not all the domain names are stored on every DNS server. To deal with this fact, the DNS server might recursively query the specific domain name by pursuing the query for the client at other servers, starting at the root zone through Top Level Domain servers towards the authoritative name server of the specific domain.

When DNS based request routing is used, a specialized DNS server is inserted in the DNS resolution process acting as a request router. The server is capable of returning a different set of A, AAAA or CNAME records based on the user's location, defined policies or other metrics. The returned record points directly to a CDN cache node also referred to as a surrogate server. The redirection procedure is depicted in Figure 1.

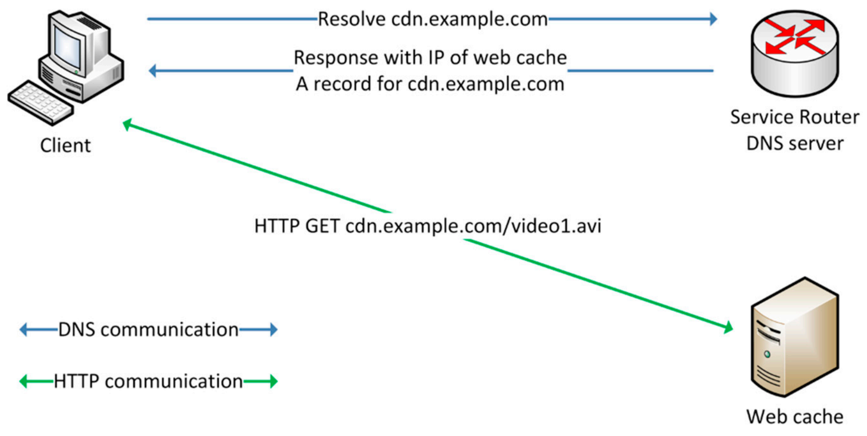

Figure 1. DNS-based redirection.

DNS is simple and does not scale well. RFC 1034 [9] states the following:

- "The sheer size of the database and frequency of updates suggest that it must be maintained in a distributed manner, with local caching to improve performance. Approaches that attempt to collect a consistent copy of the entire database will become more and more expensive and difficult, and hence should be avoided."
- "Tradeoffs between the cost of acquiring data, the speed of updates, and the accuracy of caches, the source of data should control the tradeoff."

Due to the large scale of the distributed database, it is slow to populate updates and changes in cases of changes in the CDN or during downtimes and unexpected situations. This limitation of the DNS is also reflected in the DNS based redirection. Barbir et al. summarize these reflected issues and others in RFC3568 [8]:

- "DNS only allows resolution at the domain level. However, an ideal request resolution system should service request per object level."
- "In DNS based Request-Routing systems servers may be required to return DNS entries with a short time-to-live (TTL) values. This may be needed in order to be able to react quickly in the face of outages. This in return may increase the volume of requests to DNS servers."

DNS based redirection is fast and simple; however, as only the domain name is taken into consideration a non-optimal surrogate server might be returned to a client's request. The adaptation process might take a while in case of downtimes or changes in the CDN which might result in service unavailability decreasing the QoE (Quality of Experience) for users.

2.2.2. Application Layer Request Routing

Application layer request routing redirects the requests based on application layer parameters, e.g., in HTTP, additional headers such as User-Agent or Uniform Resource Locator (URL) might further enhance the accuracy of redirection. For HTTP, the request router might act as an HTTP server which redirects the requests to surrogate servers using 301/302 redirection messages. This procedure is depicted in Figure 2. Such redirection is slower compared to a DNS based redirection but might be more efficient as multiple parameters might be taken into consideration during redirection decision. Another widely used application layer redirection for HTTP is URL rewriting when the rendered website on the origin server replaces the URLs of the distributed content to the domain name or IP address of a specific surrogate server. This approach eliminates the need to process redirect messages for the clients.

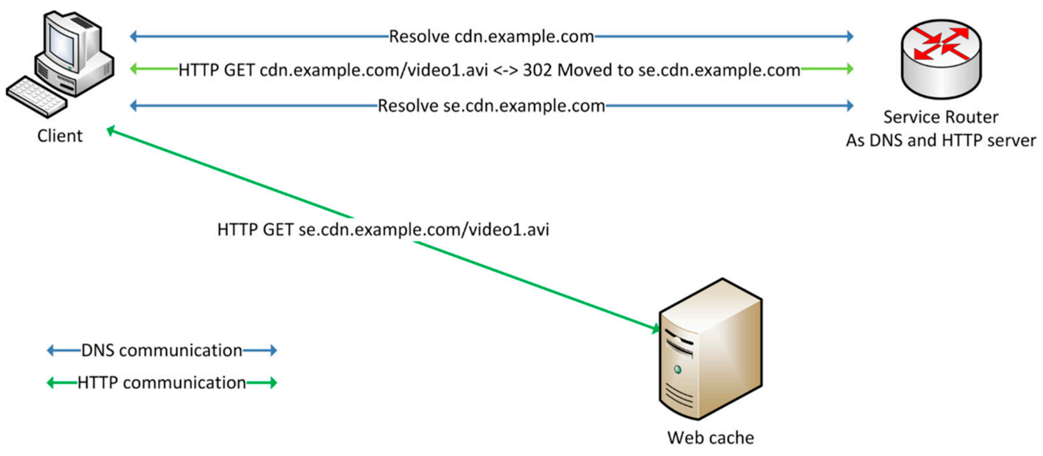
Resolve cdn.example.com
HTTP GET cdn.example.com/video1.avi <-> 302 Moved to se.cdn.example.com
Resolve se.cdn.example.com
Client
Service Router
As DNS and HTTP server
HTTP GET se.cdn.example.com/video1.avi
DNS communication
HTTP communication
Web cache

Figure 2. Application layer redirection.

2.2.3. Transport Layer Request Routing

Transport-layer request routing requires service awareness and support from the network. Transport-layer request routers use information in the transport-layer headers (optionally application layer details too) to determine to which CDN surrogate hand off the session, which proceeds to serve the requested content [8]. Such handoff can be achieved by migrating the TCP session from the request router to the surrogate server or by achieving triangular routing, where the requests go from the client towards the request router, which forwards the requests to the surrogate server. The surrogate server proceeds to serve the content directly, while the IP address and port numbers are re-written by the network (depicted in Figure 3) [10]. This article focuses on these type of redirections. Some existing solutions are presented in the next section.

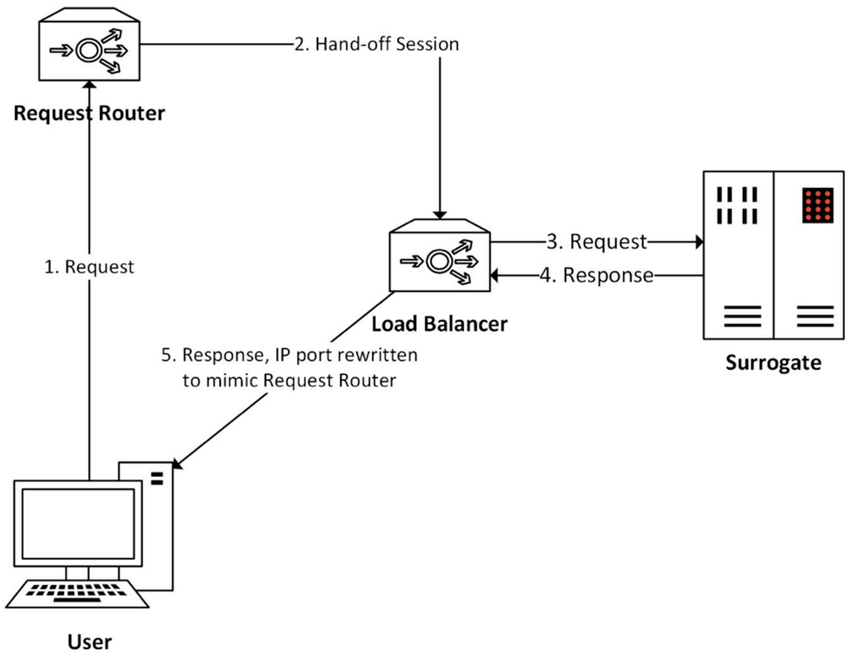

Figure 3. Triangular routing.

2.3. Existing Transparent Redirection Mechanisms

The vast majority of communication over CDN happens using HTTP protocol, which is carried in TCP. To achieve a transparent redirection, TCP session handoff must be performed from the request router (or another node) to the surrogate server. This is challenging, as the TCP is meant to be a connection-oriented point-to-point protocol.

Guerney Hunt, Erich Nahum and John Tracey in IBM T.J. Watson Research Center already came up with an idea of handing off an existing TCP session from a load balancer to a content server in 1997 [11]. Authors in this report introduce the handoff mechanism of TCP sessions from the load balancer, where the initial TCP session from the client is established. After establishment, the clients send the HTTP request, which is examined and inspected by the load balancer, and based on the requested content, the TCP session is handed off to a content server, which has the requested content cached. The initial TCP session parameters between the client and the load balancer are passed to the cache server (windows sizes, various TCP options, sequence numbers) re-using the same TCP session parameters between the load balancer and the cache server. Once the session is handed off, the communication flows directly between the client and the cache server, while the IP address of the cache server is rewritten, to mimic the load balancer. It is not clear from the report, whether a working prototype was implemented or not. Authors also mention a different approach using T/TCP (Transactional TCP) where the initial HTTP request can be sent already in the 3-way handshake phase [12,13]. This protocol is faster than TCP and delivery reliability is comparable to that of TCP. T/TCP suffers from several major security problems as described by Charles Hannum [14,15]. It has not gained widespread popularity and the protocol was moved to historic status in May 2011 by RFC 6247. Authors also mention in the report a commercial product by Resonate Inc., which adopts the same mechanism, which they call 'connection hop'; however, the product does not seem to exist anymore [16].

Li Wang introduces an implementation of TCP handoff based on the FreeBSD kernel, which adopts the same concept of triangular routing depicted in Figure 3 mentioned earlier. In this implementation, the load balancer communicates with the content servers in the same way as in the previous prototype. The implementation is limited to FreeBSD and requires modifications to the content servers to make it work [17].

