Università degli Studi di Napoli Federico II

Facoltà di Ingegneria

Dottorato di Ricerca in Ingegneria Informatica ed Automatica
XXVII Ciclo
Dipartimento di Ingegneria Elettronica e delle Tecnologie dell'Informazione

Dynamic management of real-time multimedia services in SDN-enabled cloud infrastructures

Tobia Castaldi

Ph.D. Thesis

Tutors
Prof. Simon Pietro Romano

Coordinator
Prof. Francesco Garofalo

March 2015

Abstract

This thesis has been carried out in the field of Computer Networks and has been conducted in collaboration with my colleague Lorenzo Miniero.

We have looked after a challenging and popular application family over the Internet, namely Real-time Multimedia Applications. The proliferation of new network technologies that characterizes the last years has led to the birth of a variegated set of new complex services on top of the Internet. The network, which in its early days was mainly aimed at distributing static contents, is rapidly evolving toward a paradigm based on dynamic or even real time information. One of the most appealing services is the streaming of real time multimedia contents over the Internet. In this context, the most challenging goal for operators is to provide users with services that are comparable in terms of perceived quality to the traditional TV broadcasting system. At the time of this writing the whole portfolio of real time multimedia streaming services is characterized by some common issues. First of all the need for users to install third party software or *plugins* for web browsers in order to be able to reproduce multimedia streams. It's worth noting that different services rely upon different software solutions and in most of the cases they are not inter-operable. Furthermore, existing streaming services are afflicted by high transmission delay. This fact can be very annoying when users are provided with a feedback channel to interact with the source of the streaming. For example, if during a *webinar* (online seminar) participants can interact with the current speaker, even a few seconds delay could cause misunderstandings and slow down the progress of the event. As another example, it must be said that one of the most common use cases when it comes to streaming is the broadcast of sport events. In this case, someone in the neighbourhood watching the same event on television could rejoice and scream, hence spoiling the surprise for a scored goal.

To cope with such issues, we have worked on a novel architecture aimed to provide a web-based real-time multimedia streaming service capable to serve a very large number of users. The framework we propose leverages techniques that have been successfully exploited in Content Delivery Networks (CDN)

at the application level and in the field of the so called Software Defined Networking (SDN) for what concerns the underlying network infrastructure. Our platform envisages the cloud-based deployment of several elements co-operating to deliver the service to end users. More specifically, the server side of our architecture is based on an overlay network of application level components configured to create a self-organized tree topology. Each node of the tree plays one of the following roles: (i) *root* (ii) *relay* (iii) *leaf*. The source of the stream, i.e. the broadcaster, sends multimedia flows to the *root* node which transcodes them in order to generate different quality replicas of the original flows. When new users try to access the stream (by simply opening a web page) they are directed by the framework to one of the leaves of the tree depending on the overall status of the network and/or on particular client side constraints (type of device, available bandwidth, etc.). A distributed algorithm is used to provide the root node with information about the overall load status of the network and to allow it to dynamically manage the content distribution chain (by adding or removing nodes). As for the real time requirement, communication among broadcaster, internal nodes and final users relies entirely upon the upcoming WebRTC/RtcWeb (Real-Time Communication in Web-browsers) technology, which guarantees pure web based access to multimedia streams, as well as support for real time delivery of multimedia contents.

In the second part of the thesis, we focus on SDN enabled networks, once again basing our work on a study of current architectural and performance challenges. The acronym SDN stands for Software Defined Networks and it represents an innovative approach to networking, attempting to keep the pace of the substantial diffusion and utilization we are experiencing in the latest years. By developing an application on the top of an SDN, a network administrator can obtain a substantial degree of scalability and reconfiguration that they would not be able to reach in any other way. We hence propose a framework for the performance optimization of the above mentioned streaming platform, as well as for the simplification of its management and administration.

The thesis is structured as follows. Section 1 helps position our work by providing useful information about the reference context, as well as about the motivations behind our contribution. At the end of the section a high level design of the project is presented as well. An overview of the main technologies we based our project on is presented in Section 2. High level implementation details are illustrated in Section 3 whereas in Section 4 we deal with a class of technologies allowing to optimize the monitoring and control of the architecture thanks to the design and implementation of low level components presented respectively in Section 5 and Section 6. Section 7 presents a functional testing campaign we carried out to validate the prototype we realized and to demonstrate how the system is able to satisfy all the requirements identified during the design phase.

Finally, section 8 provides some concluding remarks, together with information about our future work.

Contents

Chapter 1

Streaming of *real-time* contents over the Internet

In this section we will deal with a very common application scenario in the wide panorama of real-time multimedia services, namely the streaming of multimedia contents over the Internet. We will provide a general background of this kind of service and then we will focus on a specific use case envisaging the real time transmission of live generated contents. Some information will be given about adopted technical solutions and open issues affecting the most common commercial products. At the end of this section we will present, from an high level point of view, our idea for the realization of an innovative framework supporting, in general, real-time applications deployed in cloud environments. The platform has been designed and realized having in mind specifically the aforementioned streaming service use case which represents the main objective of our doctoral research activity and that, for this reason, it will be the *fil rouge* that will guide the reader across the entire document.

1.1 General background

The Internet is more and more frequently used as an infrastructure for transmitting real-time data and for providing geographically distributed users with advanced tools for mutual interaction and communication. *Real-time* multimedia communication is one of the *killer application* of the modern Internet indeed. It imposes a number of stringent requirements to the underlying network infrastructure. First of all, the intrinsic multimedia nature of a communication session (which in its most general form involves a combination of audio, video, instant messaging, desktop sharing, etc.) requires coping with complex management issues. Second, the real-time requirement of network-based interactions demands a high level of Quality of Service (QoS). Within this general context, one of the most appealing services is the *streaming of real-time multimedia contents.* The term *streaming* indicates that one or more media (audio, video, etc.) are constantly received by and presented to an end-user while being delivered by a provider. With reference to the streaming of audio/video flows, we can distinguish two main categories of contents to be streamed: a stored video file or a live video flow generated using an acquisition device like a webcam.

As for the transmission mechanism of a stored video file two possible paradigms have to be considered, namely, (i)the download mode and (ii) the streaming mode. In the download mode, the end-user downloads the entire file and, only at end of this process, is able to play out its video content. Obviously, the full file transfer lead to very long delay before the user can start the play out phase.

By adopting the streaming mode, instead, the video can be played out as soon as just a minimum fraction of the whole file has been received and the client has enough data to decode the first part of the video content. The video streaming scenario typically presents bandwidth, delay and packet loss requirements that are very ambitious to be respected due to the best-effort nature of the Internet. In fact, the network does not offer any quality of service (QoS) guarantees to the streaming process thus designing mechanisms and protocols for this kind of service poses several challenges.

For instance, a *multicast*[26] based delivery of a stream would be a very efficient way to transfer data to multiple destinations. The main drawback of this technique is it requires the underlaying network infrastructure supports this kind of technology and, for this reason, nowadays it is exploited mainly when the involved producer and receivers of the flow are limited in the context of the same *Local Area Network* (LAN). Nevertheless, some classes of applications have tried to obtain the great advantages introduced by multicast transmission moving the techniques adopted at the network level to the application level of the protocols stack realizing the so called *application layer multicast-over-unicast*[16]). In an overlay or end-system multicast approach participating peers organize themselves into an overlay topology for data delivery. Each edge in this topology corresponds to a unicast path between two end-systems or peers in the underlying Internet. All multicast-related functionalities are implemented at the peers instead of at routers, and the goal of the multicast protocol is to construct and maintain an efficient overlay for data transmission.

1.1.1 Real-time streaming protocols

As for the control and media delivery functions, the main standardization bodies, like the *Internet Engineering Task Force* (IETF) [19] and the *Third Generation Partnership Project* (3GPP) [11], have defined a wide set of protocols that can be exploited for the purpose. Among them, the *Real-time Streaming Protocol* (RTSP) [55] were specifically designed to stream media over networks. The protocol can be used for establishing and controlling media sessions between end points and to facilitate real-time control of playback of the media stream received from the server (e.g., pause, stop, start, etc.). The protocol is not responsible for the transmission of data itself and this task is in the most of the cases demanded to dedicated protocols like the *Real-time Transport Protocol* (RTP) [56]. The *Real Time Messaging Protocol* (RTMP) [41] can be considered the proprietary *equivalent* of the RTSP. It was initially a proprietary protocol developed by Macromedia for streaming audio, video and data over the Internet, between a Flash player and a server. An incomplete version of the specification of the protocol has been lately released for public use.

Another approach that seems to incorporate both the advantages of using a standard web protocol and the ability to be used for streaming even live content is the *adaptive bitrate streaming.* HTTP adaptive bitrate streaming [60] is based on HTTP progressive download of a sequence of small HTTP-based file segments, each segment containing a short interval of playback time of the main content.

1.1.2 Live streaming: open issues

As we anticipated, the most extreme form of streaming is *live streaming*, namely the delivery of live contents over the Internet. This scenario requires a form of source device (e.g. a web cam, an audio acquisition interface, screen capture software, etc.), an encoder to digitize the content, a media publisher, and a content delivery network to distribute and deliver the content to end users. In this field, the most challenging goal for service providers is to provide users with a perceived quality of service comparable to the one obtained by means of the traditional TV broadcasting system. At the time of this writing the whole portfolio of Internet-based, real time multimedia streaming services (e.g., Twitch.tv [27], YouTube Live [44], Ustream [30], etc.), in spite of being able to offer the service to a wide audience of end users, is characterized by some common issues. First of all the need for users to install third party software or plugins for web browsers in order to be able to play-out multimedia streams. It's worth noting that different services rely upon different proprietary software solutions and in most of the cases they are not inter-operable. Furthermore, existing streaming services are afflicted by high transmission delay. This fact can be very annoying when users are provided with a feedback channel to interact with the source of the streaming. For example, if during a *webinar* (an online live seminar) remote participants can interact with the current speaker, even a few seconds delay could cause misunderstandings and slow down the progress of the event. As another example, it must be said that one of the most common use cases when it comes to streaming is the broadcast of sport events. In this case, someone in the neighborhood watching the same event on television could joy and screams spoiling the surprise for the decisive actions of a match.

1.2 SOLEIL: Streaming Of Large scale Events over Internet cLouds

As we said existing solutions in the field of live streaming are able to serve a potentially huge number of end users by using CDN-like server side architectures but they are all afflicted by two main drawbacks: i) need for proprietary software installations (ad-hoc applications or web browser plug-ins) in order to access the service ii) *high* transmission delays. To cope with such issues, we have worked on a novel architecture aimed to provide an extremely large number of users with a *fully web-based* access to a *real-time* multimedia streaming service.

1.2.1 System requirements

The framework we propose still leverages techniques that have been successfully exploited in the context of the *Content Delivery Networks* (CDN) at the application level but also rely upon the benefits introduced by the so called *Software Defined Networking* (SDN) for what concerns the underlying network infrastructure. The system has to respect several functional requirements:

- *Scalability.* The performance of the system has not to be affected by the number of clients accessing the service. In the specific case of a streaming application, this means that the elements characterizing the delivered stream (quality, delay, jitter, etc.) has to be coherent to the imposed service level imposed by the system administrator no matter how many clients are currently connected to the system;
- *Reconfiguration.* The infrastructure design should allow a certain degree of flexibility in order to react to peculiar operational scenarios.

In the context of a streaming application, for instance, a high number of users willing to access the service could be localized in the same geographic area. In this case, the infrastructure should be able to automatically reconfigure itself in order to properly serve the requests;

- *Fault tolerance.* As a specific reconfiguration use case, we can include the reaction to network problems affecting the system infrastructure. In case of network congestion, for instance, the infrastructure should be able to reconfigure itself in order to avoid or at least minimize the service *donwntime*;
- *User-friendly interface.* The system administrator should be able to manage the overall infrastructure by means of a simple web based administration application, no matter they are aware or not of the underlying technology and of implementation details;
- *Customization and monitoring.* By means of the same application, administrators should also be able to customize the infrastructure behavior and to monitor its current state (CPU load and memory utilization of each component, number of exchanged packets, packet loss information, etc.);
- *Security.* All involved communications should be properly encrypted. Only authorized users should be able to interact with the system.

1.2.2 Overall architecture

The server side of the platform is composed by several elements cooperating to deliver the final service (this approach naturally lends itself to a cloud-based deployment). Our architecture is based on overlay network whose nodes create a self-organized tree topology. Analyzing the high level perspective of the system depicted in Figure 1.1, we can distinguish three different types of components:

- The *root node*, namely the first node of the tree. This element receives the live content by the source of the stream and inject it into the system. This will be the point of the system in which the original media streams will be adapted and converted into different formats and resolutions in order to cope with different client-side requirements (type of device used for accessing the service, bandwidth availability and so on). The root node will be also in charge of the management of the entire system. In fact, as will be detailed in the following, this node will receive statistic information about load and bandwidth consumption from the other nodes of the tree and it will be able to react properly in case of problems;

- *Relay node*, it's the role assigned to *internal* nodes of the tree. A relay node simply act as a bridge toward other nodes of the system in order to increase scalability of the overall system. Nevertheless, this kind of node will be crucial for both balancing the load of the system and for aggregating statistic information to send to the *root* node. The number of *relay nodes* is dynamic and depends on the number of end users willing to access the service in a specific instant of time. New nodes can dynamically be added to the tree in order to increase the scalability of the system. Vice versa, in order to save resources, a node can be *switched off* in case a large number of users (previously connected to the stream) disconnect and leave the system;

- *Leaf node*, the last node of the tree, the one end user will connect to in order to receive the live stream. This node represent the so-called *last-mile* of the overall architecture and it has to allow final users to access the service in a easy and standard way by means of a pure web interface.

It is worth noting that, in order this kind of architecture could work properly, communication among broadcaster, internal nodes and final users has to relay entirely upon a *very fast* communication channels. Our idea is to re-

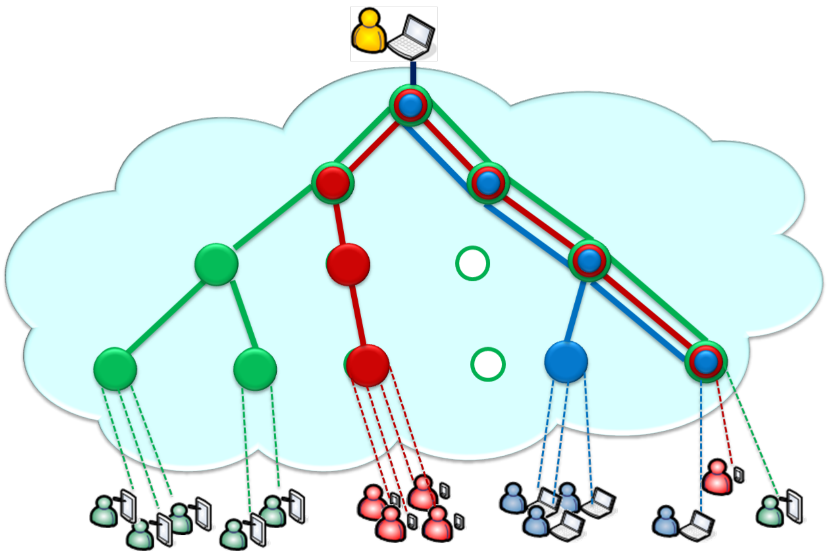


Figure 1.1: SOLEIL: High level architecture

alize these channels by exploiting the standard *Real-Time Transport Protocol* (RTP). Several reasons motivated us to adopt the RTP, first of all it's one of the most widely adopted protocols within the context of standards-based Voice over IP (VoIP) applications. Furthermore, it can be used in conjunction with the *RTP Control Protocol* (RTCP) [54] which allows to monitor transmission statistics and quality of service (QoS) information. Last but not least, RTP is the underlying technology of the upcoming *WebRTC/RtcWeb (Real-Time Communication in Web-browsers)* [32] standard, which guarantees pure web based access to multimedia streams. As we said the *root* node can generate different 'copies'of the original source in order to provide users with limited resources with an *adapted* version of the media stream. One possible option is the adoption of *Scalable Video Coding* (SVC) [57] techniques. The objective of the SVC standardization activity has indeed been to enable the encoding of a high-quality video *bitstream* that contains one or more subset *bitstreams* that can themselves be decoded with a complexity and reconstruction quality similar to that achieved using the whole data. One of the most important aspects of our architecture is the monitoring and management of the overlay network in charge of providing the service. As anticipated this task will be realized by collecting and aggregating statistic information, as will be explained in the following section.

1.2.3 Aggregation of real-time statistics

In order to let the source node properly monitor and manage the whole infrastructure, we need an ad-hoc protocol based on the information about the quality of service provided by each end user by means of the RTCP protocol. In fact the RTCP protocol is natively designed for supporting one-to-one communication. In this context, if a peer of the communication is experiencing some network problems (packet loss, delay, etc.), it can send to the other peer a report about the quality of the received media flow and ask it to adjust the transmission rate in order to solve that issue and to prevent further problems in the near future. However, in our scenario several clients can be connected to the same *leaf* node and each client can obviously access the node from a different location and it can use connections characterized by heterogeneous network conditions. Consequently, the statistic information sent by each client can be very discrepant from each others. If we thought to adapt the transmission rate of each node on the base of the *rough* information sent by end users we would risk to degrade the quality of the delivered stream for all the users served by the same sub-tree of the topology even though only one of those users is accessing the service through a *poor* Internet connection. For this reason we need to introduce a mechanism that allows the system to *understand* whether the problem is common to most of the users or only to a few of them. The general idea is to aggregate statistical information in each node and to transmit the aggregate to the immediately upper node of the topology. The receiving node (a relay node), in turn, assigns a weight to each statistic based on the number of end users served by the subtree that has sent the information and forwards the statistic to its direct predecessor in the topology. This process is iterated in order to reach the root node.