Arup Acharya and Anees Shaikh in their report [18] exploit the mobility support of IPv6. IPv6 mobility allows binding the TCP session to a different host during the TCP session establishment procedure. In the initial phase, the client sends a TCP SYN packet to the request router, which chooses an optimal surrogate server for the given session. The TCP SYN packet is sent over an IP-in-IP to the surrogate server including an IPv6 binding update information indicating that the request router is the home address for this session. This mobility header tells the IP layer to update the source IP address with the home address field of the binding update (IP address of request router) in the SYN, ACK response, which is sent directly to the client. The client's TCP layer sees a packet with the source IP of the request router, thus allowing it to complete the connection [19]. This approach works only in the TCP session establishment phase, which means, the surrogate server cannot be chosen based on application layer information; however, it eliminates the need to use DNS based redirection. Furthermore, such a feature is allowed only in IPv6, while the vast majority of clients still use IPv4 [20].

Matthias Wichtlhuber, Robert Reinecke and David Hausheer in their paper [21] did extensive measurements on live TCP socket migrations. In their experiments, they measured how the congestion window (CWND) and the slow start threshold (SSThresh) is affected, when a live TCP socket is migrated from one CDN surrogate to another one in various network conditions for long-lasting TCP sessions, e.g., VoD streaming. They applied delays and packet loss to simulate real-life scenarios. The TCP socket migration was achieved using the TCP_REPAIR flag [22], which puts the socket to maintenance mode, so the TCP parameters can be dumped and transferred to another server, where the session can be restored, thus allowing migration of negotiated parameters during the TCP handshake phase and current sequence and acknowledgement numbers. The paper does not provide measurements of the delay required to perform the TCP migration itself.

Yutaro Hayakawa, Michio Honda, Lars Eggert and Douglas Santry in their paper [23] present Prism proxy for fast load balancing technique, where the TCP sessions established with the load balancer, upon reception of the request, are handed-off to a chosen backend server from the pool. The solution is designed for data center scale load balancing; however, it could be used for CDNs as a transparent redirection engine over greater areas. The Prism proxy uses the same TCP_REPAIR [22] flag to pass the sessions between the servers. The solution uses HTTP/1.1 and is capable of reusing existing TCP sessions for subsequent requests. The authors used mSwitch, which is a kernel software switch that can forward packets at a rate of over 10 Mpps on a single CPU core [24].

To perform a transparent redirection the CDN must be able to perform a TCP session handoff; otherwise, the redirection becomes visible to the end-user. All the related work mentioned above points out the challenges associated with the handoff procedure.

2.4. TCP Session Handoff

To be able to perform a TCP session handoff to redirect an existing TCP session from the request router to a surrogate server, we must be able to predict what initial sequence number the surrogate server will choose during the TCP 3-way handshake phase, which is impossible without modifications to the TCP/IP stack on the surrogates. Another approach which was mentioned earlier is the TCP_REPAIR flag, which requires modifications on the endpoints, thus disallows the usage of commercial closed source solutions. To mimic the request router after a handoff, the IP addresses and the port numbers are re-written over the network. To keep the TCP session alive, such modifications must be applied to the sequence and acknowledgement numbers in the TCP header too. This approach could be achieved by creating a network element which can modify these values in the TCP segment. TCP session handoff requires further support from the network which could be possible thanks to the versatile possibilities of SDN (Software Defined Networking). Transparent redirection in CDN done by handing off the TCP sessions from a point of view of the visibility increases the security too, as it hides the topology behind a single IP address and does not exploit the addresses of surrogate servers to the public network.

3. SDN Based Transparent Redirection

The state-of-the-art chapter in this article introduced existing transparent redirection methods; however, each redirection method requires certain modifications on the endpoints, which prevents ISPs and content providers using legacy closed source CDN surrogate servers. The next section describes our proposed solution which allows transparent redirection of requests. The solution does not require any additional changes to the TCP/IP stack or the operating system kernel, which can be achieved using SDN.

Software Defined Networking (SDN) is a new approach toward computer networking. SDN's main idea is to separate the control plane from the data plane to create a centralized controller. This centralized controller controls multiple data plane network devices, called forwarders in SDN terminology. Forwarders are fast packet forwarding devices that are not capable of making any forwarding decisions until they are configured by the controller. The idea of this approach to eliminate the problems with compatibility of different vendors. It supports BYOD (Bring Your Own Device) and vendor independence. In today's agile and elastic networks devices must respond quickly and adapt to the changes. Implementing new services requires an enormous amount of effort with legacy devices and often results in a need to replace the existing infrastructure to newer devices [25]. SDN improves network automation by using a common API to abstract the underlying network details from the orchestration and provisioning systems and applications. The centralized environment brings a better network reliability and security, uniforms the policy enforcement and decreases the number of configuration errors [26].

3.1. System Architecture

As introduced in the state of the art, to be able to achieve TCP session handoff the underlying network must be able to provide an action that is capable of incrementing the sequence and acknowledge numbers on the fly over the network. For this purpose, it is crucial to use a software-controlled network. In the domain of SDN, the data plane consists of a set of Forwarding Elements (further referred to as forwarders in this document). The choice for the SDN architecture is based on OpenFlow 1.3 [27], as the protocol since this version supports experimenter actions, which allows us custom action implementation.

In our architecture we distinguish two types of forwarders:

- Access Forwarders: conventional OpenFlow 1.3 enabled forwarder aligned with the specification;
- Core Forwarder: conventional OpenFlow 1.3 enabled forwarder with support for experimenter actions that can modify the TCP sequence and acknowledge numbers.

This design was chosen, to eliminate the need to put modified forwarders to the access layer. Request router and the surrogate servers of the CDN are connected directly to the core forwarders. The network topology must be connected to an extended SDN controller with the additional experimenter actions defined.

This approach was chosen to eliminate the need to use modified SDN forwarders in the networks. Modified forwarders are only required where a surrogate server or the request router is attached. Other forwarders in the network can be any OpenFlow 1.3 compliant legacy forwarders.

To make the CDN work, a request router must be placed, which is capable of establishing the initial TCP connections with the clients and which further establishes TCP connections to surrogate servers. A DNS server is placed in the network too, which handles domain translation for the domain name *dizp.bt*. The content is distributed over the CDN via the *cdn.dizp.bt* domain. Surrogate servers are addressed by prepending se(ID), e.g., se1.cdn.dizp.bt. The domain name for the content origin is *www.dizp.bt*, which is a static web server hosting various files of various sizes. To be able to make redirection decisions the request router must be able to communicate with the SDN controller. The SDN controller provides a WebSocket based communication channel over the Northbound API [26]. Figure 4 depicts the proposed high-level architecture. The presented network represents a single L2 domain, where each component of the network resides on the same broadcast domain.

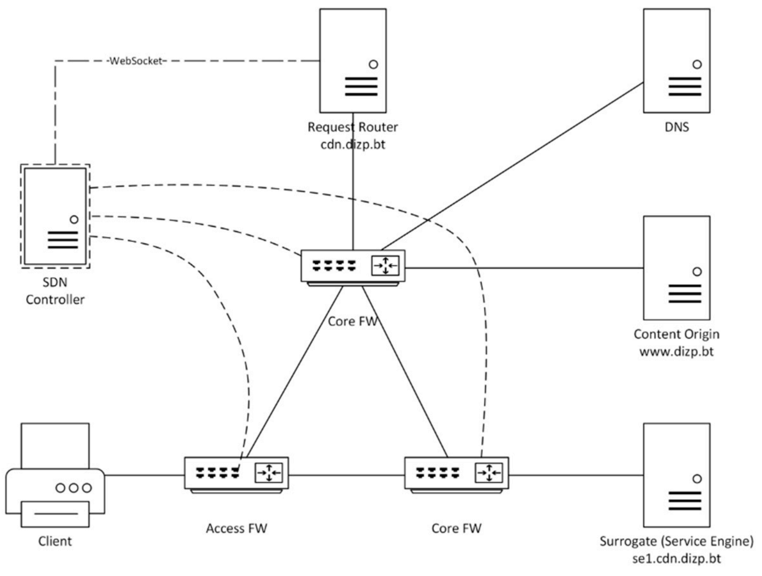

Figure 4. High-level architecture overview.

3.2. SDN Based TCP Session Handoff

A TCP session is defined and uniquely identified by a 4 tuple, which is the source IP address, destination IP address, source port, and destination port. These values cannot be changed once a TCP session is established. The main idea of the proposed SDN-based transparent redirection is not to handoff the client's TCP session from the request router to the surrogate server but to synchronize two separate TCP sessions with each other (depicted in Figure 5). The two separate TCP sessions must be established before the synchronization—one from the client towards the request router and the other from the request router towards the surrogate server (steps 1 and 2). Once the TCP sessions are established, an HTTP request is expected from the client towards the request router (step 3). This request is modified and sent over the other TCP session from the request router to the surrogate server (step 4). At this point, the surrogate server starts sending a byte stream as a response to the request. This byte stream is however directed from the surrogate server directly to the client bypassing the request router, while the IP addresses and port numbers are rewritten to match the 4-tuple value of the initial TCP session (step 5).

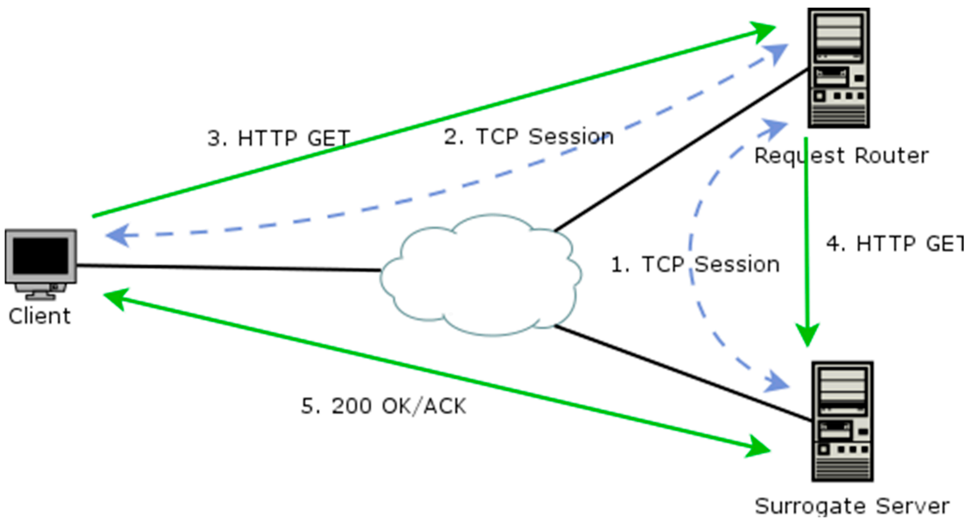

Figure 5. Transparent redirection.