This kind of information allows to properly monitor the overall status of the system. Once an alert about the status of a subtree or of a single node of the system arises, the root node can investigate, by means of direct messages, the specific load condition of that part of the system and, in case the problem persists, can act on the current topology by adding, removing or migrating nodes or even an entire subtree.

1.2.4 Migration of system's nodes

End users access the service by contacting a web based front end which redirects each client to the *best* leaf node according to configurable policies (e.g. current load of the system, user's geographic location, *type* of stream the user want access to, etc.). Due to the dynamic nature of the service the architecture provide support for, one or more of the selected leaf nodes can be overloaded by client connections or even be completely unused. In such a situation the system could decide that a migration of that node is needed in order to optimize the consumption of resources of the overall architecture and to serve client requests granting a satisfying level of *Quality of Service.* The migration, in fact, can be executed in order to move an overloaded node toward a new location able to provide both higher computational power and wider bandwidth resources. The migration process can be a very difficult operation in the context of our complex infrastructure since the node to be migrated is contributing to delivering the service. The system has to be able to achieve this task avoiding service interruptions and minimizing annoying side-effects for the end users.

Chapter 2

High level enabling technologies

2.1 CDN: Content Delivery Network

A *Content Delivery Network* (CDN) [62] is a large distributed system of servers deployed in multiple data centers across the Internet in order to provide heterogeneous contents to end-users with high availability and high performance. Nowadays CDNs contribute to host and distribute a large fraction of Internet contents, including *web objects*, multimedia files and applications. Content and service providers pay CDN operators to host and deliver their contents to the final customers for both granting them high level of QoS and for offloading their own server side infrastructure. In addition, CDNs provide the content provider a degree of protection from *Denial of Service* (DOS) attacks by using their large distributed server infrastructure to cope with malicious client requests.

In Figure 2.1 is represented the standard client-server paradigm used to serve content to end users. This approach can lead to unpredictable content delivery delay especially if an high number of requests have to be served *at same time*. In fact, the Internet is a network characterized by a *best effort* quality of service level and this fact makes it very difficult to guarantee well defined performance degree. Furthermore, the load on the Internet and the richness of its content continue to grow and any increase in available network

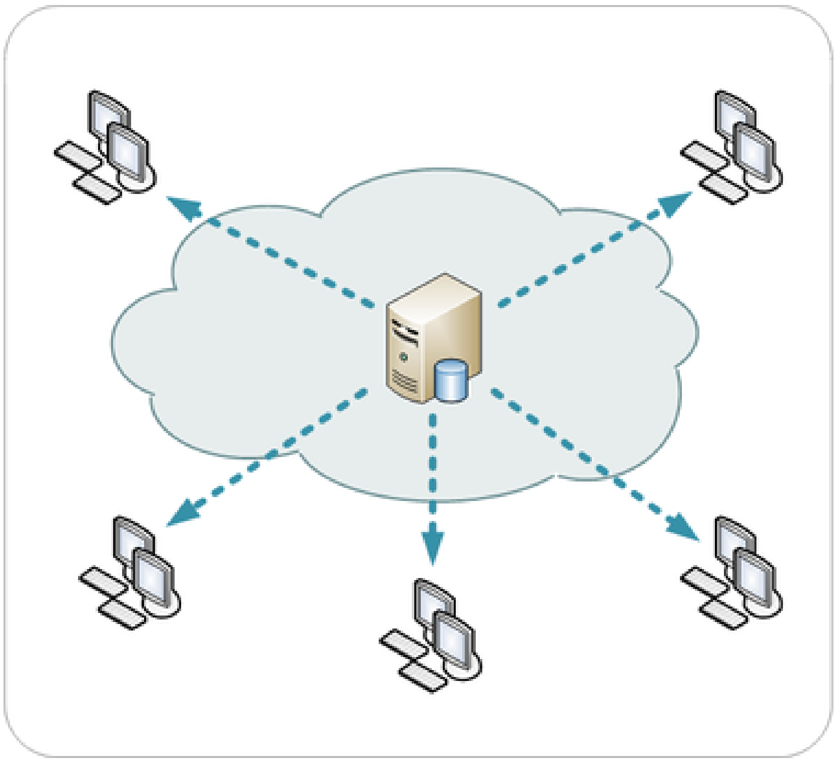


Figure 2.1: The centraliazed client-server paradigm

bandwidth and server capacity will continue to be overwhelmed.

To cope with such issues the aforementioned paradigm has been replaced by the approach depicted in Figure 2.2. The general idea behind a CDN envisages the contents are replicated on multiple servers hosting the business logic needed to optimize the delivery to end users. This results in an integrated overlay using multiple technologies (i.e. web caching, server-load balancing and request routing) to augment service availability and to reduce delivery delay.

In the following we will quickly present the aforementioned techniques in order to give the reader an overall idea of the foundations of the CDN approach:

- *Web caches* (or HTTP caches) temporary store copies of popular content on servers that have the greatest demand for the object requested.

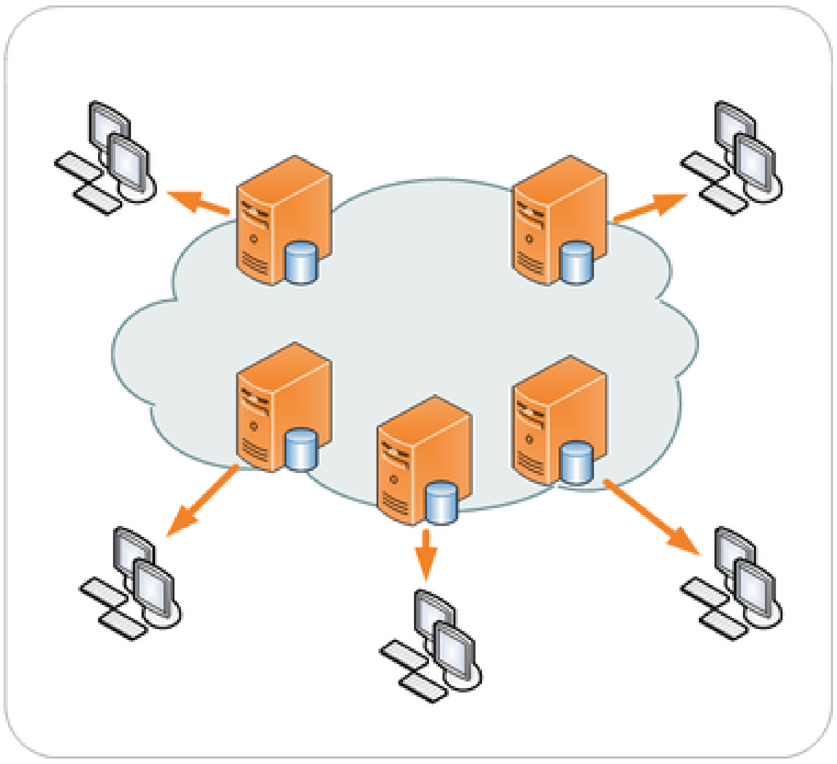


Figure 2.2: Content Delivery Network general architecture

These shared network appliances reduce bandwidth usage, server load, and perceived lag;

- *Server-load balancing* is used to share traffic among a number of servers or web caches. A single virtual name is assigned to a front-end element which is in charge of redirecting requests to one of the real web servers composing the back-end portion of the infrastructure. This technique has the advantage of balancing load, improving scalability and providing increased reliability by redistributing, for example, the load of a crashed server. A load balancing system performs in fact check operations in order to vault the health status of each server over time;

- A *Request routing* system directs client requests to the best serving node in a distributed system. Clients' requests can be redirected for instance to the closest server or to the one with the most capacity in that moment. A variety of algorithms are used to route the request including *Global Server Load Balancing*, *DNS-based request routing*, *Dynamic metafile generation*, *HTML rewriting* and *anycasting*.

In [42] are illustrated the main building blocks of CDNs (see Fig. 2.3). Fundamental building blocks are those devoted to distribution and replication of contents and the one devoted to request redirection. The distribution and replication module is the interface between the CDN operator and the content provider. It deals with three main issues:

- Server placement, i.e., strategies for the positioning of replica servers;

- Content selection, i.e., strategies for choosing the contents that have to be replicated on surrogates;

- Content outsourcing, i.e., algorithms for the replication of contents on replica servers.

The redirection block on the other hand is the interface between the CDN and the final user. It handles users' request and their redirection towards the CDN's surrogates containing the requested contents.

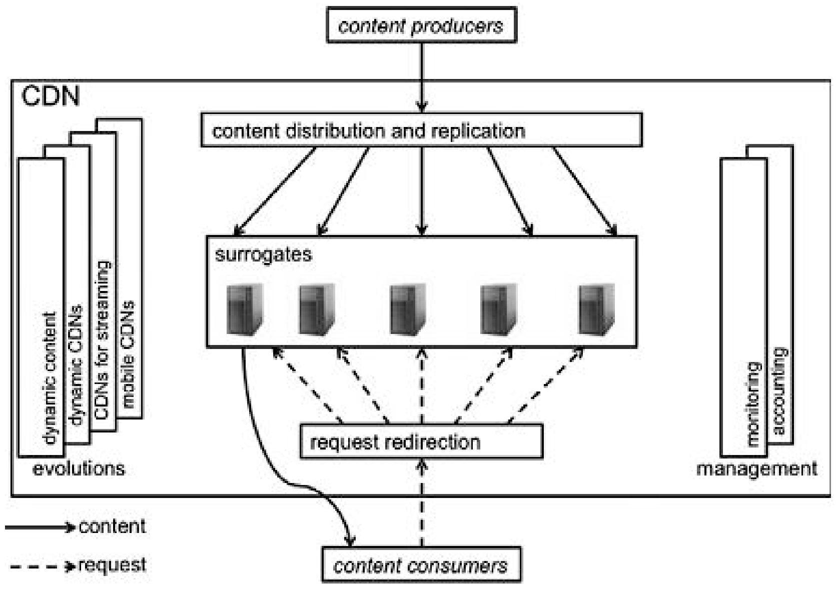


Figure 2.3: CDN building blocks

Other conceptual blocks are related to monitoring and accounting functionality. The monitoring is responsible of controlling CDN resources' state (servers' workload, network workload, and so on). The accounting module controls how contents belonging to each CP client overload the CDN network by looking at parameters such as the traffic generated by a given replicated web site.

Since their birth in the 90s, CDNs have been facing several challenges due to the evolution of the usage patterns of the Internet and new architectural modules have been added besides the aforementioned basic ones. They deal with:

- Dynamic content distribution, i.e., the replication of contents that have to be generated ad-hoc according to the received request;
- Dynamic resource deployment, in order to accomplish overload conditions and flash crowd;
- Users' mobility support;
- Multimedia streaming support.

2.1.1 Content selection

When designing the CDN replication policy for a certain CP's site, there are two main alternatives:

- performing a full-site replication: in this case, the whole site is replicated on CDN surrogates;
- performing a partial-site replication: here, only a subset of objects is replicated.

Partial-site strategies are preferred since more scalable when dealing with lots of bulky objects. However, a further complexity resides behind. Indeed, a proper content selection criterion has to be identified and applied. Typically, web objects are clustered according to a certain similarity metrics that can be based for example on object's popularity, number of incoming hyperlinks, or the reduction in the access latency that could be obtained by replicating the considered object.

2.1.2 Content outsourcing

Content outsourcing strategies can be *push-based* or *pull-based.*

Push-based mechanisms envision content pre-fetching on CDN surrogates, i.e., the replication is performed in a pro-active manner without waiting for final users' requests. Strategies of such a kind are usually strictly interconnected with partial-site replication policies of content selection. Push-based outsourcing is not exploited in commercial CDNs since it can be effectively applied only if the spatial and temporal distribution of users' request can be known a-priori. That hypotesis is very interesting but at the moment it is also very hard that it can be satisfied in practice.

Pull-based strategies, on the other hand, replicate contents on a surrogate server only after a 'miss event'. A miss event happens when a request reaches a server that does not hold the requested content. They can be further divided in 'cooperative' and 'non-cooperative' solutions.

Non-cooperative pull-based strategies correspond to conventional caching. Only contents already requested by users are replicated on surrogate servers. This policy realizes a user-driven automatic content replication on the CDN. This solution is the simplest one and is often employed in commercial CDNs as Akamai [38].

Cooperative pull-based solutions are different only in the management of the miss event: the missing content is requested to the other cooperating CDN servers rather than to the origin server, by this way reducing the load on the origin server and possibly enhance time performances.

2.1.3 Server selection

A server selection algorithm is used for selecting the surrogate server to which redirect a certain request among those that can satisfy that request. Server selection algorithms can be divided in (i) server-side and client-side and in (ii) single-server and multiple-server .

There are two algorithm families of server-side algorithm:

- adaptive: they do not take into account server load, but apply static policies. A typical example is a round-robin strategy.
- non-adaptive: they consider some information in the selection process, such as server load, the user proximity, and the like. Akamai adopts algorithm of such a kind.

Client-side algorithms envision the user selects the preferred replicas among the available ones. The CDN operator provides a list of information associated with the available replicas and the user can use this information in the selection process.

In single-server strategies, the content is taken from a single selected server, while in multiple-server the content is divided into multiple blocks and each block resides on a different server. Single-server strategies are preferred in case of non-bulky contents; in this last case, which is typical for multimedia contents, the object can be split among different replicas.

2.1.4 Dynamic content management

A web application generating dynamic contents is developed into different levels:

- front-end: the interface with the final user, i.e., the web page dynamically generated;
- application: the level responsible of generating the web page on the basis of the application logic;
- back-end: the database level containing data exploited by the application logic;
- user-profile: the database level containing user profiles enabling the contents personalization.

The replication on CDNs of sites of that kind can be performed at different levels. Replication at the first level corresponds to static contents replications, which is the one considered up to now. The replication of the application level can be useless if the bottleneck is the access to the database maintained by the origin server. Database replication can be performed in three ways: (i) context-blind caching, (ii) context-aware caching, (iii) full-replication. The full replication is quite onerous if it has to be performed on several surrogates dispersed across the network. Content-blind caching memorizes on replicas only the content resulting from previously issued requests and it is the typical commercial option. In the context-aware caching strategies, surrogates have a database which is populated in a pro-active manner through a partial replication of the original database. The pro-active replication is driven by the surrogate users' access patterns. Even if more flexible, such an approach calls for a centralized control for the distribution of the requests among the multiple copies of the database and for the coordinated query execution. This strategy is typically employed for user profiles replication, since they are accessed from a single surrogate at a time in a localized manner.

2.1.5 Handling multimedia contents

CDNs evolve in time to better support multimedia streaming applications. Multimedia contents of bulky dimensions are typically divided into sub-parts, both in time domain and quality domains, when progressive encoding algorithms are applied. The full replication of such file on surrogate servers can be inefficient, since it generates a lot of traffic on the network and since users can be interested only in limited portions of the content. Typically, a CDN solution for the distribution of multimedia contents is a two level architecture. At the higher level, there is the transport of the content between the origin server and the surrogate server by means of typical file transfer protocols. At the lower level, there is the transport of the content between the surrogate server and the final user via RTP, controlled in case through protocols such as RTSP (Real Time Streaming Protocol) that enable the final user to perform typical VCR commands. The preferred content outsourcing strategy is a non-cooperative pull-based approach where surrogates possibly pre-fetch part of the contents that will be requested by the user to the aim of avoiding undesired interruptions during the reproduction. Live streaming (also known as webcasting) is more challenging on the architectural design plane. Live contents can not be pre-fetched on surrogates and the content distribution must be performed in real-time. When CDNs are largely expanded across the globe, webcasting of popular contents can really stress the content distribution network and the origin server.

2.2 Virtualization

In this section we will give an overview about virtualization technologies. We will first try to define what it exactly is, and then we will introduce different virtualization techniques. Finally, we will focus on most common disk sharing systems, which are crucial for some aspects of our project as will be explained in Section 2.2.3. Interested readers can find more exhaustive information in [34] and [48].

When talking about virtualization most of the people immediately think to modern virtualization software, like *VMWare* [52] or *VirtualBox* [47]. Actually, virtualization is a much older technology widely adopted since the early 90s. In fact, as an example of virtualization we can cite the virtual memory, an abstraction that makes a generic program to think of having the entire memory available for exclusive use, or the logical unit division of an hard disk, which makes the user think of having two or more hard disks, even tough actually only one physical device is available. During last years this technology evolved, so that nowadays virtualization can be defined as *the process of creating a virtual version of a device or resource*. Resources can range from plain memory to an entire server with its own operating system and peripherals.

Benefits of virtualization

Even though in the literature we can find several examples illustrating the benefits of virtualizing memory and hard disks, very little explanation is given about the advantages of virtualization an entire operating system. The most common (and often only) exemplification envisages the system testing before its deploying in production environments.

There are three main reasons:

- Partitioning: this is the ability to run multiple instances of an operative system or application on the same physical host. Modern x86-based processors are designed so that just one operative system and application can be run at one time. This would make so that even little data centers had to run multiple servers, leading to a highly inefficient resource utilization. In short, thanks to virtualization, it is possible to attain up to 80% of efficiency from a node;

- Isolation: each virtual machine is in fact isolated from the other VMs running on the same machine and from the host, too. This means that any crash or virus relating to a single VM is contained in the software process the VM itself belongs to. In other words, the fact that a virtualized server is experiencing problems does not affect the physical node itself and any other virtual machine contained in it. At most, it may have influence on other VMs that are currently communicating with the crashed/infected one, but the problem can be solved with much more ease than if the problem interested the entire host;

- Encapsulation: unlike a real server, a VM is just a plain file. This means that if, for example, the VM should be infected by some viruses, a simple deletion of the infected file would solve the problem. And if, like always happens, the system administrator has some backup copy of the VM itself, the entire server could be up again in a very short time, like if nothing actually had happened. For the same reason, a new server installation is simply a file copy, so that it is possible to set up an entire complex system with very little effort.

2.2.1 Virtualization types

A preliminary distinction can be done between hardware and software virtualization. Specifically, just the first one identifies an actual virtual machine running an operating system with its related applications, while the second one is just about a particular ways to emulate a certain kind of environment or improve certain parameter. In the following, we will use the expression *guest operating system* (or simply *guest*) to refer to an OS instance installed on a virtual machine.

Hardware based virtualization

This is the most common type of virtualization and it refers to the creation of virtual (as opposed to concrete) versions of computers and operating systems. Hardware virtualization has many advantages because controlling virtual machines is much easier than controlling a physical server. This technique consists in abstracting underlying physical resources like processor, memory, etc. and in providing the guest OS with a virtual machine equipped with its own virtualized resources.The guest OS is completely unaware of being virtualized and cannot tell the difference between virtualized and real hardware. At its origins, the software that controlled virtualization was called a 'control program', but the terms 'hypervisor'or 'virtual machine monitor'are now preferred

According to the amount of virtualized resources presented to the guest, we can distinguish among different sub-families of hardware virtualization. We can considers at least three different types of hardware based virtualization: *full*, *partial* and *paravirtualization*:

- With reference to Figure 2.4 we can see how with full hardware virtualization all resources are virtualized and the guest system can run unmodified with its own kernel. This kind of technique is the most powerful and efficient one, but in the case of x86 processors, it requires specific extensions in order to work properly. *VMWare* is an example of this kind of virtualization;
- The partial virtualization (Figure 2.4) technique let to virtualize just a subset of the original hardware. An example of it is virtual memory;
- Paravirtualization consists in abstracting the whole underlying hardware as an API(Application Programming Interface) similar to the original one. This API, acting as a proxy, will forward every system call the corresponding OS functions. Of course, in this case, the guest OS has to be aware of being virtualized and so cannot be run as it is (like it happens adopting full hardware virtualization), but requires some customization. *Xen* servers implement this kind of hardware virtualization.

Software based virtualization

Software based virtualization includes technologies aimed at delivering some kind of service that would not be possible by using a specific host operating system. We can distinguish between two types of software techniques:

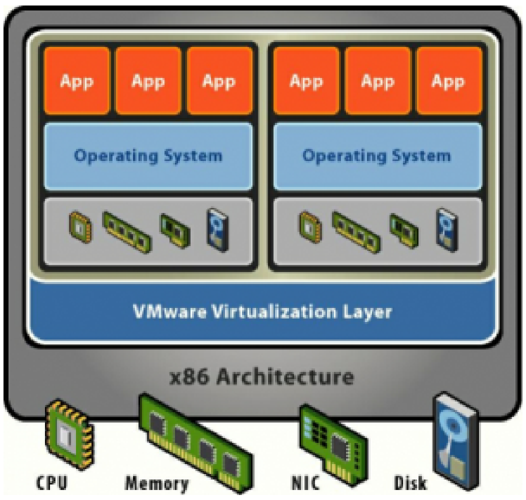


Figure 2.4: Full Hardware Virtualization

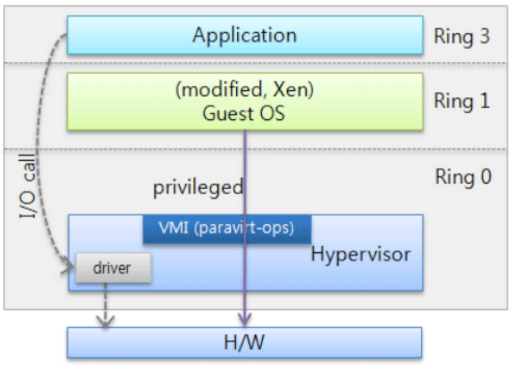


Figure 2.5: Paravirtualization

- *Application Virtualization Family.* This technology creates an additional layer between a specific application and the operating system in order to obtain compatibility and portability for the application. The most common and widespread examples of application virtualization are the *Java Virtual Machine*, present in almost every operating system, and *Wine*, which allows running programs compiled for *Windows* architectures using a *Linux* box;

- *Desktop Virtualization.* It is usually adopted inside terminals and is based on decoupling the logical view of the *Desktop* from the underlying physical host. In this way, each user connecting to the terminal has the illusion of having an entire peronal computer available for exclusive use. Examples of desktop virtualization software are the *Virtual Desktop Infrastructure* (VDI).

2.2.2 Hypervisor types

As already mentioned, a *hypervisor* is a software that allows to create and run virtual machines. As a hypervisor usually provides the administrators means to monitor the status of the VMs it hosts, it is also called Virtual Machine Monitor (VMM). In [45] hypervisors are classified in the following two families:

- *Type I hypervisor* (Figure 2.6). This is also known as *native* or *bare metal* because it directly sits on the hardware. This configuration allows virtual machines to attain best performances, as no intermediate layer is present. The hypervisor has to perform scheduling and resource allocation for all VMs. Examples of this type are *VMWare ESX/ESXI*, *Citrix Xen Server* [2] and *Microsoft Hyper-V*[4].

- *Type II hypervisor*(Figure 2.7). Also known as *hosted hypervisor.* This kind of hypervisor runs as an application inside the hosting OS. In this case, scheduling and resources allocation is performed by the host,

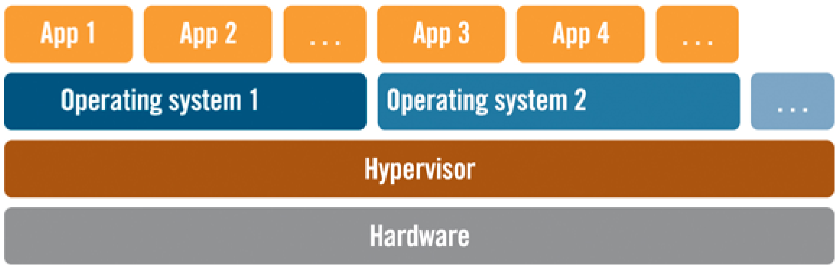


Figure 2.6: Type I hypervisor

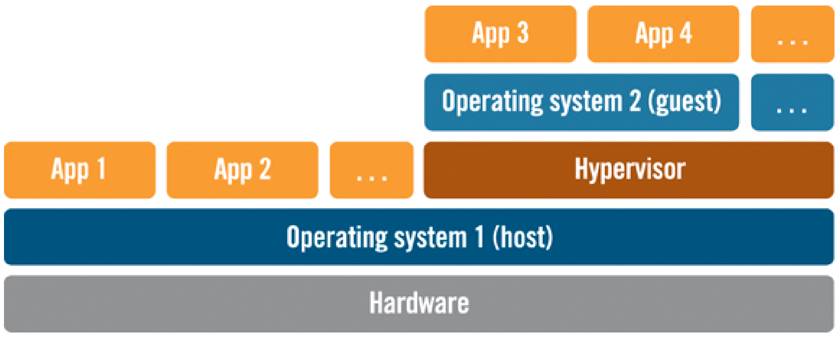


Figure 2.7: Type II hypervisor

making this type of hypervisor less complex compared to type I. An example of this type is the *VMWare Workstation*.

However, the distinction between these two types is not necessarily clear and so a third category can be defined:

- *Hybrid hypervisor*. Also identified as *HVM* (Hybrid Virtual Machine), this type is used when neither type I nor type II are supported by the processor. In this case, privileged instructions are interpreted by a software and they usually require the guest OSes to have specific drivers installed to support this process. For example, Linux's *Kernel-based Virtual Machine* (KVM) and FreeBSD's *bhyve* are kernel modules that effectively convert the host operating system to a type-1 hypervisor.

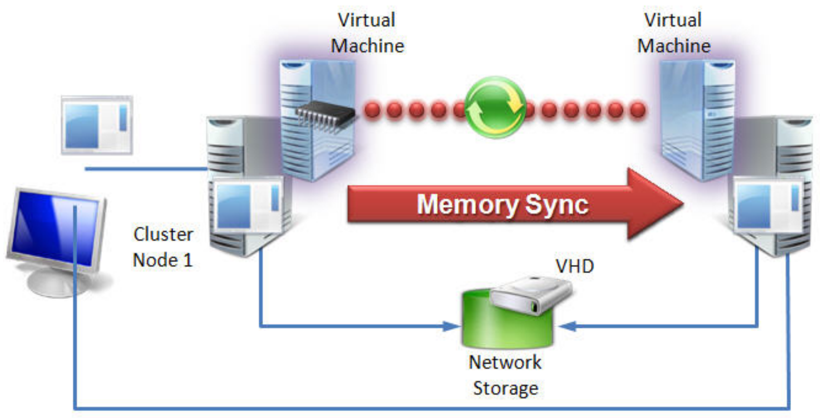


Figure 2.8: Live Migration

2.2.3 Live Migration

As *live migration* [22] is an important mechanism both for our project and for cloud computing in general, we will here describe this feature offered by most of the hypervisors.

Migration(Figure 2.8), also known as *teleport*, consists in moving a virtual machine from an hypervisor to another. If this operation is performed while the guest virtual machine is up and running, the migration can be defined *live*. In this case the process itself becomes very complex, because after migration the guest system status is required to be consistent to its previous state. Most implementations require the system administrator to specify the maximum downtime admitted, namely the amount of time the server can stay down during the process without dramatically affect the quality of service perceived by end users. This process allows to optimize the workload of an infrastructure by moving a virtual machine from a hypervisor to another, if required. For example, it is possible to move a virtual server to a more powerful hypervisor in case the server it is currently running upon is receiving too many requests and the hosting node cannot handle such working load. Another theoretical example is related to the possibility of moving one or more VMs on an hypervisor located in the geographic region from which come most of the requests. This mechanism is also called *WAN migration* or *migration over distance* and for now just a few implementations exist because, as we will explain, in this case some critical issues have to be taken into account.

Migration techniques

Two main migration techniques are available, specifically: *pre-copy* and *post-copy* memory migration. These two techniques differ in the way memory pages are copied from the source to the destination hypervisor.

- Adopting the p*re-copy* memory migration, the source hypervisor starts copying the memory pages of a VM to the destination while the VM keeps working. This phase is called *warming up*. Once the copy if over (all memory pages has been copied) the process checks for any memory pages that were *dirtied* by the guest after the copy, and sends memory differences to the target hypervisor. This is done until differences are not significant enough to allow to shut down the source VM for a specific time needed to finalize the memory copy. Afterwards, the VM is booted on the destination hypervisor. The user can specify a timeout for the copy operation so to have the VM down for a period of time not exceeding the maximum downtime allowed. Note that if the specified value is too low and/or the memory is dirtied at a higher frequency, the process may never converge;
- A *post-copy* memory migration is simpler than the *pre-copy* one but it's more affected by failure events. It simply shuts down the source VM and transfers a minimum subset of the CPU and the memory state to the destination host. After this operation, the VM is immediately booted up at its destination while, concurrently, the process copies the remaining pages from the source to the destination host. If a *yet-to-be copied* page is requested during the process, this event is marked as

page fault and an exception is sent to the source hypervisor, triggering in such a way the immediate transfer of the missing page. Some algorithms are implemented to send pages close to the missing one, as it is very likely that another page fault could occur for those pages. However, if faults happen at a too high frequency, the performance of applications running on the guest VM can be dramatically compromised.

Note that, due to their nature, in case the destination crashed during the process, using the first technique it is still possible to recover the VM we are trying to migrate, whereas in the second case the status of the VM will be permanently lost.

Migration over distance

As we anticipated, we will in this section give some information about this approach, as it is important to understand why we do not actively support this kind of migration in our infrastructure. *Migration over distance*, or WAN migration, consists in migrating a guest to a geographically far hypervisor. This would be an appealing feature since it opens up a window of opportunities in the context of several practical use cases. Nevertheless, most of current orchestration frameworks just limit migration within a datacenter or between hypervisors very close to each other. This fact can be explained by two main considerations:

- The migration process, in case of a pre-copy memory migration, may never converge. We saw how the finalization is done just if memory differences are little enough to grant an acceptable downtime, however due to Internet latency the copy could be processed so slowly that pages will always be dirtied on the source hypervisor. This problem can be solved by increasing the network band or by using a post-copy memory migration technique (but, for the exposed reasons, it could fail as well due to the overhead);

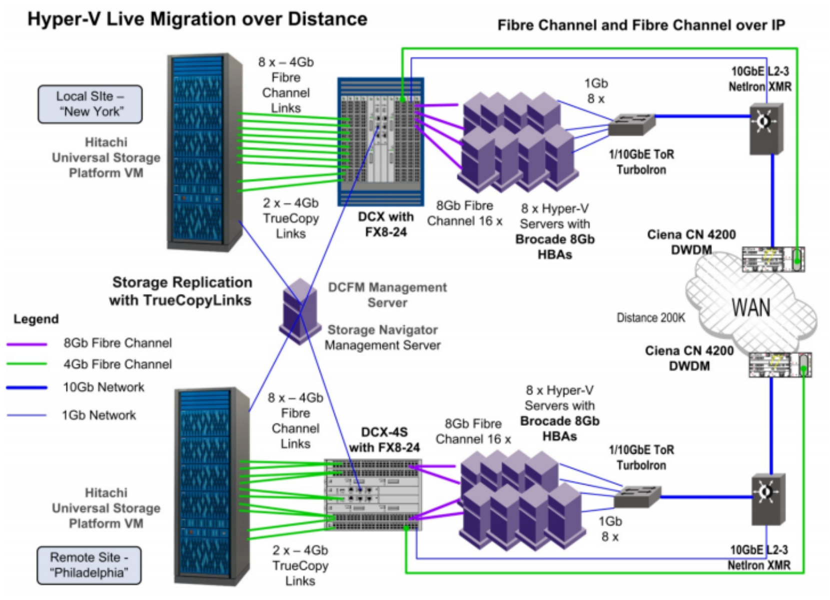

Hyper-V Live Migration over Distance
Fibre Channel and Fibre Channel over IP
Local SIte – "New York"
Hitachi Universal Storage Platform VM
8 x – 4Gb Fibre Channel Links
2 x – 4Gb TrueCopy Links
DCX with FX8-24
8Gb Fibre Channel 16 x
8 x Hyper-V Servers with Brocade 8Gb HBAs
1Gb 8 x
1/10GbE ToR TurboIron
10GbE L2-3 NetIron XMR
Ciena CN 4200 DWDM
WAN
Distance 200K
Storage Replication with TrueCopyLinks
DCFM Management Server
Storage Navigator Management Server
Legend
8Gb Fibre Channel
4Gb Fibre Channel
10Gb Network
1Gb Network
DCX-4S with FX8-24
Remote Site - "Philadelphia"

Figure 2.9: Hyper-V long distance migration

- I/O operations from/to storage devices connected through the Internet are not efficient enough. Dealing migration in fact, we just focused our attention on system memory copy and we didn't consider the copy of persistent memory. The reason is that a mass storage can consist of several terabytes of data and so, being the copy not an option, it is usually shared among hypervisors by adopting a so-called *disk sharing* technique.

Furthermore, this kind of migration is actually being studied by researches and companies which see in it a challenge and/or a way to make profits. An attempt of long distance migration has been done by Microsoft with its *Hyper-V* technology [63]. Specifically, a migration between 200 Km distant sites has been done, involving New York and Philadelphia data centers, but Figure 2.9 shows how many components have been involved and let us understand how this process can be complex.

In fact, to allow this kind of migration a consistent copy of storages is maintained through periodic updates, so that the migrated VM will not end up using a far distant hard disk cluster, but a local copy of it. Second, dedicated high-speed connections are required to keep disk consistence and to allow the migration process itself. More detailed information about long distance migration can be found in [4].