To make this solution work we must synchronize the sequence and acknowledgement numbers in the TCP headers too. Such modification could be implemented on a data plane as an experimenter action. The action must be able to increment these numbers to map the other TCP session's values. Once the sequence/acknowledge number reaches the maximum value of a 32bit number, it overflows and starts from 0. Upon successful installation of actions to the data plane, the client consumes the response message from the surrogate server, which was originally generated for the request router (1st TCP session), while the client "thinks" the response is coming directly from the request router (2nd TCP session). In the end, to finish the handover procedure, the SDN controller must create reset (RST) or FIN messages to mitigate both sessions which were established with the request router.

To be able to perform the handoff, we have to keep track of the existing TCP sessions in the network. This can be achieved by intercepting the initial TCP handshakes with the request router and the surrogate servers. The request router prepares a pool of established TCP sessions towards the surrogate servers, which are reused for the handoff.

Once the client establishes a TCP session with the request router a valid HTTP request message is expected. From this point, the handoff procedure begins. The whole handoff procedure is depicted in divided into 4 parts. The solution is designed to work with the standardized HTTP 1.1 in unencrypted form [28]. In the 1st part, the clients send an HTTP GET request including the URL and additional request headers. This request is sent to the controller (CNT) and then forwarded to the request router.

Once the full request is sent to the request router the controller blocks the communication from the request router towards the client (Flow Mod:1). The request router parses the request and reports the request size to the controller over the management WebSocket channel in a *setrequestsize* message. This confirms the controller, that a full request was received (request might be fragmented into multiple packets). Once confirmed, the controller chooses a surrogate server, where the TCP session will be handed off (synchronized with).

The request router queries this information in a *getmatchingsess* message, which returns the TCP session parameters. Based on the returned information the request router modifies the request by updating the *Host* header in the HTTP request. The size of the new request is reported again to the controller via the *setrequestsize* message.

In part 2 of Figure 6, the request router sends out the request towards the surrogate server via an already pre-established TCP session (this session was chosen by the controller). The message is sent out and sent to the controller in a Packet IN message. At this point, the controller has all the required information to perform the handoff. The controller installs flow mods to the core forwarder, where the surrogate is connected, and to the access forwarder, where the client is connected. The following flow mods are installed:

- Flow mod 2: rewrites destination IP, port and MAC address for the communication from the surrogate server towards the client, where the IP, port and MAC address are rewritten to mimic the request router. Sequence number and Acknowledge number is incremented to synchronize with the TCP session established between the client and request router.
- Flow mod 3: rewrites the source IP, port and MAC address to match the parameters of the client. The packet sent out from the surrogate server is originally destined to the request router.
- Flow mod 4: rewrites source IP, port and MAC address to match the parameters of the request router for the communication from the client towards the surrogate server. Sequence and acknowledgement numbers are incremented to synchronize with the TCP session between the request router and the surrogate server.
- Flow mod 5: rewrites the destination IP, port and MAC address to match the parameters of the surrogate server as the outgoing packet from the client is destined to the request router.
- Flow mod 6 and 7 blocks any further communication from the request router towards the client or surrogate server over the two established TCP sessions.

If there are several other forwarders between the core and the access switch, the intermediary routers forward the packets based on the destination IP address. The destination IP address is correctly updated on the core and access forwarders for the right routing direction. At this point, all the flow mods are prepared for the synchronization and the corresponding communication continues between the client and the surrogate server exclusively without sending the packets to the SDN controller.

In part 3 of Figure 6, the modified request is sent out from the controller to the surrogate server, which parses the request and sends a response upon a valid request. The response and acknowledgement packets are forwarded through the network and the parameters are rewritten on the core and access forwarders based on the flow mods from part 2.

To correctly finish the handoff procedure, it is crucial to mitigate the initial TCP sessions established with the request router. To mitigate the sessions the controller generates TCP packets with RST flags set for both TCP sessions, which are sent out via PACKET OUT messages in part 4 of Figure 6. Once the session is mitigated, the request router creates a new TCP session towards the surrogate server to maintain the pool of available TCP sessions.

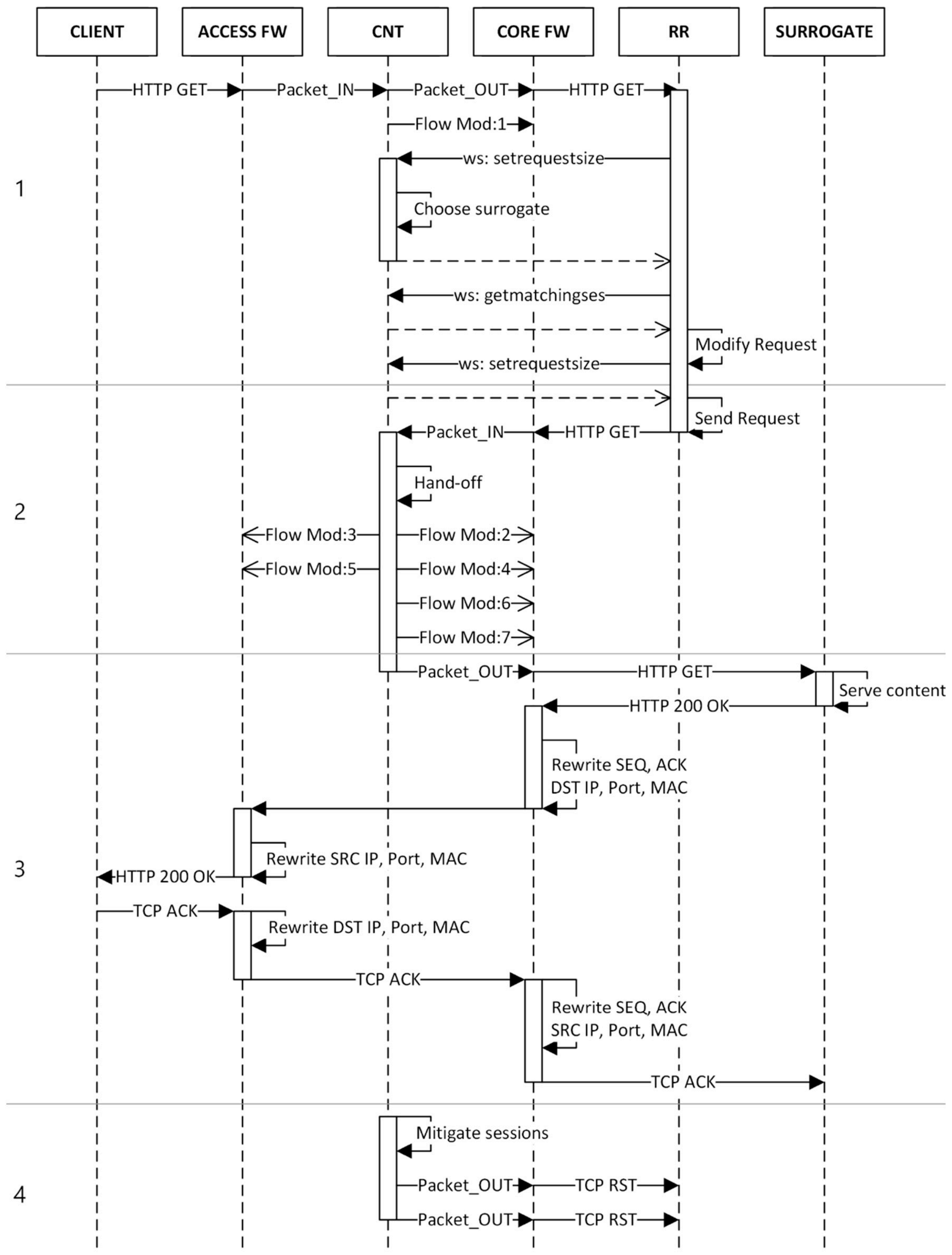

Figure 6. TCP Session handoff procedure.

To calculate the synchronization values of the sequence and acknowledgement numbers the following formulas are used:

$$\text{Sinc\_cs} = ((2^{32}) + (\text{Srs} - \text{Scr}) + (\text{Rrs} - \text{Rcr})) \bmod (2^{32}) \tag{1}$$

$$\text{Ainc\_sc} = ((2^{32}) - \text{Sinc\_cs}) \bmod (2^{32}) \tag{2}$$

$$\text{Sinc\_sc} = ((2^{32}) + (\text{Src} - \text{Ssr})) \bmod (2^{32}) \tag{3}$$

$$\text{Ainc\_cs} = ((2^{32}) - \text{Sinc\_sc}) \bmod (2^{32}) \tag{4}$$

where:

- Srs: Initial sequence number from request router to the surrogate server;
- Scr: Initial sequence number from client to the request router;
- Rrs: Request size from the request router to the surrogate server;
- Rcr: Request size from client to the request router;
- Src: Initial sequence number from the request router to the client;
- Ssr: Initial sequence number from the surrogate server to the request router.

As the sequence and acknowledge numbers are 4-byte numbers. Modulo 32 is applied to prevent the overflow of these numbers. SeqCS (1) is applied in direction from the client to the surrogate server as inc_seq(SeqCS) on the core forwarder (Flow mod 4). AckSC (2) is applied in direction from the surrogate server towards the client as inc_ack(AckSC) on the core forwarder (Flow mod 2). SeqSC (3) is applied in a direction from surrogate to the client as inc_seq(SeqSC) (Flow mod 2) and AckCS (4) is applied in a direction from the client to the surrogate as inc_ack(AckCS) on the forwarder where the surrogate server is attached (Flow mod 4). Action inc_seq and inc_ack are interpreted as datapath actions on the forwarders, which increment sequence and acknowledge numbers, respectively.

Secure TLS based HTTPs connections could not be handed off with this design, as it is not possible to share the negotiated parameters of the TLS context between the endpoints without modifications to them. Successfully handing off such a TCP session via the network only would require a serious vulnerability in the TLS standard, which would deprecate this protocol. Securing the content itself is possible with other types of methods, e.g., DRM (Digital Rights Management). Alternatively, we could deploy HTTPs proxies close to the access forwarders and terminate TLS based connections there and proxy the requests towards the CDN in an unencrypted form. However, this is inefficient and limits the scalability of the network.

4. Verification

This section will provide information about the implementation details starting with the data plane extension, the SDN controller and the CDN request router which are the key components to make the system work. Additional services such as DNS, origin server and the surrogate servers will be described with the deployed topology and network design.