2.2.4 Available technologies

Virtualization technologies

In this section we provide the reader with an overview of the most widespread and powerful virtualization technologies pointing our attention to the specific solutions we decided to rely upon for the implementation of our architecture. We can distinguish two big families of product:

- Open source solutions
  - *KVM*[3]. It stands for Kernel Based Virtual Machine and it is a solution implemented for Linux systems on x86 platforms. It requires specific processor extensions to work, like Intel VT-x or AMD-V, and it is implemented as a kernel module. Starting from Linux kernel version 2.6.20, it is officially included in almost every distribution. In order to make an user program communicate with the hypervisor, it supplies an interface /dev/kvm;
  - *Qemu*[8]. It is often used with KVM, as the user space program interacting with the hypervisor. Qemu, by itself, is just an emulator, but when used together with KVM it becomes a virtualization tool. Thanks to Qemu, it is possible to create virtual networks or virtual devices to use with virtualized guests
  - *Xen*[2]. Xen is an example of virtualization tool that makes use of paravirtualization. It is sometimes considered as a bare metal hypervisor, as it provides its own layer sitting directly on the hardware, and allows the execution of modified guests. It also can run unmodified operating systems by making use of hardware extensions, when present;

  - *VirtualBox*[7]. It is structurally very similar to KVM, because it is implemented as a kernel module as well. However, it also provides its own user space software and, unlike KVM, it is available for Windows and OSX too, being the most portable open source solution. Moreover, it allows the user to pause/resume a guest at will and to take snapshots, so to provide a form of backup. Similar to Qemu, it allows emulation of hardware and network devices.

- Proprietary solutions

  - *VMWare* [12]. It is one of the most active virtualization software companies. It provides both type I and type II hypervisors, respectively *VMWare ESXi Server* and *VMWare workstation.* It makes use of a technique called binary translation. Using this technique, a guest system directly sits on the hardware, and when privileged instructions are called they are redirected to the hypervisor and emulated. This approach had a lot of success when introduced because, at that time, no processor extensions were present to support full hardware virtualization;
  - *Microsoft* [4]. Microsoft focused on the type I approach with its *Hyper-V* technology. *Hyper-V* can come with *Windows Server* distributions or as a standalone product. In order to support isolation, it requires to distribute guests in isolated partitions of the host environment and that at least one guest has a running instance of Windows Server, in order to provide user with management capabilities;
  - *Apple* [1]. Like VMWare, it provides two types of hypervisors under the form of *Parallels Server* and *Parallels Workstation* products. Being a proprietary technology, very limited details are available about its implementation. Anyway, their distinctive trait is the full virtualization of the *Graphics Processing Unit*(GPU).

Disk Sharing technologies

As we said, the migration process requires source and destination hypervisors could share the same external storage device. For this reason, it is important to describe the most widely adopted techniques in this field. We can distinguish between two main approaches: *Network Attached Storage*(NAS) and *Storage Area Network*(SAN).

The main feature of Network Attached Storage technology (whose architecture is summarized in Figure 2.10) is that it is a file-level data storage, providing access to its files by means of network file sharing protocols such as *Network File System (NFS)* [37], *Server Message Block (SMB)*, *Apple Filing Protocol(AFP)*, etc. It is usually implemented as a server with its own operating system (typically a Linux distribution) and it provides an abstraction of physical disk storages through the aforementioned file sharing protocols. This kind of storages mechanism can sometimes be organized by some sort of redundancy configuration such as *Redundant Array of Independent Disks* (RAID) that combines multiple disk drive components into a logical unit in order to improve the overall performance of the system.

Since a NAS system provides by its own nature an abstraction of the file system, no additional effort is required to the client operating system, whose only requirement is to contact the device and communicate by means of an appropriate network protocol.

Common NAS implementations, however, provide multiple file sharing protocols so to improve their accessibility performance. The weakness of this technology is just that, providing a file system abstraction, it could be not efficient enough for some classes of applications. For this reason, this technology is mainly adopted in environment not affected by delay and latency, for example in *Local Area Networks.*

Storage Area Network main feature consists in the ability to provide a low-level block device interface, with no file abstraction. As the name itself suggests, it is a dedicated network, although physical storages are not directly accessible. About this aspect, while NAS makes use of a server in order to

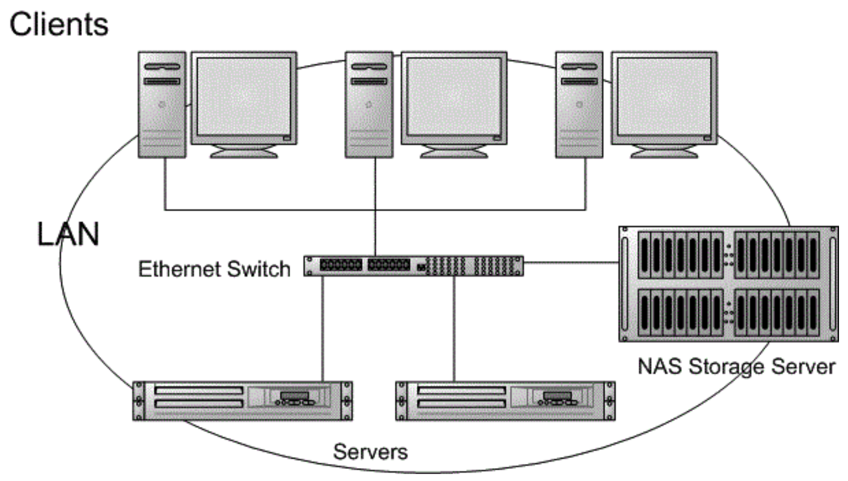


Figure 2.10: Example of NAS infrastructure

share data with clients, SAN can be either a server or a dedicated piece of hardware. This is because the resource sharing is realized at block level and so there could be no need for complex applications handling data. As a drawback, for the same reason, client operating systems are required to implement their own file system management mechanism.

Fiber Channel and *ISCSI* are two well known disk sharing techniques making use of the SAN paradigm:

- *Fiber channel* is a high-speed network technology that can reach up to 16 Gigabit per second rates. Its name is due to the fact that its first implementations ran on optical fiber. Nowadays, it can run on an electrical interface as well. Fiber channel assures the best performances, although it is very expensive because of the dedicated hardware it requires;

- *ISCSI* (Internet Small Computer System Interface), on the other hand, directly runs on IP with no need for dedicated hardware and, for this reason, has become more and more popular.

Interested readers can find further details in [10].

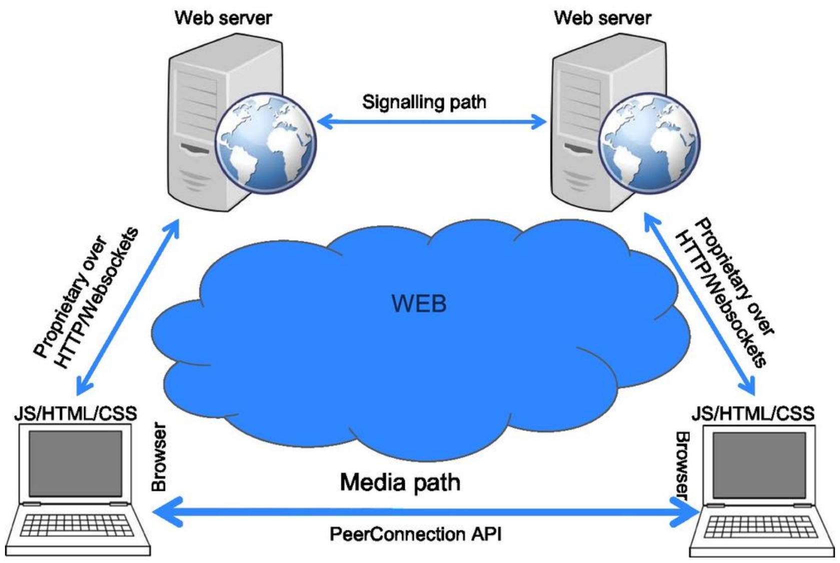


Figure 2.11: The WebRTC Trapezoid

2.3 Web Real-Time Communication (WebRTC) and Real-Time Communication in WEB-browsers (RTCWeb)

WebRTC/RTCWeb is a joint standardization effort conducted by the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF). This brand new technology supports *browser-to-browser* applications for voice calling, video chat, and P2P file sharing without the need of either internal or external plugins. The standardization goal is to define a standard API that enables a web application running on any device, through the secure access to the input peripherals (such as webcams and microphones), to send and receive real-time media and data in a peer-to-peer fashion between the browsers. In fact WebRTC extends the common client-server approach adopted by the web application and introduces a peer-to-peer communication paradigm between the browsers. The most general WebRTC architectural model (see Figure 2.11) is based on the SIP Trapezoid [51].

In the this model both browsers are running a WebRTC application, downloaded from a different Web Server. A Peer Connection configures the

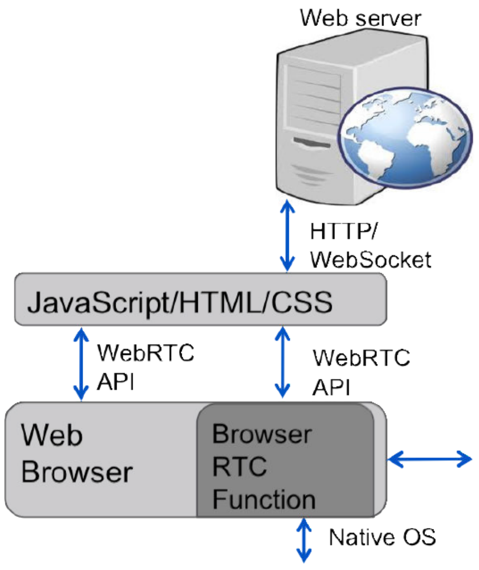


Figure 2.12: A WebRTC application in the browser

media path to flow directly between browsers without any intervening servers. Signaling goes over HTTP or WebSockets, via Web Servers that can modify, translate or manage signals as needed. It is worth noting that the signaling between browser and server is not standardized in WebRTC as it is considered to be part of the application. The two web servers can communicate using a standard signaling protocol such as Session Initiation Protocol (SIP) or Jingle [33]. Otherwise, they can use a proprietary signaling protocol. A WebRTC application is a web application typically written in a mix of HTML and JavaScript (Figure 2.12).

The application interacts with web browsers by means of the standard WebRTC API which allows to properly exploit and control the Real-Time browser functions. The *WebRTC API* also provides a wide set of functions, like connection establishment and management, media negotiation and control, firewall and NAT (Network Address Translation) [61] traversal.

The API is being designed around the following three main concepts:

- *PeerConnection*, it represents an association with a remote peer in the context of a *browser-to-browser* communication. Usually the remote side of a *PeerConnection* is another instance of the same JavaScript application.

Communications are set up and manged via a signaling channel provided by scripting code in the page via the web server. Once a peer connection is established, media streams (locally associated with ad hoc defined MediaStream objects) can be sent directly to the remote Browser. The peer-connection mechanism uses the Interactive Connectivity Establishment (ICE) [49] protocol together with the *Session Traversal Utilities for NAT* (STUN) [50] and *Traversal Using Relays around NAT* (TURN) [35] servers to let UDP-based media streams traverse NAT elements and firewalls. Using ICE provides also a security measure, as it prevents untrusted web pages and applications from sending data to hosts that are not expecting to receive them;

- *MediaStream* is an abstraction of an actual stream of audio/video data. It allows to actually manage and act on the stream, namely displaying the stream's content, recording it, or sending it to a remote peer. A MediaStream object may be used to represent a stream that either comes from (remote stream) or is sent to (local stream) a remote node. A *LocalMediaStream* represents a media stream from a local media-capture device (e.g., webcam, microphone etc.). To create and use a local stream the web application must request access from user through the `getUserMedia()` function. The application specifies the type of media (audio and/or video) to which it requires access. The browser interface presents the user the option to grant or deny access to the corresponding devices. In any case, once the application is done, the browser has to release the resources. Media-plane signaling is achieved by means of out-of-band mechanisms between the peers of the communication. *Secure Real-time Transport Protocol* (SRTP) is used to carry the media data while the *RTP Control Protocol* (RTCP) information are used to monitor transmission statistics associated with data streams. *Datagram Transport Layer Security* (DTLS) is used for SRTP

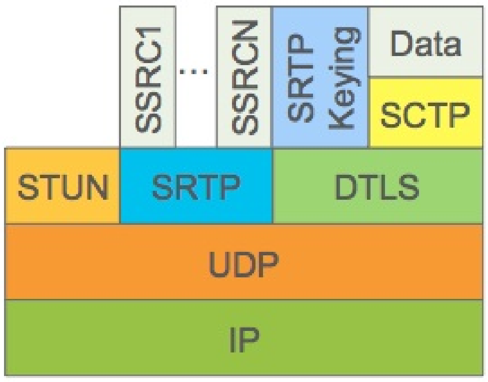


Figure 2.13: The WebRTC protocol stack

key and association management;

- The *DataChannel* is a communication channel designed to provide a generic transport service allowing web browsers to exchange generic data (not limited to audio and video) in a bidirectional peer-to-peer fashion. The standardization work within the IETF has reached a general consensus on the usage of the *Stream Control Transmission Protocol* (SCTP) [59] encapsulated in DTLS to handle non-media data types.

Figure 2.12 shows the WebRTC protocol stack.

2.3.1 Real-time Transport Protocol (RTP)

The Real-time Transport Protocol (RTP), provides end-to-end delivery services for data with real-time characteristics, such as interactive audio and video. Those services include payload type identification, sequence numbering, timestamping and delivery monitoring. Applications typically run RTP on top of the *User Datagram Protocol* (UDP) to make use of its multiplexing and checksum services. Both protocols contribute parts of the transport protocol functionality. However, RTP may be used with other suitable underlying network or transport protocols. RTP supports data transfer to mul-

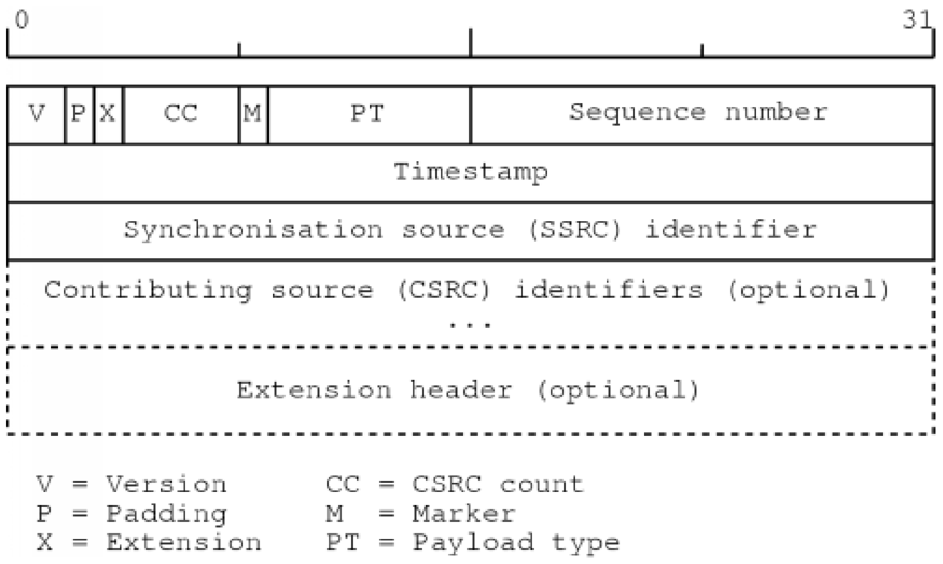

0
31
V
P
X
CC
M
PT
Sequence number
Timestamp
Synchronisation source (SSRC) identifier
Contributing source (CSRC) identifiers (optional)
...
Extension header (optional)
V = Version
P = Padding
X = Extension
CC = CSRC count
M = Marker
PT = Payload type

Figure 2.14: Real-time Transport Protocol header

tiple destinations using multicast distribution if provided by the underlying network.

The protocol, whose header is depicted in Figure 2.14, provides facilities for jitter compensation and detection of out of sequence arrival in data, which are common during transmissions on an IP network. Note that RTP itself does not provide any mechanism to ensure timely delivery or provide other quality-of-service guarantees, but relies on lower-layer services to achieve this task. It does not guarantee delivery or prevent out-of-order delivery, nor does it assume that the underlying network is reliable and delivers packets in sequence. Nevertheless, the sequence numbers included in RTP allow the receiver to reconstruct the sender's packet sequence. Furthermore, sequence numbers might also be used to determine the proper location of a packet, for example in video decoding, without necessarily decoding packets in sequence.

The *Secure Real-time Transport Protocol* (SRTP) [18] defines a profile of RTP intended to provide encryption, message authentication and integrity, and replay protection to the RTP data in both unicast and multicast applications.

2.3.2 Real-time Transport Control Protocol (RTCP)

The RTP specification also describes the *RTP Control Protocol* (RTCP) which provides *out-of-band* statistics and control information for an RTP session. It works in association to RTP for delivering and packaging of multimedia data, but does not transport any media data itself. In fact, the only function of RTCP is to provide feedback on the *Quality of Service* (QoS) in media distribution by periodically sending statistics information to participants in a streaming multimedia session.

RTCP provides basic functions expected to be implemented in all RTP sessions:

- The primary function of the protocol is to gather and trasmit to the other peer of the communication statistics on quality aspects of the media delivery. Such information may be used by the source for adapting the flow rate or even changing the media encoding in order to prevent congestions and faults;

- *RTCP* also provides canonical end-point identifiers (CNAME) to all session participants. This is a key fnction because, although a source identifier (SSRC) of an RTP stream is unique, the instantaneous binding of source identifiers to end-points may change during a session. Using the *CNAME* the application can univocally identify the end-points of the sessions;

- Provisioning of session control functions. RTCP is a convenient means to reach all session participants, whereas RTP itself is not. RTP is only transmitted by a media source.