4.1. Forwarder Implementation

Open vSwitch was chosen as a forwarding element due to its popularity and performance [29]. The forwarder consists of two components. The userspace implementation of the forwarding datapath called ovs-switchd and the datapath kernel module written specifically for the host operating system to achieve high performance [29]. The performance of the Open vSwitch kernel datapath reaches up to 800 Mbit/s [30] on a single UDP flow tested, which can be further enhanced by using a different datapath, e.g., DPDK (Data Plane Development Kit) [31] or a hardware implementation. We modified the Open vSwitch open source software implemented forwarder, which currently handles OpenFlow up to version 1.5. As OpenFlow since version 1.3 provides experimenter headers, this version of the protocol became our choice. We added a new action to the standard list of actions with IDs 32 and 33 (inc_seq and inc_ack) to increment the sequence number and the acknowledge number respectively. Both actions take a single 4 Byte parameter, which represents the value, by which the original value in the header will be incremented. The action was both implemented in user-space and kernel-space to achieve high forwarding performance.

The full patch is available at the URL (https://github.com/fr6nco/ovs/compare/v2.8.1...fr6nco:seq_ack_experimenter).

The applied patch extends the OpenFlow interface with new actions:

- inc_seq(increment): increments sequence number by the supplied increment argument. The argument is a 4 Byte unsigned integer

- inc_ack(increment): increments the acknowledge number by the supplied increment argument. The argument is a 4 Byte unsigned integer

We created a testbed, where we attached hosts A and B and manually added flow entries, where we incremented the sequence number of the TCP header in the direction from A to B, and we decremented the acknowledge number in the opposite direction. Decrement is not implemented, however, if we increment the number by $2^{32} - 1$, we will achieve a decrement by 1 as the acknowledge number overflows. We achieved a goodput of around 600 Mbit/s on a single-core CPU with a 2.7 GHz frequency in a virtualized environment using iperf [32], which is similar to the forwarding performance achieved in [30]. The measured goodput is shown in Figure 7. The data plane is emulated using the Mininet emulator [33].

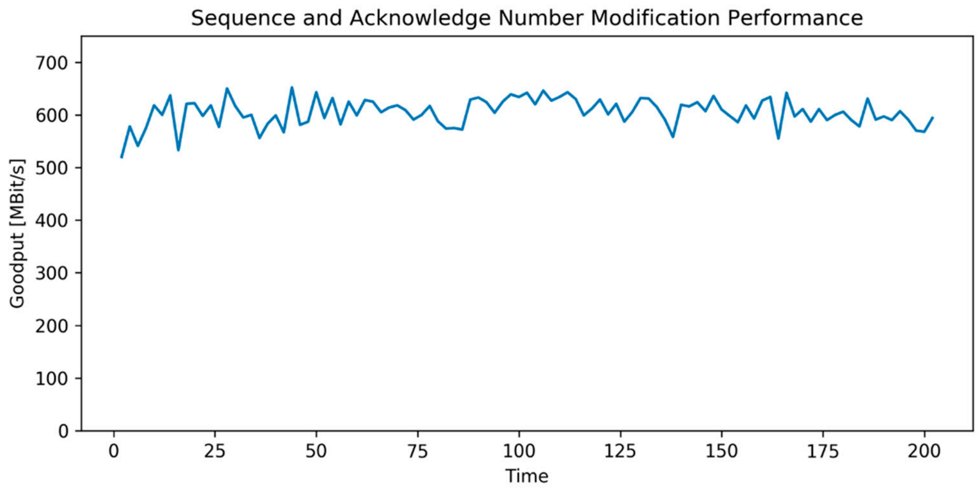

Figure 7. Sequence and acknowledge number modification performance.

4.2. *SND Controller Implementation*

RYU [34] controller was chosen as the SDN controller. The SDN controller is responsible for topology discovery, handover procedure management, and TCP session connection tracking. It provides an API interface for control purposes. The framework is implemented in Python, and it was extended by the new actions defined on the Open vSwitch forwarder to be able to execute these actions on the data plane.

The patch to the RYU framework is available at the URL (https://github.com/fr6nco/ryu/compare/master...fr6nco:dizp):

The implemented controller is using the RYU framework and consists of 4 different modules; these are the following:

- ForwardingModule: responsible for simple L2 forwarding, and provides shortest path lookup for other modules
- CDNModule: responsible for controlling the CDN, implements connection tracking of TCP sessions and controls the handoff procedure
- DatabaseModule: provides an in memory single source of truth data storage for the SDN controller
- WsEndpointModule: provides a Websocket API using JSON-RPC over the Northbound API

As the TCP session establishment must traverse through the SDN controller some additional initial delay is imposed. By capturing the TCP handshake, we can get the imposed initial RTT (round trip time). After a few tests, we found out it is around ~13 milliseconds (approximately ~6 milliseconds one-way delay). Session establishment in the network without having to reach out for the SDN controller brings down the initial RTT to ~0.9 milliseconds (~0.45 milliseconds one-way delay). This imposed delay is acceptable and will not affect the end user's experience. We did a TCP Iperf test between a client and a server, while we pushed the whole communication through the SDN control plane, not just the TCP handshake, and we were able to achieve a goodput of around ~15 Mbits/s. This goodput was a one way TCP transfer test, while the sent and received goodput was reported by both the client and server respectively (Figure 8).

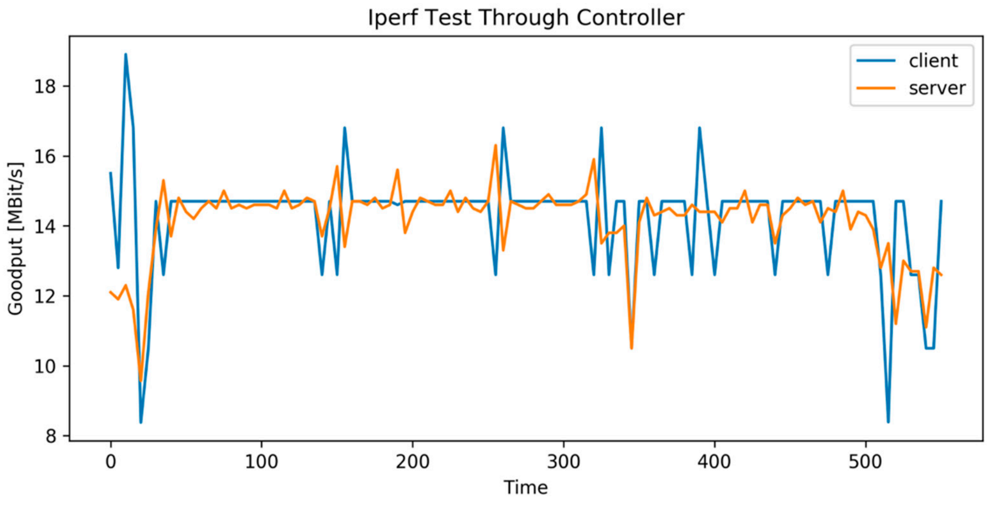

Figure 8. Packet IN, Packet OUT performance of the SDN controller.

The source code of the SDN controller is available on github (https://github.com/fr6nco/cdNgine.git).

4.3. *Request Router Implementation*

The request router is responsible for handling the redirections. Although we use a transparent redirection mechanism which is mostly handled by the SDN controller, we implemented a request router to be able to fully handle TCP connections and HTTP messages. The request router is implemented in NodeJS due to its robustness and performance [35].

The request router consists of 4 main modules:

- Http endpoint module: this module opens a TCP socket, which listens for incoming HTTP requests. The endpoint is running on port 8082. If a request is received, it is parsed and then prepared for handoff.
- Controller endpoint connector module: this module is responsible for communicating with the SDN controller's Northbound API over the WebSocket based JSON-RPC endpoint.
- Request router module: this module is responsible for handling the CDN logic. The module stores information about the assets and maintains the pool of established TCP connections toward the surrogate servers. On incoming requests, this module is responsible for modifying the request and sending it out to the surrogate server. The surrogate server selection is done by the SDN controller, which is queried by the controller endpoint connector module.
- API endpoint: this module provides a REST API for the GUI. The REST API listens on port 3000.

The request router implementation is available on github (https://github.com/fr6nco/reqNRouter.git).

4.4. *Topology and Additional Services*

The three main components of the CDN were described. To evaluate the system, we deployed the solution on a virtualized platform VMWare vSphere ESXi powered by a CPU of an Intel Xeon e5-2450L. The deployed network is depicted in Figure 9. The network consists of 2 types of networks. The management network and list of private networks, which connect the VMs.

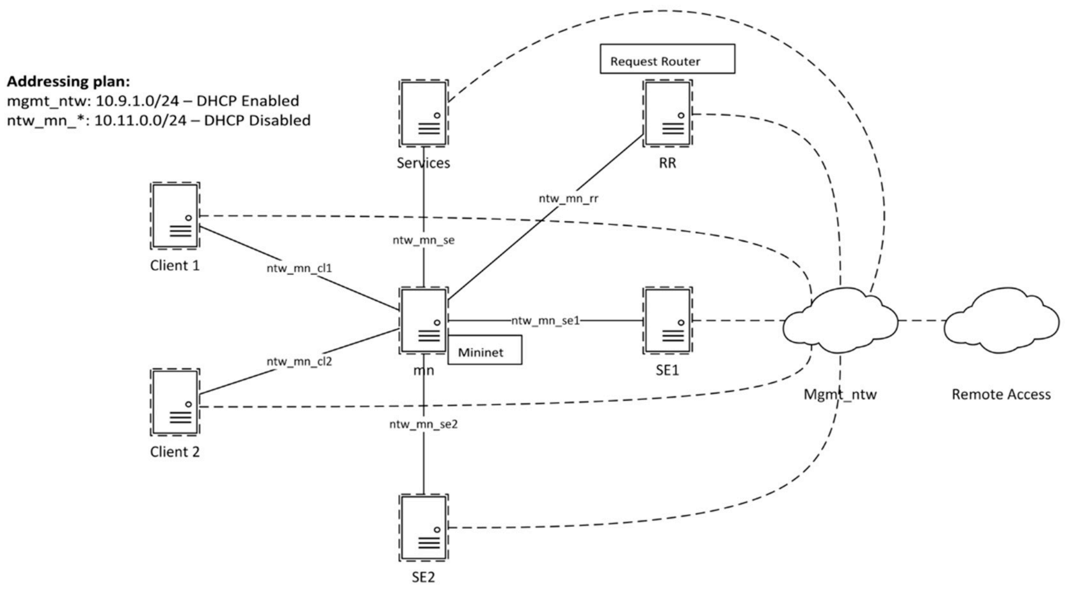

Figure 9. Physical network topology.