The protocol natively envisages several types of messages: *sender report*, *receiver report*, *source description* and *goodbye*. In addition, the protocol can be extended thus allowing application-specific RTCP packets.

- The *Sender Report* (SR) message is sent periodically by the active peer in an communications session and it reports transmission and recep-

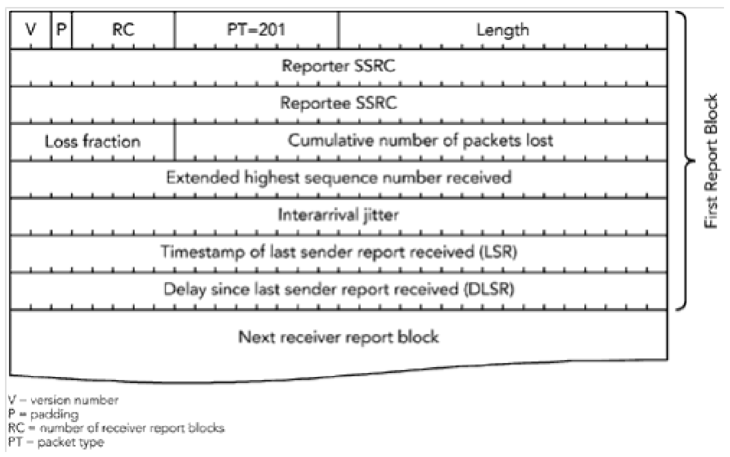

V
P
RC
PT=201
Length
Reporter SSRC
Reportee SSRC
Loss fraction
Cumulative number of packets lost
Extended highest sequence number received
Interarrival jitter
Timestamp of last sender report received (LSR)
Delay since last sender report received (DLSR)
Next receiver report block
First Report Block
V – version number
P = padding
RC = number of receiver report blocks
PT – packet type

Figure 2.15: Example of Receiver Report message

tion statistics for all RTP packets sent during that period of time. The sender report includes an absolute *timestamp* that allows the receiver to synchronize RTP messages. This information is particularly important when both audio and video are transmitted simultaneously, because audio and video streams use independent relative timestamps and thanks to the RTCP timestamp they can be synchronized from the receiver;

- The *Receiver Report* (RR) is sent by the passive peer of the session and informs the other parties of the communication session its perceived quality of service;

- *Source DEScription* (SDES) is the message used for sending the aforementioned CNAME information to the other parties of the sessions. This message can be used to send user-level information (e.g. e-mail address or phone number of the user sending the media flow);

- The *Goodbye* (BYE) message is used to announce that a peer is leaving the session.

Figure 2.15 depicts an example of RR message.

2.4 Scalable Video Coding

In this section we will briefly introduce the *Scalable Video Coding* [57] process since it can very helpful to the realization of our application logic, specifically such a kind of technique could represent a standard and easy way for creating the different copies of the original video stream to be delivered to end users in order to cope with their heterogeneous capabilities and network conditions (see Section 1.2.2). A *Scalable Video Coding* technique allows indeed the encoding of a high-quality video bitstream by creating two or more subset bitstreams that can themselves be decoded with a complexity and reconstruction quality similar to that achieved using the existing standard codecs (i.e. H.264 [65] or VP8 [17]). The subset bitstreams can be obtained by the original one by dropping packets. From a practical standpoint, the coded flow can be seen as a basic bitstream which allow to decode a the video with a minimum quality and a set of further layers used to add quality and enrich the amount of data the decoder can use to reconstruct the video and present it to the final user. Multiple techniques can be used, individually or in combination, for deriving different scaled version of the same video flow. They are based on the spatial and temporal redundancy properties of a video flow and they lead to the following modalities:

- —*Spatial scalability.* In this scenario video is coded at multiple spatial resolutions, i.e. different picture sizes. In the decoding process of the high resolution images can be used data contained in the low resolutions ones;

- *Temporal scalability.* This technique allows to dynamically change the number of frames sent per second (*frame rate*). The motion compensation dependencies are structured so that complete pictures (i.e. their associated packets) can be dropped from the bitstream without compromise the decoding process;

- *SNR/Quality scalability.* In this case the video is coded using a single spatial resolution but multiple quality degrees. Again, data and de-

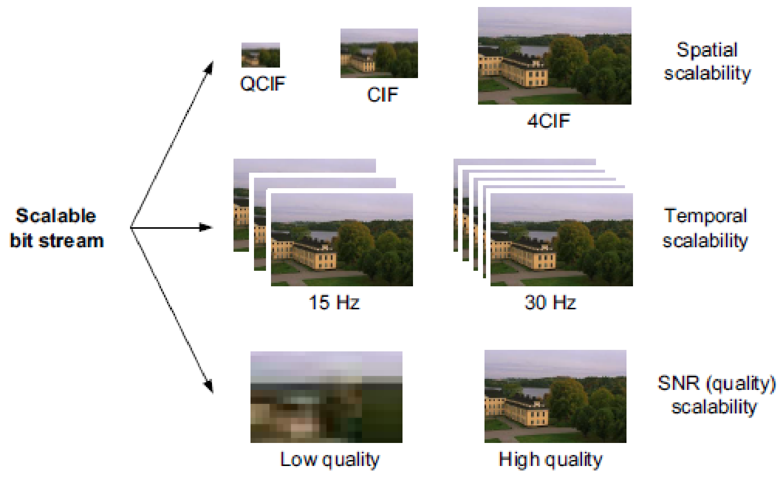


Figure 2.16: Representation of different types of scalability

coded samples of lower qualities can be used to predict data or samples of higher qualities.

The aforementioned concept are exemplified in Figure 2.16

2.5 The Gstreamer framework

In this section we will briefly introduce *GStreamer* [64], the foundation we used to implement the high level streaming application chosen as the main use case of this work. GStreamer is a framework for creating streaming media applications, its fundamental core element is based on the video pipeline abstraction presented at *Oregon Graduate Institute*[29]. In this context a pipeline consists of a chain of processing elements arranged so that the output of each element is the input of the next one. Such a kind of architecture makes it possible to write any type of streaming multimedia applications and to handle audio or video or both. Nevertheless, the framework allow to process any kind of data flow. The pipeline design is made to have little overhead above what the applied filters induce. This makes GStreamer a good framework for designing our audio/video streaming application which puts high demands on latency. The tool is based on a core function which

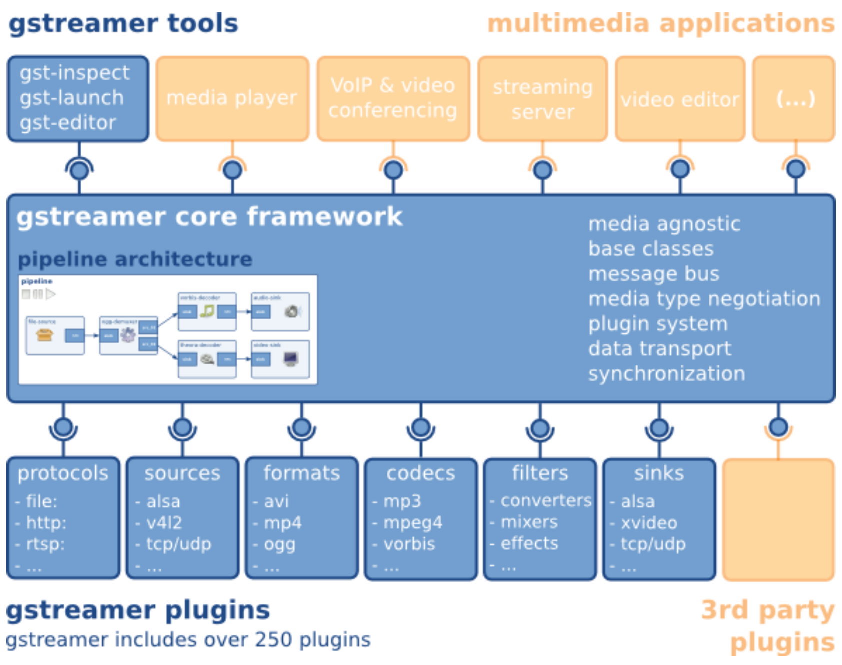


Figure 2.17: GStreamer architecture overview

can be enriched by means of plugins. The core provides data flow and media type handling/negotiation functions and an API to write applications using the various provided plugins.

Figure 2.17 shows an overview of the GStreamer architecture. As we can see the framework provides:

- A core providing a pipeline architecture;
- An API for multimedia applications;
- Entry points for both 3rd party and built-in plugins;
- Mechanisms for media type handling and negotiation;
- A set of tools to easily create and manage an application.

2.5.1 GStreamer foundations

The GStreamer multimedia framework is built around the following basic concepts:

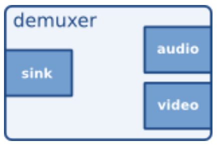


Figure 2.18: Source element with multiple output *pads*

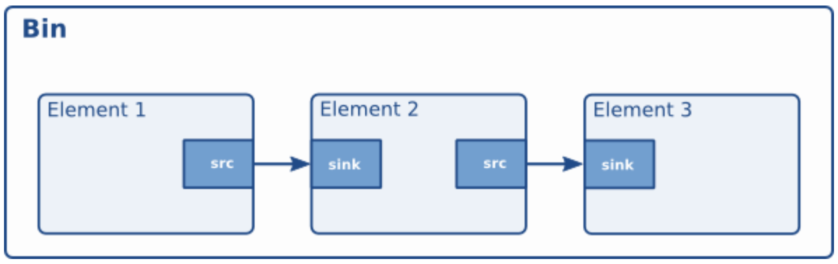


Figure 2.19: Bin of elements

- *Element.* It is the most important class of objects in GStreamer. It'is in fact possible to create a chain of elements linked together and let data flow in the proper way through this chain. An element is in charge of just one the following specific function (i) reading of data from a file (ii) decoding of this data (iii) outputting this data to a sound card. A pipeline can be created by chaining together several such elements. A graphical representation of a GStreamer element is depicted in Figure 2.18;

- *Pads* are element's input and output points used to connect them to other elements. From an high level perspective, a pad can be viewed as a 'plug' or 'port' on an element where links can be established with other elements, and through which data can flow to or from those elements. Pads have specific data handling capabilities: a pad can restrict the type of data that flows through it. Links are only allowed between two pads when the allowed data types of the two pads are compatible. Data types are negotiated between pads using a process called caps negotiation. Data types are described as a *GstCaps*. Figure 2.18 shows an example of element with one input end two output *pads*;

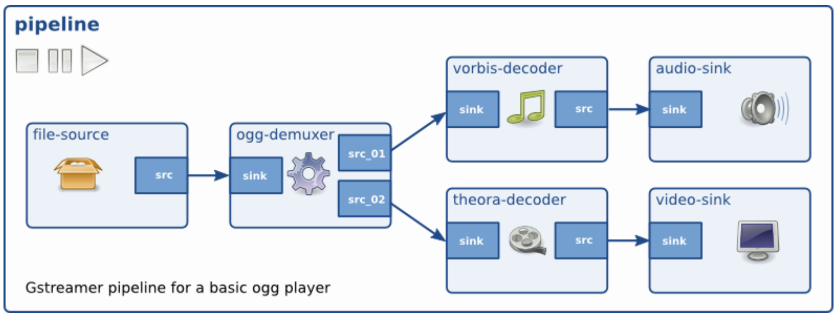


Figure 2.20: Example of pipeline for a basic audio player

- A *bin* is a container for a set of *elements*. Since bins are subclasses of elements themselves, a *bin* can mostly be controlled as if it were an element, in such way decreasing the overall application complexity. It is possible, for instance, to change state to all elements in a bin by acting on the state of the bin itself. A bin is represented as in Figure 2.19;
- A *pipeline* is a top-level bin. It provides a bus for the application and manages the synchronization for its children. Setting the `PLAYING` state on the pipeline triggers the starting of data flow and media processing. Once started, pipelines will run in a separate thread until they are stopped or the end of the data stream is reached. A pipeline is composed by a chain of elements and can be represented as in Figure 2.20

2.5.2 Communication and events

The framework provides several mechanisms for communication and data exchange between the application and the pipeline. With reference to Figure 2.21 we can distinguish:

- *Buffer* objects, represented by the blue arrows in the figure, used to pass streaming data between elements in the pipeline ;
- An *event* object (red arrows) instead used for sending generic information between elements and from the application to elements;

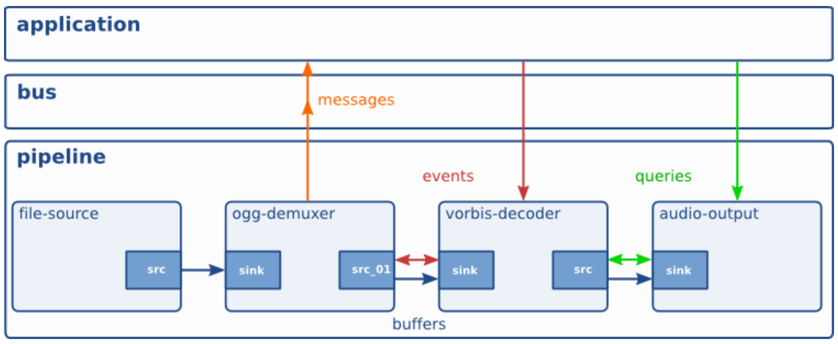


Figure 2.21: GStreamer components' communication mechanism

- *Messages* (orange arrows) which are objects posted by elements on the pipeline's message bus, where they will be held for be collected by the application and they usually are used to transmit information such as errors, tags, state changes, buffering state, redirects etc. from elements to the application in a thread-safe way;
- *Queries* (green arrows) allowing applications to request information to the pipeline in a synchronous way. Elements can also use queries to request information from their peer elements.

Basing on the lesson learned by the architectural solutions analyzed and presented in the first part of this section and taking advantage of the benefits achievable by exploiting the tools and technologies introduced in the second part, we started to cope with the realization of our architecture. As a first step we devoted our efforts to the realization of the different components needed to realize a distributed and dynamically configurable streaming application able to provide users with media flows with an extremely low transmission delay. The design and implementation of such components will be presented in the next session.

Chapter 3

Application level main components

As it has been detailed in Section 2.5, we decided to adopt live streaming as the main use case of our research activity and we opted for building a prototype application on top of the Gstreamer framework. In this chapter we will present the design and implementation of the various bricks composing the streaming delivery chain and, in order to let the reader better understand how these elements have been deployed and executed on the SOLEIL infrastructure, we will point out the mapping of each application level component on the underlying tree topology elements. Since the system has to allow proper mechanisms to let the various nodes can join, leave and send messages to each other we also had to take care of this aspects in every element. For this purpose we designed and implemented an ad hoc protocol in charge of both management and monitoring functions. It will be the object of the first section of this chapter.

3.1 SOLEIL management and monitoring protocol

As we anticipated in Section 1.2, in order to add to the system automatic monitoring and management functionalities an *ad hoc* protocol is needed. In fact, since in our scenario several clients access the service by means of the same 'system access point'(i.e. one of the *leaf* node) and each client connection is characterized by its own specific network condition, it is not possible to simply rely on the native QoS monitoring and statistic functionalities provided by the *RTCP* protocol in order to adjust the transmission rate. Furthermore, we need specific tools to manage the streaming mechanism and the reconfiguration of the system topology. The SOLEIL Protocol is based on the *eXtensible Markup Language* (XML) [21] hence it can be easily extended and encapsulated into *XML-compliant* protocols (e.g. HTTP, XMPP, etc.). A generic protocol message has the following general structure:

```
<message type="MESSAGE-NAME" time="TIMESTAMP" to="
   DESTINATION-NODE"
      from="SOURCE-NODE">
      <...>
            MESSAGE BODY
      </...>
</message>
```

The message attributes define the following properties of the message:

- `type` defines the format of the XML message body;
- `time` contains a *timestamp* information describing the instant in time in which the message has been sent, defined as the number of seconds that have elapsed since the *Epoch time* ($00:00:00$ Coordinated Universal Time (UTC), Thursday, 1 January 1970);
- `to` and `from` contain respectively the *id* of the destination and of the source of the message. Some special characters are reserved and used to identify the following entities of the system:

  ^ is used to identify the root node and can be a value both, for the `from` and `to` attributes, not simultaneously of course;

  * is used to identify all nodes in the network and can be used as a value for the attribute `to` in order to send broadcast messages;

Once a message is receive by one of the nodes its `to` field is checked in order to understand how the message has to be handled, according to the flow diagram depicted in Figure 3.1.

3.1.1 Types of message

The protocol envisages the exchange of different types of messages. We can group these messages by their high level function:

- Nodes management: `HELLO, BYE, CHILDSUPDATE, ERROR, KILL, PING, PONG, REDIRECT`;
- Stream management: `PIPEDEF, PIPEDEFINFO, PLAYPIPELINE, STOPPIPELINE, REMOVEPIPELINE, ROUTEPIPELINE`;
- System monitoring and statistic aggregation: `STATS, REQFULLSTATS, FULLSTATS`.

For example, is a new node enter the network, it can presenting itself to the other nodes of the network by sending to the ROOT node an `HELLO` message containing the IP addresses and the ports of its server and client components:

```
<message type="hello" to="^" time="1367231998886">
      <info>
            <name>gstreamer_Notebook-PC_14982</name>
                  <minaServerAddress host="
                     143.225.229.199" port="14982"/>
                  <minaClientAddress host="
                     143.225.229.134" port="12054"/>
                  <pipelines/>
      </info>
</message>
```

3.1.2 Statistic aggregation algorithm

As we said, the system has to be able to collect *RTCP* statistic and monitoring information and to aggregate them in order to properly deal with specific

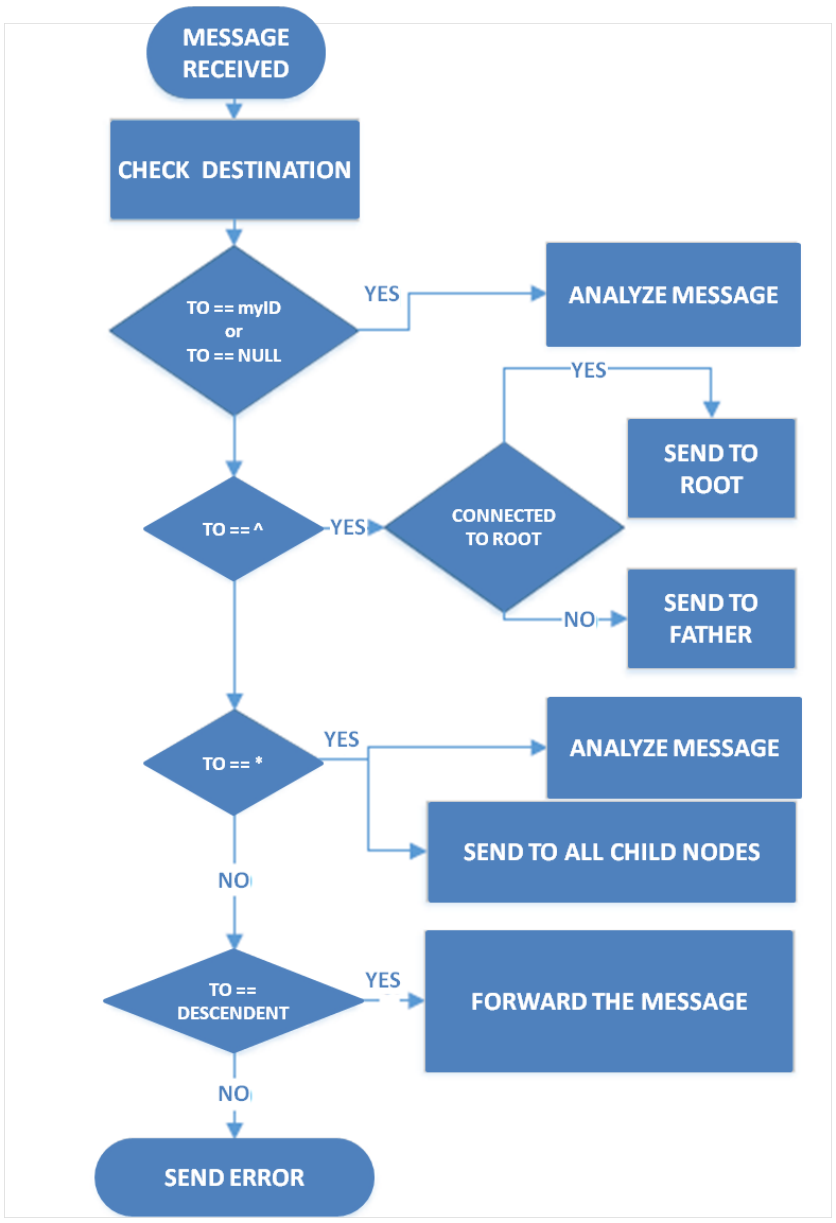


Figure 3.1: SOLEIL Protocol: messages handling flow diagram

constraints imposed by the overall architecture (see Section 1.2. This kind of information allows to properly monitor the overall status of the system and to react to critic load conditions by either temporarily scaling the quality of the delivered stream or migrating an overloaded node of the tree in order to increase its power (both in terms of computational resources and bandwidth availability). For this reason, we need to provide the entity in charge of monitoring the system both information about the overall status of the entire topology and tools to query specific nodes about their current load and resource consumption status in a proactive way.

This goal is obtained by means of explicit messages provided by the *SOLEIL Protocol*. Specifically, the aforementioned protocol messages have the following meanings and structures:

- A `STATS` message is sent by each node to its direct predecessor and contains information about the status of the different streams handled by the sender node. The message can in fact contain one or more children, namely the `statsContainer` tag, containing the following statistical information:
  - `startTime` is a temporal information indicating the end of the interval of time during which the statistic information have been collected;
  - `endTime` is a temporal information indicating the end of the interval of time during which the statistic information have been collected;
  - `numOfChildren` is the number of children of the sender node;
  - `id` contains an identification of the stream;
  - `statArray` contains an array of statistic information. In our current implementation of the protocol the conveyed information are *CPU load*, *inter-arrival jitter* and *cumulative packet lost*. Obviously more detailed information can be added in the future by extending the protocol

An example of STATS message is reported in the following:

```
<messagetype="stats" time="1368548879868">
      <statsContainers>
            <statsContainer>
                  <startTime>1368548254</startTime
                     >
                  <endTime>1368548879</endTime>
                  <numOfChildren>3</numOfChildren>
                  <id>1234567</id>
                  <statArray>
                  <DBValue>
                        <name>injitter</name>
                        <value>255.3</value>
                  </DBValue>
                  <DBValue>
                        <name>cumpktlost</name>
                        <value>12</value>
                  </DBValue>
                  </statArray>
            </statsContainer>
            <statsContainer>
                  <startTime>1368548254</startTime
                     >
                  <endTime>1368548879</endTime>
                  <numOfChildren>3</numOfChildren>
                  <id>1234568</id>
                  <statArray>
                  <DBValue>
                        <name>injitter</name>
                        <value>240.2</value>
                  </DBValue>
                  <DBValue>
                        <name>cumpktlost</name>
                        <value>8</value>
                  </DBValue>
                  </statArray>
            </statsContainer>
      </statsContainers>
</message>
```

- a `REQFULLSTATS` message is used by the ROOT node to ask a node to send information about the streams it is handling. More in detail, the ROOT node can specify a `pipeid` field to identify the stream it is interested in. The `to` and `from` attributes allow to properly address the message to the right node of the architecture. The `time` field allows to trace and sort multiple requests received by the ROOT node. An example of `REQFULLSTATS` message is reported in the following:

```
<message to="gstreamer_U36SD_52846" from="^" type=
   "reqfullstats" time="1368548950349">
      <pipeid>1234567</pipeid>
</message>
```

- a `FULLSTATS` message can then be used by the destination node to reply to the `REQFULLSTATS` request. This message is addressed by means of the `to` and `from` attributes. The `time` field has to be set using the `time` value read from the request message, thus allowing the ROOT node to properly associate requests and responses. The body of the message is formatted according to the aforementioned `statsContainer` syntax. An example of `FULLSTATS` message is reported in the following:

```
<message to="^" from="gstreamer_U36SD_52846" type=
   "fullstats" time="1368548950391">
      <xmlDb>
                  <statsContainer>
                  <startTime>1368548254</startTime
                     >
                  <endTime>1368548879</endTime>
                  <numOfChildren>3</numOfChildren>
                  <id>1234568</id>
                  <statArray>
                  <DBValue>
                        <name>injitter</name>
                        <value>240.2</value>
                  </DBValue>
                  <DBValue>
                        <name>cumpktlost</name>
```

```
                                <value>8</value>
                        </DBValue>
                        </statArray>
                </statsContainer>
        </xmlDb>
        <endTime>1368548238</endTime>
</message>
```

In Figure 3.2 is schematized the typical interaction among the main components of the network. As we can see, a generic *Leaf* node elaborates and sends its statistic information to its direct predecessor (a *Relay* node). This node, in turn, creates its own aggregate statistic information according to an algorithm that will be detailed in the following, and sends it to the ROOT node. At this point, the ROOT node, figuring out an overload condition can afflict the *Leaf* node, can decide to analyze more in detail the potentially critic situation by requesting the actual statistic to that node. The *Leaf* current statistic information are sent by means of a `FULLSTATS` message.

The aggregation of statistic information is performed by each internal node of the network in order to provide the ROOT node with an overview of the overall status of the system. Each node receives the statistic information from its direct *child* nodes by means of a *RTCP ReceiverReport* message. The node hence calculates the statistic to send to its direct predecessor in the tree topology by assigning to the received statistics a weight depending to the number of *heirs* of each sender node. Figure 3.3 can help to better understand this process. As we can see, the node A has 3 direct child nodes and 6 total descendants. So, the nodes will assign weight 4 to the statistic information received by the node B and weight 1 to the statistic information received by both the node C and the node D.

The node will then compute its own global statistic by using the following formula:

$$s(x_i) = \frac{\sum_{n=1}^{N}(s(x_n)*p(x_n))}{\sum_{n=1}^{N} p(x_n)}$$

where $p(x_n) =$ *weight* associated to *statistic* $x_n =$ number of descendants.

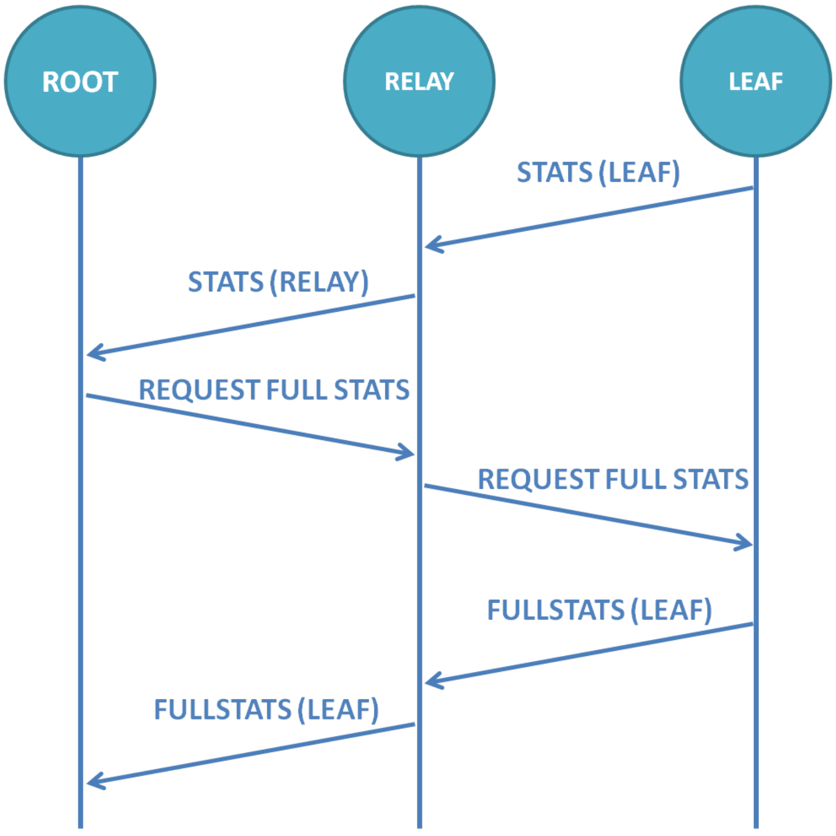


Figure 3.2: SOLEIL protocol: typical messages exchange path

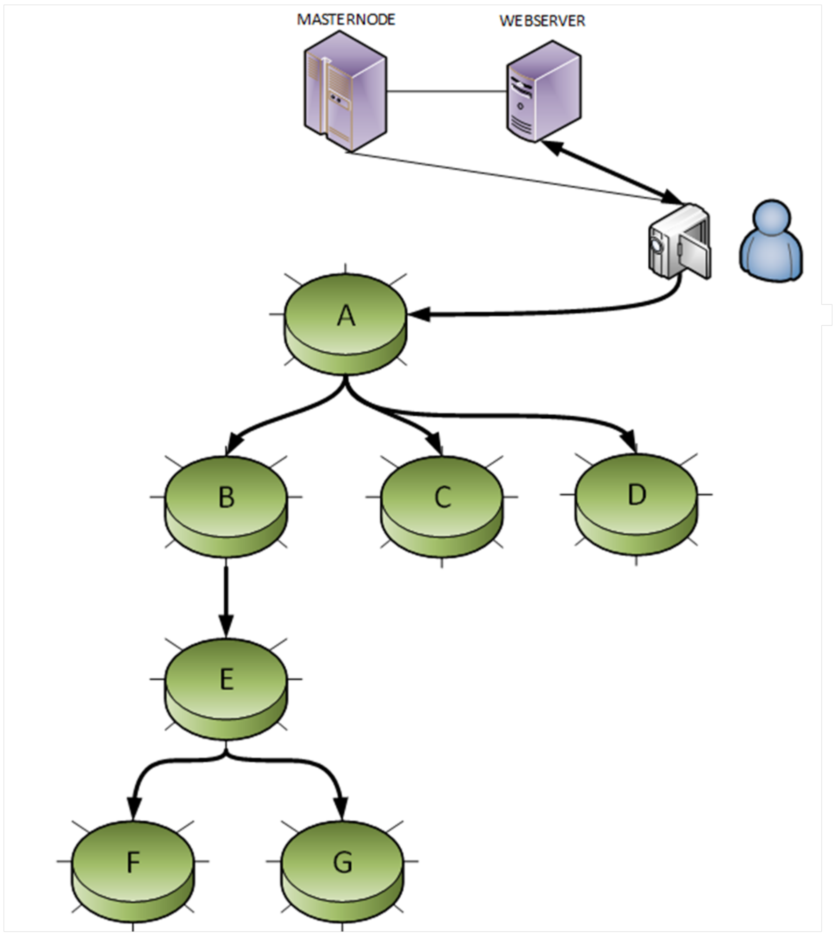


Figure 3.3: Example of SOLEIL topology

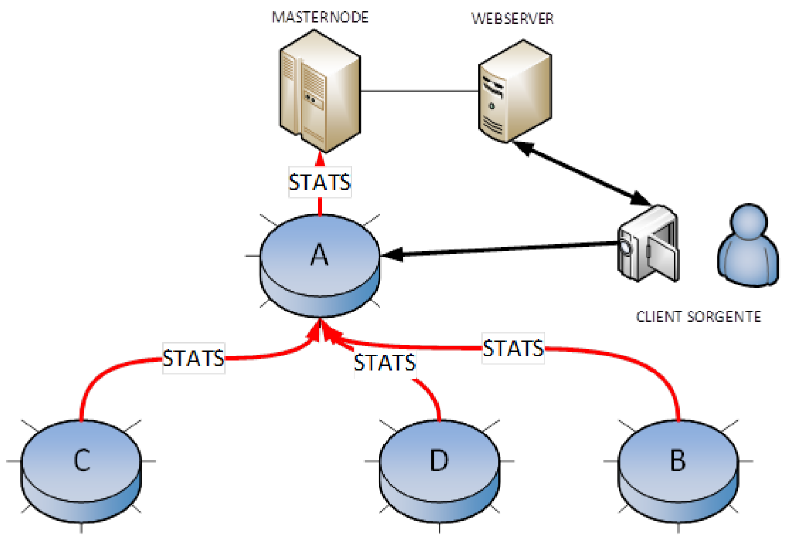


Figure 3.4: SOLEIL aggregate statistic flowing toward the ROOT node

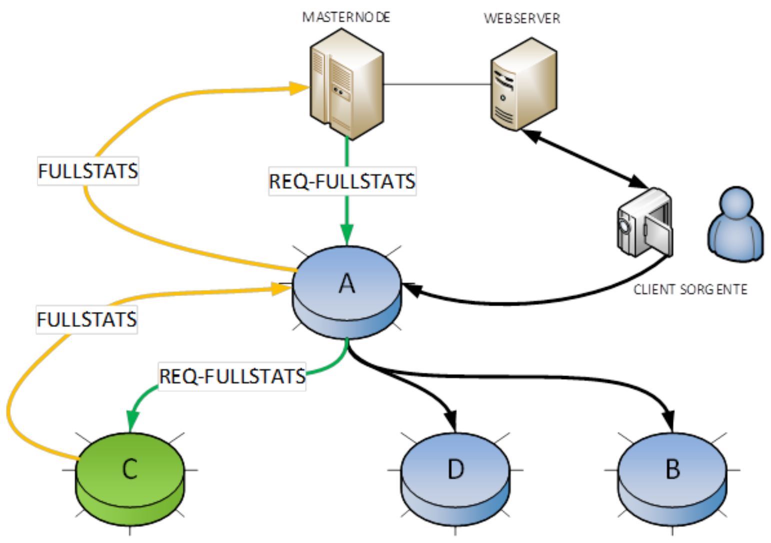


Figure 3.5: SOLEIL: example of `REQFULLSTAT` request and `FULLSTATS` response

As we announced, we will in the next sessions present the structure of the high level components of the SOLEIL system. We will show the main components of a generic node of the network and we will delve into details regarding the specific roles they have to play according to business logic described in Section 1.2.

3.2 Overlay network's architectural elements

3.2.1 Relay node

Internal nodes of the tree are called *relay* nodes and are in charge of dispatching the media flows to other nodes of the system in order to increase the overall scalability. The number of *relay nodes* is dynamic and depends on the number of end users willing to access the service in a specific instant of time. New nodes can dynamically be added to the tree in order to increase the scalability of the system. Vice versa, in order to save resources, a node can be *switched off* in case a large number of users (previously connected to the stream) disconnect and leave the system. This kind of node will be then crucial for both balancing the load of the system and for aggregating RTCP Receiver Report statistic information to send to the *root* node. Each internal node has to manage connection to other nodes (both at a higher and lower level in the tree topology) and to provide methods to exchange and forward SOLEIL Protocol messages with this nodes and with the root node.