In the center of the topology a mininet (mn) node is deployed, where we can simulate various network topologies. The Services node runs the SDN controller, a DNS server, a Graphical User Interface and serves static content as content origin, while the content is distributed and cached by the surrogate servers. The RR node runs the request router in the network. We attached two machines named SE1 and SE2, which are the surrogate servers. We added two clients for evaluating purposes named Client 1 and Client 2. The servers are connected via the mininet network. Servers are also connected via the management interface to allow management access and which allows the services to communicate with each other over the various APIs. The addressing plan for the network is shown in Table 1.

Table 1. IP addressing plan.

Server	MGMT Network	Internal L2 NTW
mn	10.9.1.10/24	-
services	10.9.1.11/24	10.11.0.99/24
rr	10.9.1.12/24	10.11.0.100/24
se1	10.9.1.13/24	10.11.0.11/24
se2	10.9.1.14/24	10.11.0.12/24
client1	10.9.1.15/24	10.11.0.101/24
client2	10.9.1.16/24	10.11.0.102/24

Once the topology is created the various services can be easily installed using automated scripts via ansible. The repository with the ansible playbooks is on github (https://github.com/fr6nco/ansible-dizp).

4.4.1. DNS Server

The DNS server is responsible for translating domain names to IP addresses. We have chosen bind9 as a DNS server. The DNS server is an authoritative DNS server for the domain dizp.bt. It translates domain names to IP addresses in the 10.11.1.0/24 private network. The available domain names are listed in Table 2.

Table 2. Configured domain names.

Domain Name	Record	IP/Domain	Node
dns.dizp.bt	A	10.11.0.99	DNS server
rr.dizp.bt	A	10.11.0.100	Request Router
se1.cdn.dizp.bt	A	10.11.0.11	Surrogate 1
se2.cdn.dizp.bt	A	10.11.0.12	Surrogate 2
origin.dizp.bt	A	10.11.0.99	Origin
www.dizp.bt	CNAME	origin.dizp.bt	Origin
cdn.dizp.bt	CNAME	rr.dizp.bt	Request Router

4.4.2. Content Origin

The content origin is a static website for hosting content for the clients and the CDN surrogate servers. The content is served using a popular web server Apache2 [36]. The web server serves the content of various sizes over the internal private network. It severs a static HTML file, which contains a gallery of images. The server accepts requests destined to host cdn.dizp.bt and www.dizp.bt. The assets (images) are pointed to domain cdn.dizp.bt:8082, so they are retrieved via the CDN. The host cdn.dizp.bt:8082 points to the request router in the network. Expires module in apache2 was enabled, which sets the cache-control header for these images and validates them for 30 days (*Cache-Control: max-age* = 2592000 header is appended to responses).

4.4.3. CDN Surrogate Servers

The surrogate servers are responsible for caching the content from the content origin and serve it to the clients when requested. We decided to use Nginx web server in a proxy cache setup [37]. The server is configured as a reverse proxy pointing to the origin server namely origin.dizp.bt. The surrogate server serves content for host {name}.cdn.dizp.bt, while the name is the hostname of the surrogate server. There are 2 surrogate servers installed with hostnames se1 and se2.

5. Evaluation

The evaluation of the performance was done on two setups. The proposed transparent redirection was compared to the application layer redirection using 302-redirection. In both cases, the request router redirected the requests to the closest surrogate server based on the client's location.

The solution was evaluated in two different scenarios. The first scenario focuses on the overall delay and performance of the proposed solution. The second scenario presents the solution as a load balancer and compares the performance to an application-layer proxy.

5.1. TCP Option Prerequisites

The transparent redirection was not working in the beginning. Later we found out that some of the TCP options caused the packets to be dropped by the Linux kernel.

Some of these headers had to be stripped off when they were sent through the SDN controller during the initial 3-way handshake:

- TCP Timestamps: Timestamp Value (TSval) timestamps are carried in both data and <ACK> segments and are echoed in Timestamp Echo Reply (TSecr) fields carried in returning <ACK> or data segments, originally used primarily for timestamping individual segments, as the properties of the timestamps option allow for taking time measurements. The Timestamps option is important when large receive windows are used to allow the use of PAWS (Protection Against Wrapped Sequence numbers) mechanism. This option further enhances security by eliminating TCP session hijacking [38]. In the moment of the handoff, the TSecr field does not match the TSval of its peer, as this packet was generated by a different node. Stripping off this TCP Option made the handoff work. This option could be synchronized via the control plane too, as during the handoff procedure the packets are traversing through the SDN controller.

- TCP Selective Acknowledgement Option (SACK): TCP may experience poor performance when multiple segments are lost from one window of data. With the limited information available from cumulative acknowledgments, a TCP sender can only learn about a single lost packet per round trip time. An aggressive sender could choose to retransmit packets early, but such retransmitted segments may have already been successfully received. A Selective Acknowledgment (SACK) mechanism, combined with a selective repeat retransmission policy, can help to overcome these limitations. The receiving TCP sends back SACK packets to the sender informing the sender of data that has been received. The sender can then retransmit only the missing data segments [39]. After the handoff, endpoints would see ranges in incorrect values. This could be solved by implementing the incrementing feature of this TCP option too. However, such a feature is challenging, as the order and the position of the TCP options are not fixed in the TCP header. Currently TCP SACK Option is stripped off during the 3-way handshake by the SDN controller.
- Window Scale Option: The three-byte Window Scale option MAY be sent in an SYN segment by a TCP. It indicates that the TCP is prepared to send and receive window scaling and communicate the exponent of a scale factor. The value of the Window Scale option cannot be changed on an established connection [38]. Window scaling option might be problematic when synchronizing different TCP sessions. When a handover is done via the TCP_REPAIR, initial TCP options are synchronized too, which will not happen when two sessions are synchronized with each other. We run several tests and found out when the actual WSCALE is different from the negotiated value the connection control might have issues correctly setting the window sizes, which can end up in a stalled connection. This might happen if the WSCALE TCP option is different on the client and the request router. This issue could be fixed by pre-establishing a pool of TCP sessions towards the surrogate servers using various WSCALE options and choosing the TCP session when doing the handoff, which matches the WSCALE option of the client. This will ensure, that the negotiated WSCALE value is going to be the same even after the handoff.

Keeping these TCP Options is possible; however, further control mechanism is required to achieve precise synchronization of these options. Managing these TCP options is out of the scope of this work.

5.2. Redirection Delay

The redirection procedure on client requests imposes some delay. It is crucial to keep this delay as low as possible. In the next set of measurements, we evaluated the imposed delay for both transparent and 302 application layer redirection. In the first phase, we examined the delay in various networks. We prepared a linear topology over the network, where we attached several forwarders in line, and we attached a client to one and a surrogate server to the other side of the topology. We attached the content origin and the DNS server halfway between the client and the surrogate server. We have to mention, that in this case, the origin server is closer to the client; however, we wanted to measure the delay over a various number of hops when dealing with the handoff procedure. The topology is shown in Figure 10.

On each request 7 flow mods are installed, and 2 extra Packet OUTs are made to mitigate the sessions on the core and the access forwarder together. On each forwarder over path, there are 2 flow mods. S1 (access forwarder) gets 2 flow mods, S(n) (core forwarder) gets 5 flow mods and each forwarder on the path get 2 flow mods installed.

This sums up to:

$$\mathrm{FM\_SUM} = 7 + 2(n - 2) \tag{5}$$

This Formula (5) applies only for the transparent redirection as the 302 redirection method does not go through the SDN controller, thus no path calculation and installation are needed. Flow mods are only installed if the client has not communicated with the surrogate before. This represents 2n flow mods.

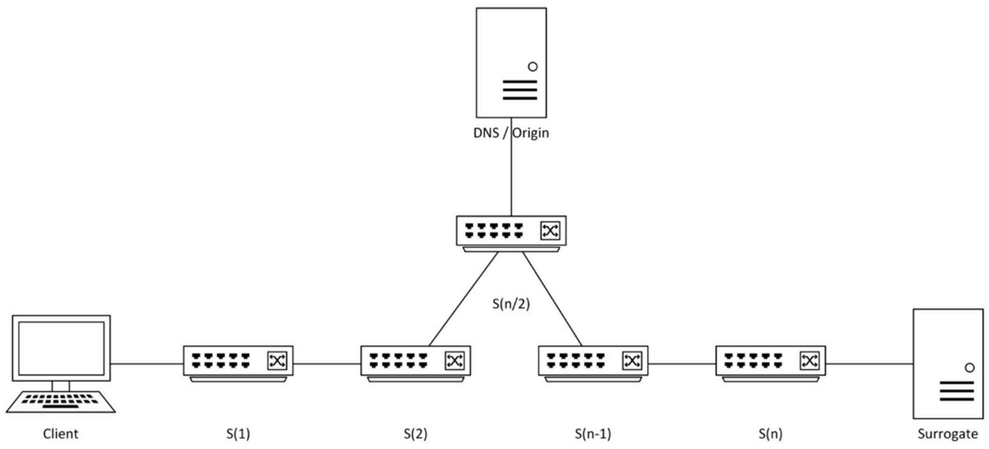

Figure 10. Linear testing topology.

We made several tests on various sizes of the linear network (3, 5, 10, 30, 50, 100), and we measured the time required for the first bytes of data to arrive from the surrogate server to the client.

Figures 11 and 12 show how the transfer delay is affected by the number of forwarders in the path.

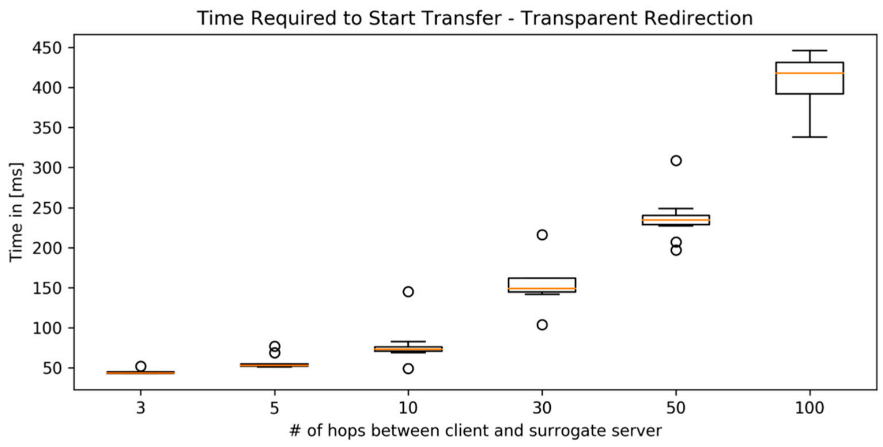

Figure 11. Start transfer delay—transparent redirection.

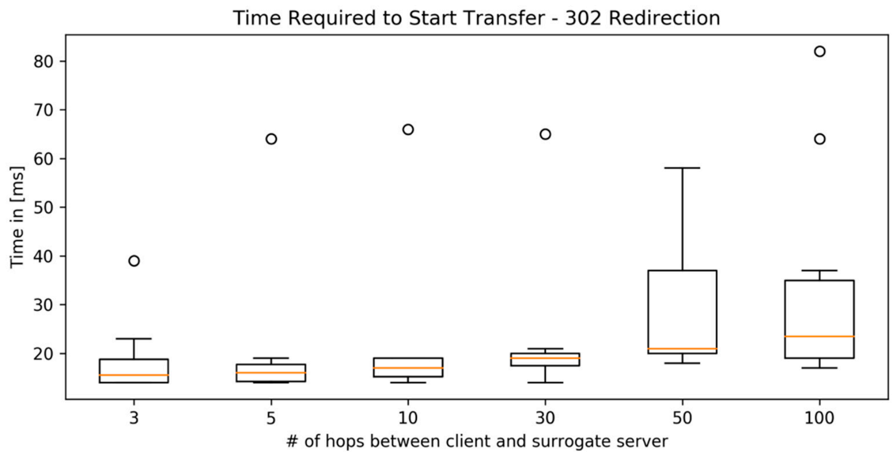

Figure 12. Start transfer delay—302 redirection.

The difference in the start transfer delay is significant. The 302 redirection is relatively constant; only the network itself imposes some delay. The transparent redirection's start transfer delay is increasing linearly with the number of switches. Based on the graphs and the formula above we can see, that the delay is imposed by flow mods that have to be installed to the network. If we put a trendline over the graph, we can see the linearly increasing complexity, which linearly increases the start transfer delay too.

Such delay using transparent redirection is unacceptable. We further modified the SDN controller, where we enabled caching of the shortest paths from the surrogate servers to the clients. Once we installed a path between the client and the surrogate, we do not install it again. In such case, the delay affects only the first request if the client did not make any communication with the surrogate before. This solution significantly decreases the delay. With this approach we have to listen to topology changes, which are handled by the framework itself (Ryu) and trigger installed path recalculations.

The delay of the 302 redirection remains unchanged as the redirection does not require any flow modifications. This redirection only communicates with the SDN controller's Northbound API when the closest surrogate server is chosen based on the client's location.

Figures 13 and 14 show how the delay is affected when path caching is enabled.

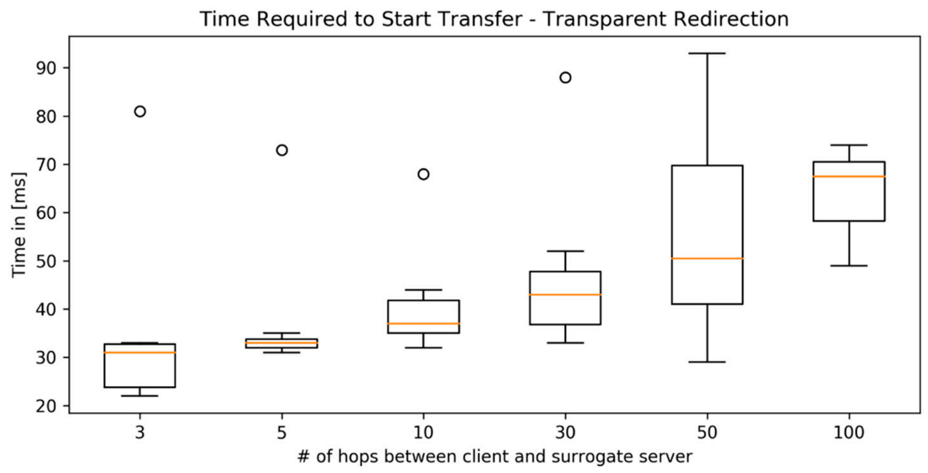

Figure 13. Cached start transfer delay—transparent redirection.

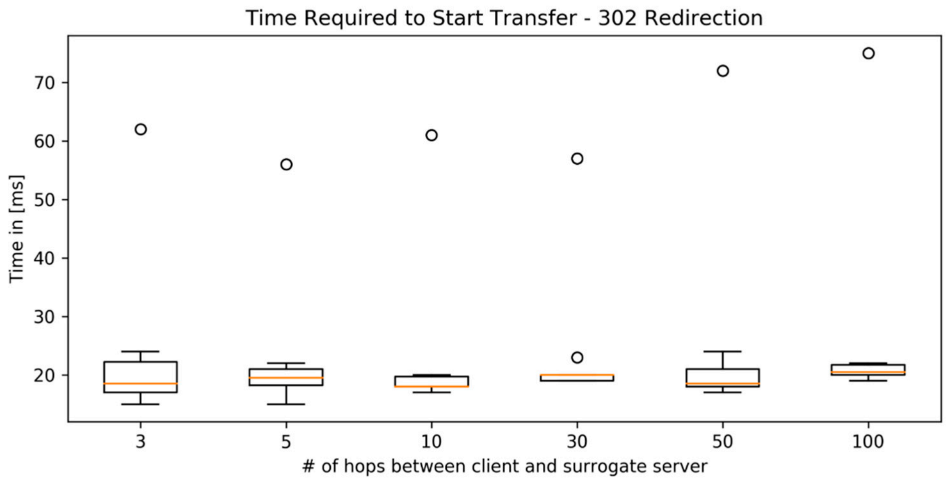

Figure 14. Cached start transfer delay—302 redirection.

The delay is still increasing by the number of hops over the network, but the complexity of the transparent redirection turns constant once the first request from the clients is received. The delay of the first request is visible as a flying dot in the upper outlier for both transparent and 302 redirections. The delay is caused by the network and by the server, which has to fulfill the received request. The limited performance of the testbed, which was shared with other services too during our measurements, caused having large confidence intervals in transparent redirections. Transparent redirection requires way more computational overhead compared to the 302 redirection. The controller on each redirection has to install flow mods to the forwarders, while the 302 redirection requires no additional flow table changes. With increasing resources, the confidence intervals would get narrower. We should allocate more CPUs with the increasing number for forwarders in the network. In ideal conditions 1 CPU for a single forwarder.

5.3. Overall Delay and Bandwidth Measurements

Next we made delay and bandwidth measurements on a network with 2 hops between the client and the surrogate server and we compared the proposed transparent redirection with the 302 redirection side by side. The topology used is shown in Figure 15.

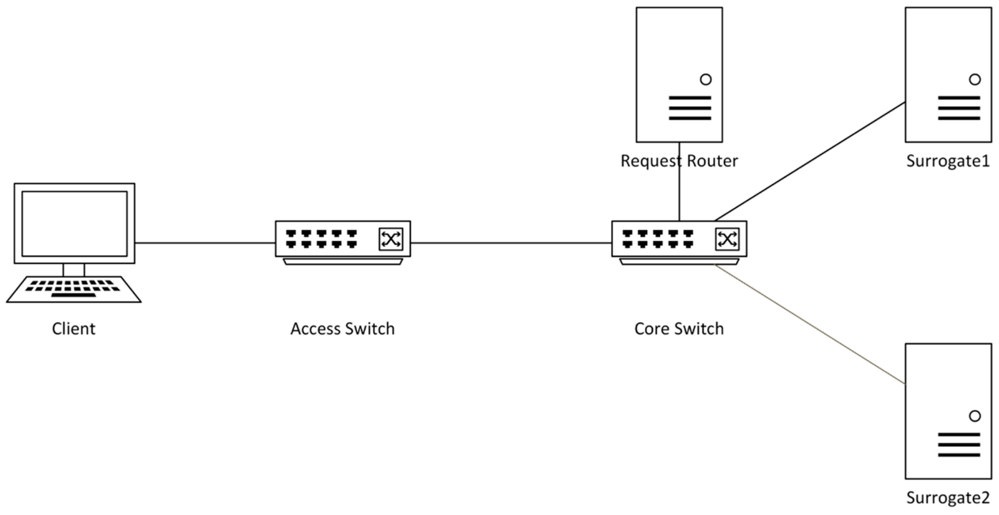

Figure 15. Topology for measuring overall redirection delays.

As already mentioned, the transparent redirection requires to be sent over the SDN controller, so the TCP session handshake traverses to the control node via the Packet IN and Packet OUT messages. This adds some initial delay to the 3-way handshake. The time required to establish a TCP connection with the request router is shown in Figure 16.

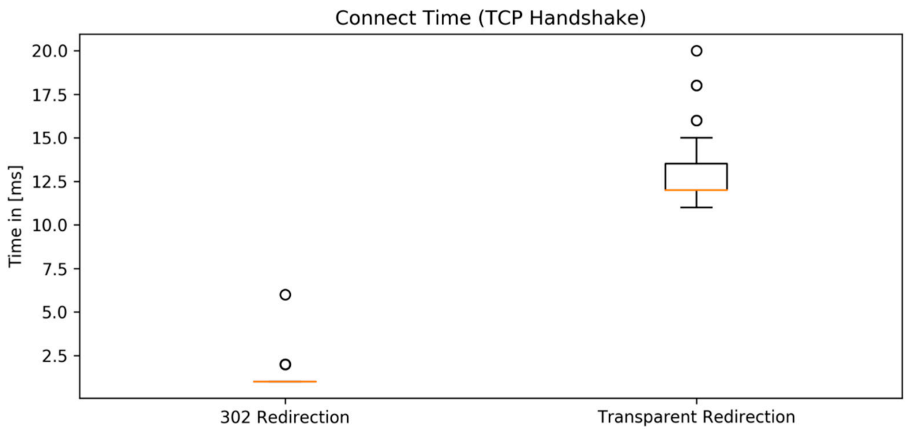

Figure 16. 3-way handshake delay.

Once the TCP session is established, the request router expects an HTTP GET message from the client. Upon a valid request, the client is redirected using a 302-redirection method toward the surrogate server, from where the content is served. Using the transparent redirection, a TCP session handover occurs to the surrogate server. Figure 17 shows the time required to process the redirection methods. The time between the sent HTTP GET message and receiving the first data bytes is measured.