The logical architecture of an internal node is depicted in Figure 3.6. Two logical components can be distinguished, the *pipeline* handler and the *session* manager:

- Session manager. This component is in charge of handling connections and SOLEIL Protocol messages. The node has to be able to receive and send message from and to both a directly connected relay node and the root node (in order to receive a `REQFULLSTATS` request and replay with a `FULLSTATS` message. A *Message Handler* element will provide applicative level handling and routing function in order to for-

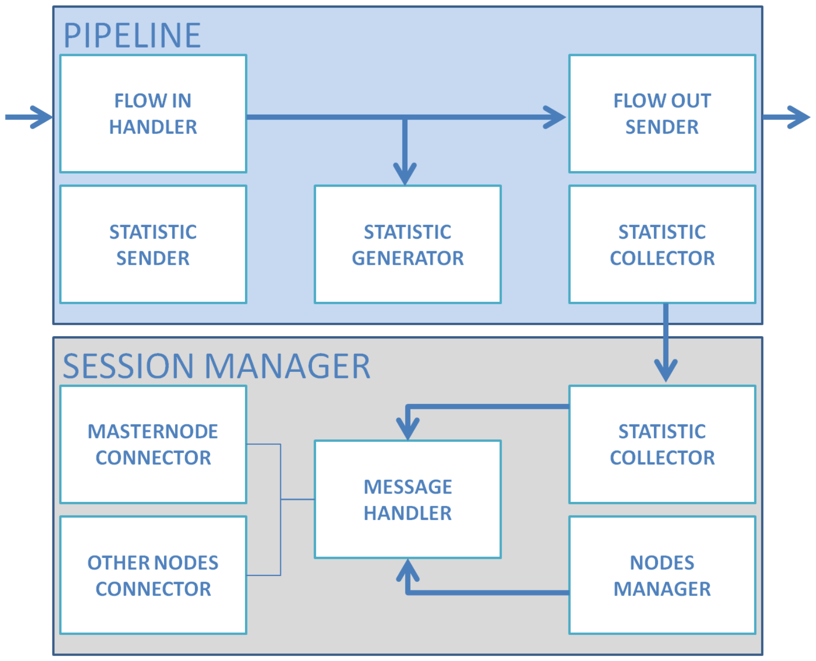


Figure 3.6: Internal node logical architecture

ward messages to the edge nodes of the topology. Last but not least, dedicated elements will collect statistic information coming from both other node (the aggregate statistics) and from the RTP connections the node is involved in;

- Pipeline handler. This is the component of the node which actually handles the media flows. It receives the RTP packets from the root node or from another relay node handles them on order to apply the business logic enforced by the master node and dispatch them to other relay or leaf nodes. This part of the node will also collect statistic information from the RTCP sessions it handles and will send them to the element which elaborates and aggregates them.

It is worth noting that the aforementioned functions are available on root an leaf nodes as well. In fact, from an architectural point of view, they can be seen as relay nodes with further functionalities that will be presented in the following sections.

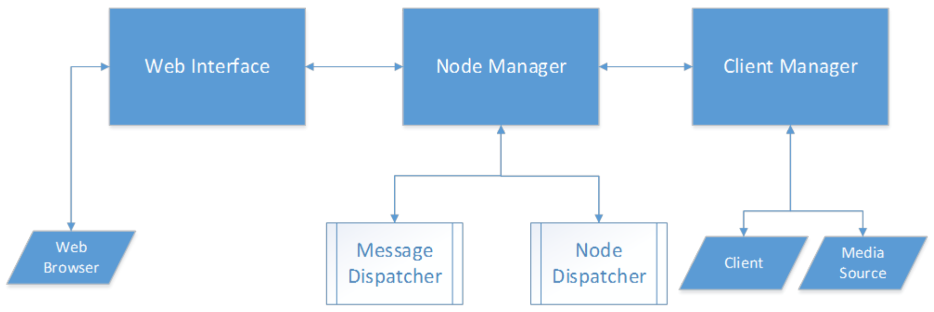


Figure 3.7: Root node logical architecture

3.2.2 Master node

As we explained in Section 1.2.2, the *root* or *master* node is the first node of the tree topology. From an application level point of view this is the entry point of the media streams, namely the place where they are adapted and converted into different formats and resolutions in order to cope with different client-side requirements.

The root node will be also in charge of the management of the entire system. In fact, as will be detailed in the following, this node will receive statistic information about load and bandwidth consumption from the other nodes of the tree and it will be able to react properly in case of problems.

As it is shown in Figure 3.7, in this node are logically concentrated crucial functions of the system:

- Platform administration. In this node is located the administration and control of the entire system. The root node receives information about the status of other node and allows the administrator to control and modify the system itself by means of a web interface. It also embeds automatic monitoring and control functions;

- Topology management. All nodes willing to enter the network have to interact with the *root* node by means of the aforementioned SOLEIL protocol. The *root* will be the glue fixing together all other nodes and the brain controlling them in order to optimize the resource consumption and the load balancing.

- Application management. The *Client Manager* is the part of the system in charge of serve clients' requests. Its two main functions allow to decide to which leaf node a new client has to be connected and to ensure the client itself receives the proper media flow.

3.2.3 Source node and Leaf node

Leaf node, the last node of the tree, the one end user will connect to in order to receive the live stream. This node represent the so-called *last-mile* of the overall architecture and it has to allow final users to access the service in a easy and standard way by means of a pure web interface. In order to fit such a requirements we decided to rely upon the innovative WebRTC technology introduced in Section 2.3. As we explained, WebRTC has been natively conceived to allow communication by means of a pure web browser (i.e., no needing to install any additional extension or plugin) in peer-to-peer, one-to-one scenarios. In order to adapt such paradigm to our slightly different use case, we had to realize a server side component implementing the role of a WebRTC peer at server side. This component, called Janus, plays the role of the leaf node of the SOLEIL architecture. Even though this specific component has been designed and realized by our colleague Lorenzo Miniero in the context of his doctoral research activity, it will be briefly presented in the following section since it represents a key component of the overall system.

The Janus WebRTC gateway

Janus is a WebRTC Gateway [15] conceived to be a general purpose one. As such, it doesn't provide any functionality per se other than implementing the means to set up a WebRTC media communication with a browser, exchanging JSON [20] messages with it, and relaying RTP/RTCP and messages between browsers and the server-side application logic they're attached to.

Since the beginning, the Janus architecture has been conceived as modular. Specifically, it has been designed as a core with a specific set of responsi-

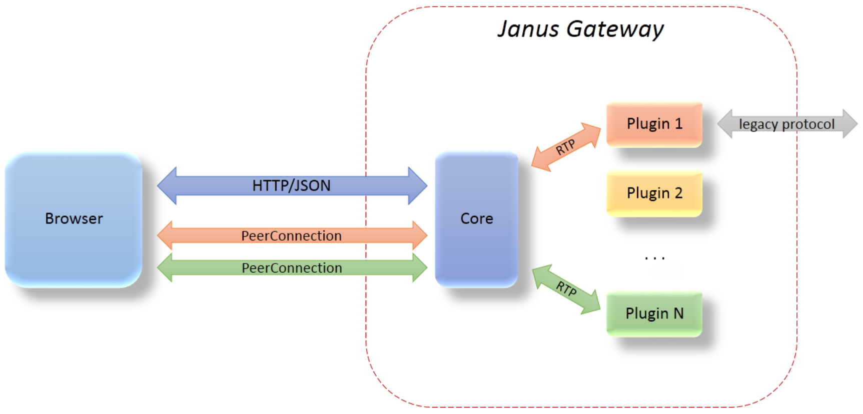


Figure 3.8: Janus modular architecture

bilities, and pluggable modules to provide specific features, namely support for legacy technologies and protocols. We were motivated in following this approach from our former experiences within the IETF MEDIACTRL Working Group [14] . This WG was devoted to the definition of a standard way to implement an effective communication among Application Servers handling application logic, and Media Servers enforcing the related media manipulation tasks. Communication relied on so called *control packages*, allowing the usage of a generic protocol to drive the communication between an application and one or more packages providing specific functionality in a pluggable way.

Considering the effective approach fostered by MEDIACTRL, we chose to follow a similar path for Janus as well, with a core handling the high level communication with users (sessions and handles management, WebRTC-related protocols) and server-side plugins to provide specific functionality in a way that is transparent to WebRTC itself, and as such independent from the web application. An overview of the architecture and the related interactions is depicted in Figure 3.8.

As we conceived it, the core is mostly responsible for three things: i) managing application sessions with users through a REST-ful API; ii) implementing the WebRTC communication with the same users, by taking care of the whole WebRTC lifecycle (negotiation, establishment and management of PeerConnections); iii) attaching users to plugins, in order to allow them to exchange messages (based on a per-plugin ad-hoc protocol) and more importantly media (relaying plain RTP/RTCP). This allows plugins to easily communicate with WebRTC users, as most, if not all, of the complications associated with the WebRTC stack are masked by the gateway core. All plugins need to do is implementing the related plugin API to set up a specific session with users that want to take advantage of their features, and get prepared to receive and/or send RTP packets and related RTCP messages, in case of need.

In order to suit the Janus gateway to our specific use case, a streaming plugin has been realized as well. The streaming plugin allows WebRTC peers to watch/listen to pre-recorded files or live media generated by another tool. Specifically, the plugin currently supports three different types of streams:

1. on-demand streaming of pre-recorded media files (different streaming context for each peer);
2. live streaming of pre-recorded media files (shared streaming context for all peers attached to the stream);
3. live streaming of media generated by another tool (shared streaming context for all peers attached to the stream).

For what concerns types 1. and 2., the only pre-recorded media files that the plugin currently supports are raw PCM mu-law and a-law files: support for other additional widespread formats will be added in the future. For what concerns type 3., instead, the plugin is configured to listen on a couple of ports for RTP: this means that the plugin is implemented to receive RTP (and related RTCP messages as well) on those ports and relay them to all peers attached to that stream. Any tool that can generate audio/video RTP streams and specify a destination is good for the purpose, e.g., *GStreamer*, *FFmpeg*[1], *LibAV*[2]. This makes it really easy to capture and encode whatever desired using one's own favorite tool, and then have it transparently broadcast via WebRTC using Janus.

As it is obvious, this plugins allows us to exactly implement the scenario we had imagined during the design phase of our project.

Once we realized the business logic needed to realize both the mechanisms for creating the distributed delivery tree and the overhead streaming application, we devoted our efforts to the design and implementation of the underlaying enabling network components. We in fact identified and deeply analyzed a class of innovative network technologies that allows to highly improve the performance and management flexibility of our system. The next chapter will be therefore devoted to the presentation of architecture and protocols envisaged by this new approach to the networking known as *Software Defined Networking*(SDN).

[1]http://www.ffmpeg.org/
[2]http://libav.org/

Chapter 4

Low level enabling technologies

The acronym SDN stands for *Software Defined Networks* [39] and it represents an innovative approach to networks, attempting to keep pace with their great diffusion and utilization we are experiencing in latest years. By developing an application on the top of an SDN, a network administrator can obtain a substantial degree of scalability and reconfiguration that they would not be able to reach in any other way. In order to better introduce this revolutionary way of seeing network management, we will first give a brief overview of how actual networking is achieved, with special regard to routing and forwarding functions. We will then expose problems and limitations of this old approach and explain how SDN solves them without affecting existing systems. Finally, we will present OpenFlow [36], a protocol that enables SDN on a network by decoupling the forwarding and the data plane components of a switch.

4.1 Internet: state of the art

Even though a lot of network protocols were developed in the years, the modern Internet makes use of the IP, a datagram based, connectionless protocol. Basic elements of the Internet architecture are routers. These are fairly complex network devices that, once received a datagram, forward it to the next router according to precise rules based on the source and destination IP addresses. Those particular rules are very simple and are essentially based on

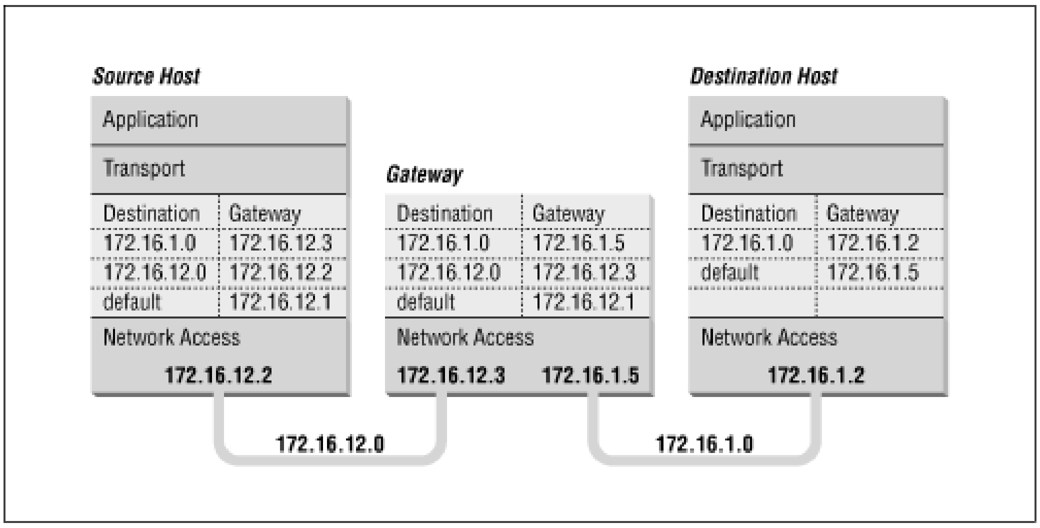


Figure 4.1: Router forwarding table

shortest path algorithms. This means that, once received a packet, a router will try to send it to the next router which is on the shortest path to the destination. To be more precise, there are variations based on the political authority that a router has toward other routers but, putting that aside, the rules are not overall very complex themselves.

To take its decisions in the shortest time possible, a router has a table, called forwarding table, containing associations between a generic address and the interface to whom send it to. This table is constantly updated by automated routing algorithms, making sure that the network view is always consistent and the path is always the one at minimum cost (for example, in case certain links were to go down). This approach has three basic flaws:

1. It is absolute and too simple. Just a few metrics are used to commutate to the next router on the path, and these metrics are mainly focused on the distance;

2. It is essentially static. It does not allow to easily change a route toward an host;

3. It is automated and not customizable. A network administrator cannot

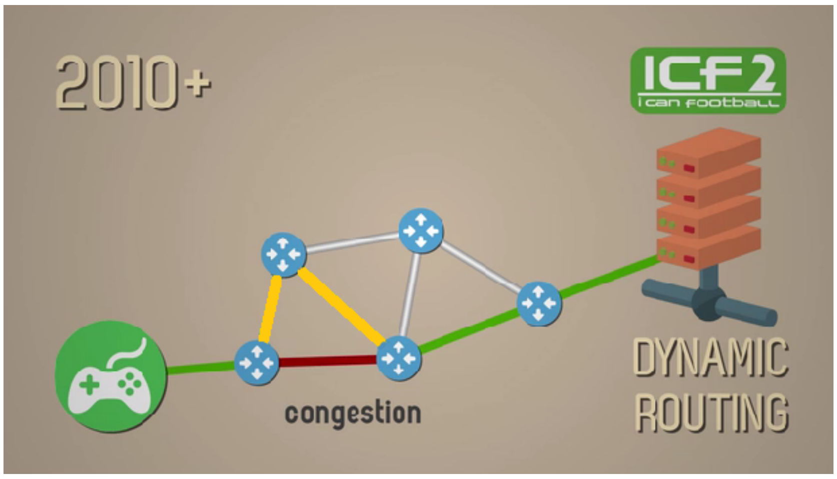


Figure 4.2: Dynamic routing

impose their own rules on an Internet router, nor decide the path their packets will take. Once a packet is sent to the Internet, it will obey to the Internet rules;

Reasons why these constraints are present should be clear: the Internet is public, so metrics and routing rules have to be as more general as possible.

4.1.1 Benefits of network customizations

The approach we just explained was acceptable when the Internet was not very complex, when just a few network based applications existed. Let us briefly analyze, using a couple of practical examples taken from the presentation of the *Ofertie Project* [5], the reasons for which network administrators are interested in altering the logic behind the Internet routing rules. We will examine this matter in depth when we will present the benefits introduced by the *SDN* architecture.