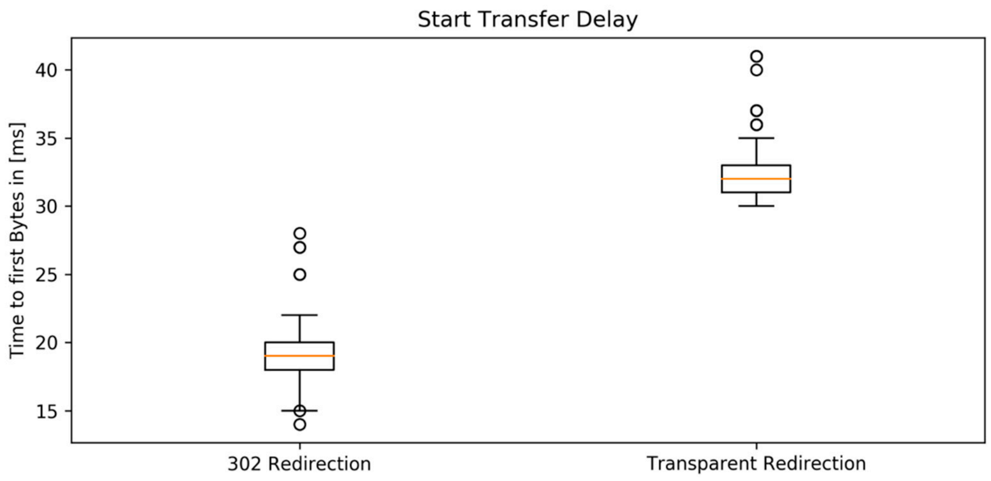

Figure 17. Time required to process redirection.

To give some insight into the performance of the data plane, we did measurements on a large file (1 GB). This size of the file is enough to reach high performance on the data plane due to the increasing window size. Figure 18 shows the whole time required to download the 1 GB file from the surrogate server including TCP session establishment and redirection.

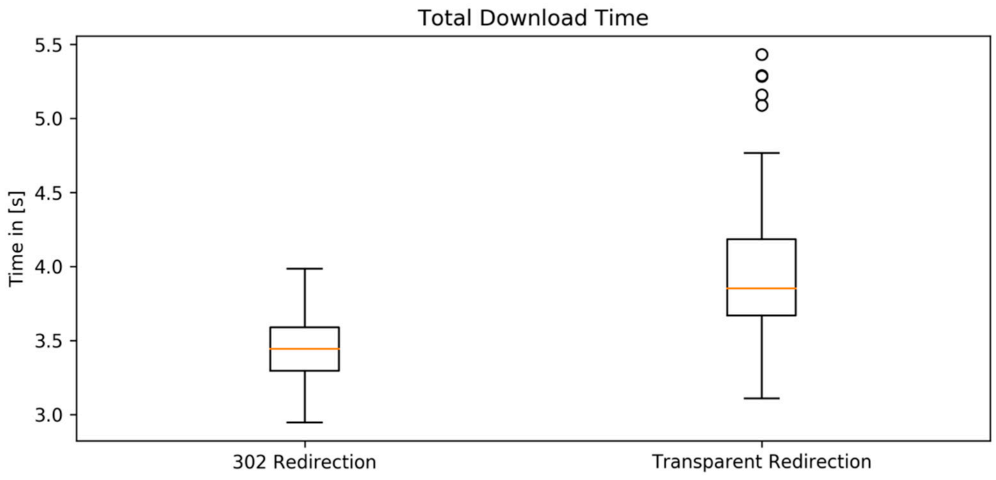

Figure 18. Total time required to download file.

After each test, we reported the mean bandwidth reached during the file transfer, which we put on a boxplot to evaluate the performance of the data plane itself. Figure 19 shows the average bandwidth reached during the file transfers. We have to note when 302 redirection is used the forwarding elements only match the packets and forward them to the next hop based on output action in the flow table. No other modifications are made to the packets. When transparent redirection is used, due to the TCP session synchronization, source IP, destination IP, source port and destination port are rewritten and sequence and acknowledge numbers are incremented. As we made our tests on an L2 network, source and destination hardware (MAC) addresses were rewritten too. This comes to a fact, that transparent redirection does 8 modifications on the packets, while the 302 redirection does not do any changes.

Any change applied to a TCP or IPv4 Header also triggers a recalculation of the CRC checksum. Based on the achieved bandwidth (~280 Mbit/s compared to ~320 Mbit/s), we can state that the modifications made on the Open vSwitch forwarder did not degrade the overall performance.

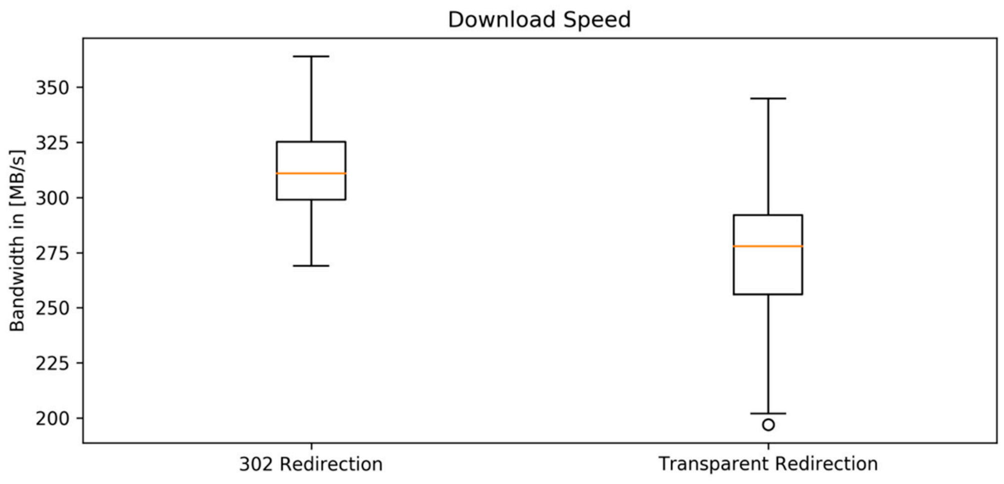

Figure 19. Average bandwidth reached.

5.4. Use Case of a Transparent Proxy

The proposed solution could be used as a transparent proxy, which in a load-balancing fashion distributes the incoming requests to multiple backend web services similarly, as presented in [23]. In the next set of tests, we compared the performance of the transparent redirection to a performance of an L7 proxy on the same topology (Figure 15). We also compared how the performance is affected when connections are kept alive for subsequent requests. The measurements are visible in Figures 20 and 21 with connection closed and kept alive after each request respectively.

As we can see, the L7 load balancing even outperforms 302-redirection in Figure 17 as the client does not have to process the redirection messages, while the transparent load balancing shows the same results as the transparent redirection as these procedures are the same.

When the connection is kept alive for subsequent requests, on average, the proposed transparent load balancing (transparent redirection) outperforms even the L7 load balancing as the subsequent requests do not have to pass the load balancer (request router) itself.

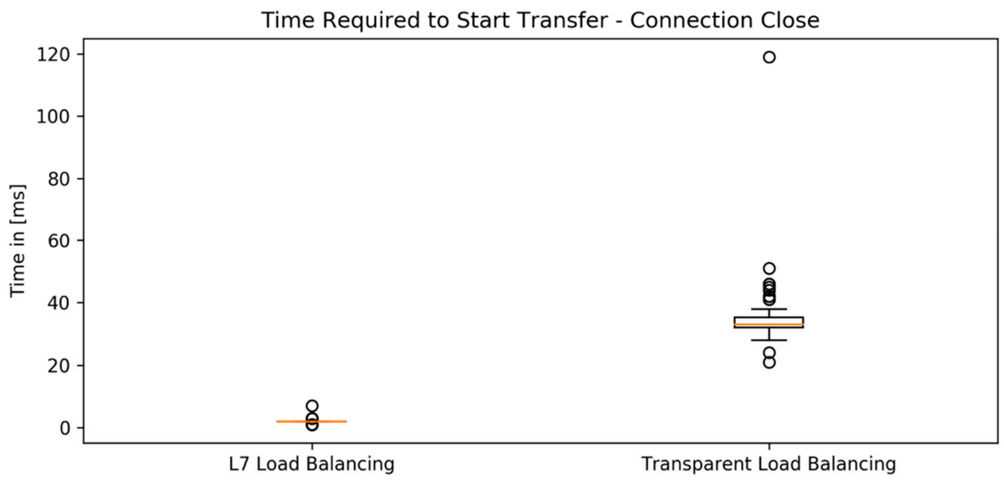

Figure 20. Time required to start transfer—connection close.

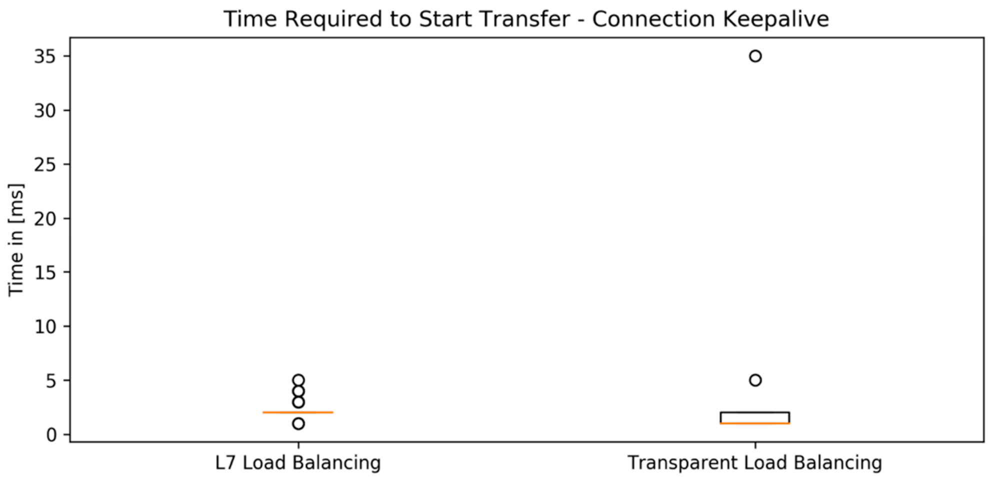

Figure 21. Time required to start transfer—connection keepalive.

6. Conclusions and Future Work

This article introduced the main concepts of Software Defined Networking, Content Delivery Network and the concepts of content distribution on a large scale. It further described existing redirection methods used in CDNs and devoted a section for existing transparent redirection methods. Transparent redirection methods are not often used as they require changes and service awareness in the network. Several transparent redirection methods were already implemented and tested; however, there is no transparent redirection method existing right now, which would not require modifications to the surrogate server. This limitation eliminates the use of legacy closed source solutions.

The article proposes a TCP session handoff mechanism as a transparent redirection which synchronizes two existing TCP connections with each other. This approach does not require any modifications to the endpoint. The only modification to the data plane is required on the core forwarders, where a surrogate server or a request router is attached. The transparent redirection ensures that the TCP session is handed off to a CDN surrogate from the request router, while the procedure stays unnoticeable to the endpoints. The proposed solution can be used in Load Balancing scenarios too.

The redirection decision is based on the network topology and the location of the surrogates and the users. The results acquired show that current redirection methods (302 Redirection) are faster than the proposed solution; however, this performance could be enhanced by using a more mature and faster controller. Currently, the controller represents a single point of failure, does not scale and represents our performance bottleneck in the proposed solution. In the case of a transparent load balancer, the solution achieved better results when the TCP connections were kept alive for subsequent requests. The achieved performance is suitable for production use and compared to existing solutions does not degrade the Quality of Experience for the end-users.

There were several issues found, which should be solved in the future. One of them is the issue of pre-establishing TCP sessions form the request router to the surrogate servers with various WSCALE options to be able to make exact matches when we synchronize the TCP sessions. The incrementation of the acknowledgement numbers in the selective acknowledgement TCP Option should be implemented on forwarding elements too. Currently, these TCP Options including TCP Timestamps are turned off, which could degrade the performance in challenging environments. Furthermore, secure TLS based TCP sessions currently are not supported by the proposed solution, so connections over HTTPs are broken.

Author Contributions: The project was analyzed, implemented and evaluated, including writing, by T.B. The high-performance load balancing use case was analyzed and implemented, and article revisions were done, by R.B. I.K. contributed with project supervision and article revisions.

Funding: This research was supported by Ministry of Education, Science, Research and Sport of the Slovak Republic, Incentives for Research and Development, grant agreement number: 2018/14427:1-26C0; and Operational Program Research and Innovation for the project: Research of advanced methods of intelligent information processing (ITMS code: NFP313010T570), co-financed by the European Regional Development Fund.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Pathan, A.M.K.; Buyya, R. *A Taxonomy and Survey of Content Delivery Networks, Grid Computing and Distributed Systems GRIDS Laboratory*; University of Melbourne: Parkville, Australia, 2006; Volume 148, pp. 1–44. ISBN 9783540778868. ISSN 1876110.
2. Boros, T.; Kotuliak, I. SDN-based transparent redirection for content delivery networks. In Proceedings of the 2019 42nd International Conference on Telecommunications and Signal Processing (TSP), Budapest, Hungary, 1–3 July 2019; pp. 373–377, ISBN 978-1-7281-1864-2. [CrossRef]
3. Thompson, K.; Miller, G.J.; Wilder, R. Wide-area internet traffic patterns and characteristics. *IEEE Network.* **1997**, *11*, 10–23. [CrossRef]
4. Cáceres, R.; Danzig, P.B.; Jamin, S.; Mitzel, D.J. Characteristics of wide-area TCP/IP conversations. In Proceedings of the ACM SIGCOMM, Zurich, Switzerland, 3–6 September 1991; Volume 21, pp. 101–112, ISBN 0-89791-444-9. [CrossRef]
5. Postel, J. Transmission Control Protocol, Darpa Internet Program, Protocol Specification, RFC 793. Available online: https://tools.ietf.org/html/rfc793 (accessed on 21 April 2019).
6. Fielding, R.; Reschke, J. (Eds.) Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests, RFC 7232. Available online: https://www.rfc-editor.org/info/rfc7232 (accessed on 21 April 2019).
7. Fielding, R.; Nottingham, M.; Reschke, J. (Eds.) Hypertext Transfer Protocol (HTTP/1.1): Caching, RFC 7234. Available online: https://www.rfc-editor.org/info/rfc7234 (accessed on 21 April 2019).
8. Barbir, A.; Cain, B.; Nair, R.; Spatscheck, O. Known Content Network (CN) Request-Routing Mechanisms, RFC 3568. Available online: https://www.rfc-editor.org/info/rfc3568 (accessed on 21 April 2019).
9. Mockapetris, P. Domain names—Concepts and facilities, STD 13, RFC 1034. Available online: https://www.rfc-editor.org/info/rfc1034 (accessed on 21 April 2019).
10. Tanenbaum, A.; van Steem, M. *"Server Clusters," in Distributed Systems—Principles and Paragdims*; Paerson Education: London, UK, 2006; ISBN 0-13-239227-5.
11. Hunt, G.; Nahum, E.; Tracey, J. *Enabling Content-Based Load Distribution for Scalable Services*; Watson Research Center: New York, NY, USA, 1997.
12. Braden, R. Extending TCP for Transactions—Concepts, RFC 1379. Available online: https://www.rfc-editor.org/info/rfc1379 (accessed on 21 April 2019).
13. Braden, R. T/TCP—TCP Extensions for Transactions Functional Specification, RFC 1644. Available online: https://www.rfc-editor.org/info/rfc1644 (accessed on 21 April 2019).
14. Hannum, C. Security Problems Associated with T/TCP. Cambridge, Massechusetts, 1996.
15. Hannum, C. T/TCP vulnerabilities. *Phrack Mag.* **1998**, *8*, 6–15.
16. Cooney, M. Getting corporate intranets under control. *Netw. World* **1996**, *13*, 17.
17. Wang, L. Design and Implementation of TCPHA 2005. (Draft Release).
18. Acharya, A.; Shaikh, A. Using mobility support for request-routing in IPv6 CDNs. In *7th International Workshop on Web Content Caching Distribution*; Watson Research Center: Hawthorne, NY, USA, 2002.
19. Johnson, D.; Perkins, C.; Arkko, J. Mobility Support in IPv6, RFC 3775. Available online: https://www.rfc-editor.org/info/rfc3775 (accessed on 21 April 2019).
20. Google Inc. IPv6 Statistics. 2019. Available online: https://www.google.com/intl/en/ipv6/statistics.html#tab=ipv6-adoption&tab=ipv6-adoption (accessed on 14 April 2019).
21. Wichtlhuber, M.; Reinecke, R.; Hausheer, D. An SDN-based CDN/ISP collaboration architecture for managing high-volume flows. *IEEE Trans. Netw. Serv. Manag.* **2015**, *12*, 48–60. [CrossRef]
22. Corbet, J. TCP connection repair. 2012. Available online: https://lwn.net/Articles/495304/ (accessed on 14 April 2019).

23. Hayakawa, Y.; Eggert, L.; Honda, M.; Santry, D. Prism: A proxy architecture for datacenter networks. In Proceedings of the 2017 Symposium on Cloud Computing—SoCC, Santa Clara, CA, USA, 24–27 September 2017; pp. 181–188, ISBN 9781450350280. [CrossRef]
24. Honda, M.; Huici, F.; Lettieri, G.; Rizzo, L. mSwitch: A highly-scalable, modular software switch. In Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research—SOSR '15, Santa Clara, CA, USA, 17–18 June 2015; pp. 1–13, ISBN 9781450334518. [CrossRef]
25. Nadeau Thomas, G.K. *SDN—Software Defined Networks*; O'Reilly Media Inc.: Sebastopol, CA, USA, 2013; pp. 1–9. ISBN 978-1449342302.
26. Open Networking Foundation. *Software-Defined Networking: The New Norm for Networks*; ONF White Paper: Menlo Park, CA, USA, 2012.
27. Heller, B. *OpenFlow Switch Specification version 1.3.5*; ONF: Menlo Park, CA, USA, 2009.
28. Fielding, R.; Gettys, J.; Mogul, J.; Frystyk, H.; Masinter, L.; Leach, P.; Berners-Lee, T. Hypertext Transfer Protocol—HTTP/1.1, RFC 2616. Available online: https://www.rfc-editor.org/info/rfc2616 (accessed on 21 April 2019).
29. Pfaff, B.; Pettit, J.; Koponen, T.; Jackson, E.; Zhou, A.; Rajahalme, J.; Gross, J.; Wang, A.; Stringer, J.; Shelar, P. The design and implementation of open vSwitch. In Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, Oakland, CA, USA, 4–6 May 2015; pp. 117–130.
30. Pan, H.-Y.; Wang, S.-Y. Optimizing the SDN control-plane performance of the Openvswitch software switch. In Proceedings of the 2015 IEEE Symposium on Computers and Communication (ISCC), Larnaca, Cyprus, 6–9 July 2015; pp. 403–408, ISBN 978-1-4673-7194-0. [CrossRef]
31. Shanmugalingam, S.; Ksentini, A.; Bertin, P. DPDK Open vSwitch performance validation with mirroring feature. In Proceedings of the 2016 23rd International Conference on Telecommunications (ICT), Thessaloniki, Greece, 16–18 May 2016; pp. 1–6, ISBN 978-1-5090-1990-8. [CrossRef]
32. Ashley, M.; Brown, A.; Jaißle, A.; Sahani, S.; Simpson, B.; Tierney, B. iPerf—The ultimate speed test tool for TCP, UDP and SCTP. Available online: https://iperf.fr/ (accessed on 21 April 2019).
33. Lantz, B.; Heller, B.; McKeown, N. A network in a laptop. In Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks—Hotnets '10, Monterey, CA, USA, 20–21 October 2010; pp. 1–6, ISBN 9781450304092. [CrossRef]
34. Ryu SDN Framework Community, COMPONENT-BASED SOFTWARE DEFINED NETWORKING FRAMEWORK: Build SDN Agilely. 2017. Available online: https://osrg.github.io/ryu/ (accessed on 30 September 2017).
35. Tilkov, S.; Vinoski, S. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Comput.* **2010**, *14*, 80–83. [CrossRef]
36. Greenstein, S. How Much Apache? *IEEE Micro* **2013**, *33*, 80. [CrossRef]
37. Reese, W. Nginx: The High-performance Web Server and Reverse Proxy. *Linux J.* **2008**, *173*. Article no. 2.
38. Borman, D.; Braden, B.; Jacobson, V.; Scheffenegger, R. (Eds.) TCP Extensions for High Performance, RFC 7323. Available online: https://www.rfc-editor.org/info/rfc7323 (accessed on 21 April 2019).
39. Mathis, M.; Mahdavi, J.; Floyd, S.; Romanow, A. TCP Selective Acknowledgment Options, RFC 2018. Available online: https://www.rfc-editor.org/info/rfc2018 (accessed on 21 April 2019).