Flexibility

In Figure 4.2 we can see the path taken by the set of packets generated by the client side of a gaming application (marked in green) toward its related server. In the actual Internet, if a link becomes congested (the red link marked in

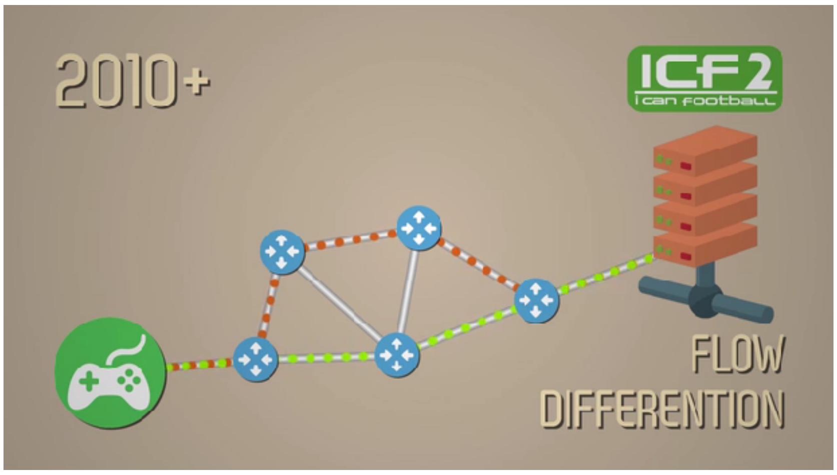


Figure 4.3: Flow differentiation

the figure), packets will keep following the pre-defined path. Of course, some congestion control operations should be adopted anyway for certain routers belonging to an Internet Service Provider's autonomous system, but the real problem is that you cannot be sure if they are actually enforced, nor you can decide the routers crossed by the new path. You also have no way to predict when routing tables will be updated and, even so, the alteration is very limited.

Being able to alter the network not only means to be able to avoid congestions by using alternative paths (like the one marked yellow in the figure) at the chosen moment, but also to freely choose which routers will be actually used in the new route, by implementing custom application-suited algorithms.

Flow differentiation and traffic prioritization

Having again in mind gaming applications, we could think to differentiate flows, as shown in Figure 4.3. The green path represents an high-priority, high-speed path, while the red one is a normal path. This differentiation would allow an administrator to deploy different traffic plans, so providing the users who are paying for the service with better network performances

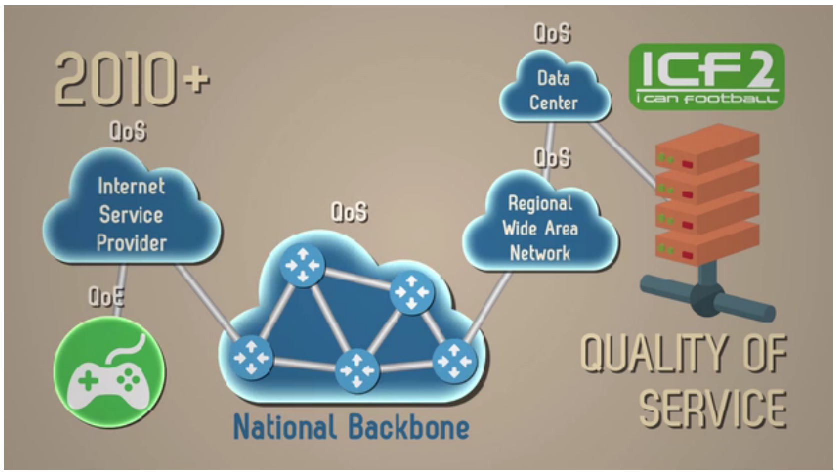


Figure 4.4: Quality of Service

than the ones granted to users accessing it for free. In the actual Internet this is obviously not possible, as routing decisions are made solely based on source/destination IP addresses: when an Internet router receives a packet, it has no extra information than the level 3 header (i.e. no information about the traffic owner), and as its software is single-purpose and does not allow for any sort of customization, it does not give relevance to other fields. Some ISPs can actually do something about flow differentiation, but obviously the number of devices on which they are allowed to act upon is very limited.

Quality of Service

In relation to flow differentiation, a network administrator could be interested in implementing different policies of QoS. In Figure 4.4 we can see different QoSs according to the type of network they can be applied to.

4.2 Software Defined Networks

As stated in the introduction of this chapter (Section 4.2), SDN is an innovative networking approach aimed to give network administrators flexible tools to design, build and manage networks under dynamic and fast-paced environments.

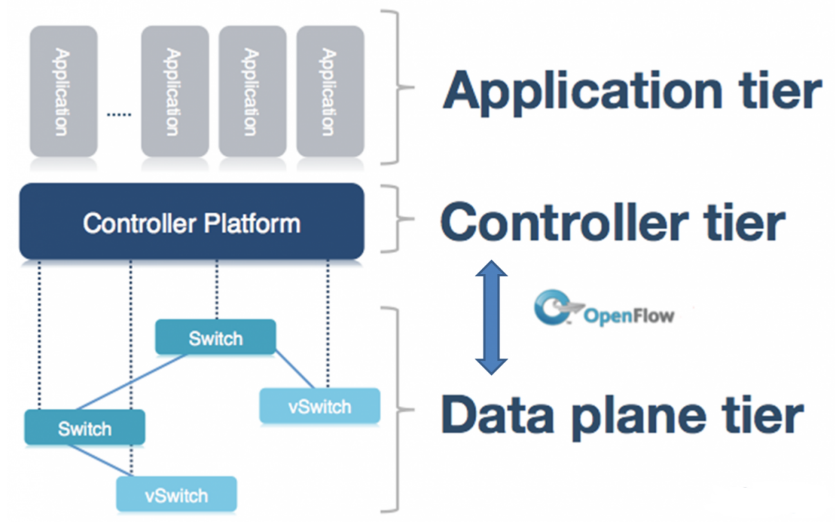


Figure 4.5: SDN Architecture

The basic trait of SDN consists in decoupling the part of a switch that handles decisions, called control plane, from the part that just refers to the forwarding table and forwards a packet on a specific port, named data plane. Thus, an SDN-compliant switch will have a forwarding table as well, updated by following external directives from external entities.

The entity in charge of instructing an SDN-based switch is named controller, and it is basically a software component that, on a side, interacts directly with one or more switches through a well-defined protocol (at the moment of writing, OpenFlow) and, on the other side, checks requests from business applications.

Specifically, as shown in Figure 4.5, an SDN controller has two interfaces:

- A SouthBound API, used by the OpenFlow protocol to communicate with SDN switches;
- A NorthBound API, used for interactions with business applications. Most controllers implement this API as a REST service, but different implementations on different programming languages can also be found.

It's worth noting that two different types of switches are shown:

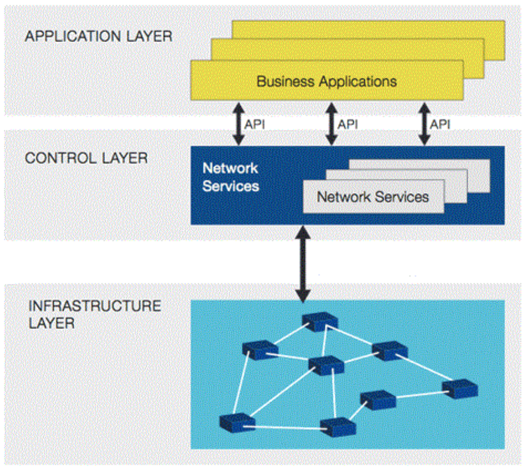


Figure 4.6: SDN General Architecture

- *Normal switches* indicate real hardware devices, optimized for SDN. They allow achieving the highest performances but, as SDN is still young, they are still rare;

- *Virtual switches* indicate software implementations of SDN switches. These are the most common ones and allow every kind of device to become an SDN switch, so not only routers but common PCs and servers as well

Just to clarify, we have to mention that SDN allows more than just one controller to be part of the controller tier and that, in the future, new protocols like OpenFlow could be used for the communication between controller and switches. This leads to the more generic schematization depicted in Figure 4.6.

Here we can observe more than one controller, and more than one NorthBound API. Just one API is shown as SouthBound, as the protocol between a controller and a switch, no matter the controller uses OpenFlow or not, should be unique. The motivations behind this choice are explained in the following section.

4.2.1 SDN: network enhancement

In this section we will give an overview of what are the benefits of the SDN approach and what can be achieved by decoupling the control and data plane. In order to put the focus on the concepts on which an SDN network is built upon rather than on implementation details, we will refer to the schematization of an SDN network shown in Figure 4.5, in which just one controller has envisaged. We will also consider OpenFlow as *the* controller-switch protocol.

First of all, this kind of approach offers a centralized management and control on different network devices. One of the most problematic tasks for both a network administrator and a programmer is interaction with different vendors' switches. In the network of a big company, indeed, it is very uncommon to find only one kind of switch so, in a real-world scenario, multiple devices and software are involved. As each switch has different software modules making use of different protocols, management is often complex and requires the implementation of different communication mechanisms.This problem can be solved if all the switches in the network implemented OpenFlow as communication protocol. Switch vendors can differentiate their switches by many features but as long as their device implement the OpenFlow protocol they can be controlled by the controller in an easy and standard way. In fact, being the controller a single management point, all maintenance and control operations are hugely simplified. Some readers at this point could express their doubt that the OpenFlow controller can represent a single point of failure. It certainly can. Actually, as the controller's only task is to just occasionally instruct switches, its failure will not be as catastrophic as it may appear. Moreover, in case of a component crash, be it a switch or the controller itself, it will be much easier to find its related fault(s) by just examining the controller logs.

Second, SDN offers an abstraction of the underlying network. This point directly relates to the previous paragraph, where we exposed the general benefits of network customization. All the operations we introduced, like altering traffic patterns or offering to users specific QoS levels, are hindered by the number and the complexity of the low level devices, which have to be contacted in order to achieve the expected result. Furthermore, since other technologies could already be in place, it's very important to interact with them as well. In such a situation, an high-level operation like traffic orchestration, is mixed with low-level operations, like communication with switches. By exploiting the aforementioned NorthBound APIs of an SDN controller the real network is abstracted from the business application. This feature makes programmers' job easier as well, because they just have to learn one API for all switches no matter of the internal structure of the switches nor of software changes or updates. It is not uncommon for administrators to keep older versions of network devices software, as an update could, in the worst case scenario, disrupt an entire service and affect interactions with other existent apparatus. With SDN and OpenFlow, software updates have a very limited impact: vendors will have to simply make sure that a new switch update does not alter the OpenFlow protocol part, whereas network administrators will keep on dialoging with the switches. Figure 4.7 shows a practical example of what we are talking about. We can note how different kinds of business applications, cloud-based or not, interact with the underlying network just by contacting the NorthBound API. Also note how the OpenFlow network can coexist, setting a very simple configuration, with other generic virtual network software components.

The last important benefit we have to talk about is related to the previous section as well, and it is about the agility and scalability that SDN brings to the network. Let us start by pointing out that the assertion that the actual Internet is static is clearly in contrast with the dynamic nature of modern server applications and services. Nowadays, most servers are virtualized and almost all applications running on such VMs interact with each other using various communication protocols, so they need to be properly configured. We illustrate virtualization in-depth in 2.2, as its understanding is required to fully comprehend our application. However, what we want to make clear here is that while in the past a server application typically ex-

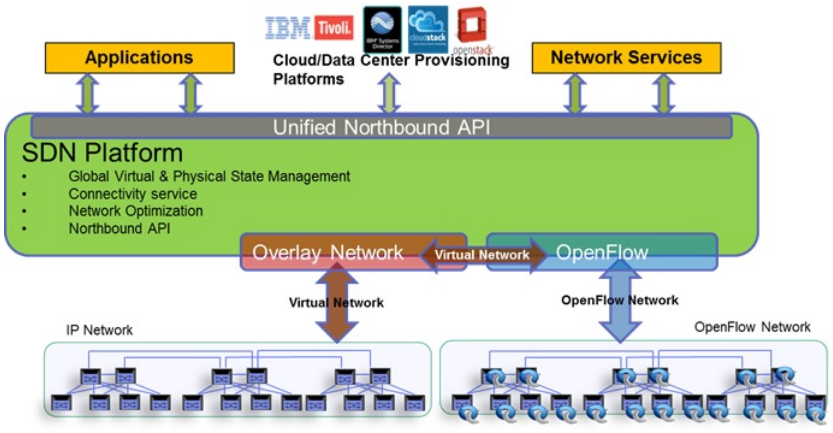


Figure 4.7: Overview of SDN's provided abstraction

changed data just with clients, applications have now to work toether and communicate with each other in order to provide the final service. In addition, being each server virtual, this can be migrated from a location to another. Motivations behind migration can be multiple, but this is mainly done with the purpose of optimize and rebalance servers workload. Nevertheless, this operation, which includes simpler cases as boot up/shut down, has to be done trying to avoid long period of service downtime. Finally, given their constant interaction and increase in number, we have to admit that a static configuration of the network could not be enough anymore to assure fast data exchanging among modern and complex applications. Just to give an example, Google computing nodes, in order to maintain consistency among the Internet and their internal databases (representing the source for user searches), have to periodically exchange data on the order of petabytes. To allow this mass of data to flow, an hyper scalable network infrastructure is needed. In fact this type of network cannot be statically configured, as a single change would involve a not negligible downtime to reconfigure. With SDN, again, a controller can act as a single management point and implement ad-hoc algorithms to dynamically alter the network. As the controller can see the state of all the network switches, it is aware of their changes and so, properly configuring them, can manage any network change in a consistent way. For example, switches can be instructed to change destination of a traffic flow in case the recipient changed its location or was shut down, thus solving the migration problem.

4.3 OpenFlow

OpenFlow [36] is, at the moment, the SDN-defined controller-switch protocol and is supported by all known SDN switches implementations. Given its importance we will spend this paragraph to explain it in detail.

In order to implement the SDN architecture, OpenFlow introduces the concept of flow. In telecommunications, a flow is a very generic term and represents an amount of data exchanged by two or more end-points which can be identified or categorized in some way. For example, a TCP communication can be identified as a flow because it is easy to recognize TCP packets between two hosts by just looking at the couples `<source IP address, source TCP port>`, `destination IP address, destination TCP port>`. Anyway, for the same reason, you could distinguish HTTP flows by restricting the flow to the destination port 80. Or you could just stop to layer 3 and identify a flow as data exchanged by a specific couple `<source IP address>`, `<destination IP address>`. So, to clarify this point, in the following section will be explained how OpenFlow actually manages the flows.

4.3.1 Flows and flow tables

A flow is composed by two parts:

- A matching rule;
- An action to be executed on a packet belonging to the matched flow.

A matching rule is what is used by OpenFlow to univocally identify a flow and also the way for the end user to set the specification degree suited

Ingress Port	Metadata	Ether src	Ether dst	Ether type	VLAN id	VLAN priority	MPLS label	MPLS traffic class	IPv4 src	IPv4 dst	IPv4 proto / ARP opcode	IPv4 ToS bits	TCP/ UDP / SCTP src port ICMP Type	TCP/ UDP / SCTP dst port ICMP Code

Figure 4.8: Openflow's matching fields

for their application. The Figure 4.8 shows all the current packet fields supported for the identification of a generic flow.

Most common parts of layer 4 and layer 3 headers, in addition to some VLAN fields and an OpenFlow specific one that works as an ingress port (the specific switch port a certain packet was actually received on), are supported for the identification of a generic flow. The action field can be composed by a single action or by multiple actions to be executed in a specific order. In the OpenFlow specification, many actions are defined and, as the standard keeps evolving, new actions can be introduced. We will in the following give just some examples of applicable actions:

- `Output action`, probably the most common action, consists in forwarding a packet to a specific output port or to another flow table. The specifications allowes the use of some wildcards representing, for example, 'all ports', 'all ports but the port on which the packet was received', 'forward to controller';

- `Set action` can alter a packet source/destination address or port. It cannot exists on its own, but requires that at least one output action has been specified;

- `Drop action`. Simply drops a packet;

- `Enqueue action` to store packets in a switch queue for further processing.

Flows are organized in flow tables. Each row of a table is called flow rule and it represents a rule to be applied by a switch for a specific flow. A

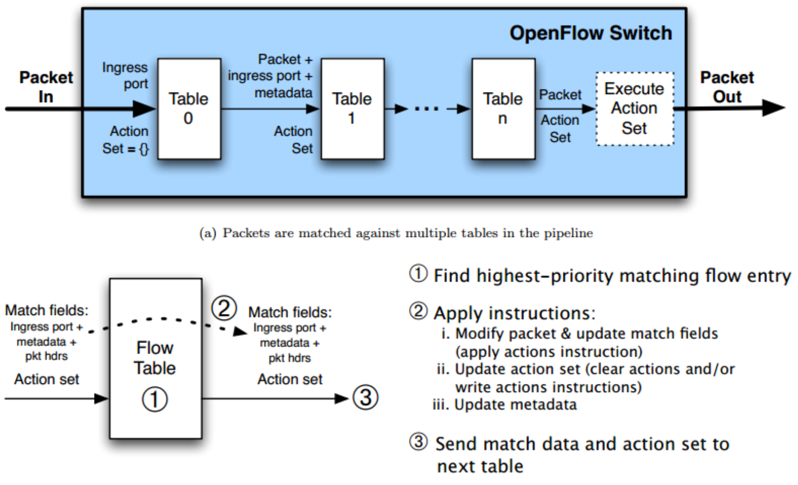

OpenFlow Switch
Packet In
Ingress port
Action Set = {}
Table 0
Packet + ingress port + metadata
Action Set
Table 1
Table n
Packet
Action Set
Execute Action Set
Packet Out
(a) Packets are matched against multiple tables in the pipeline
Match fields: Ingress port + metadata + pkt hdrs
Action set
Flow Table ①
② Match fields: Ingress port + metadata + pkt hdrs
Action set
③
① Find highest-priority matching flow entry
② Apply instructions:
i. Modify packet & update match fields (apply actions instruction)
ii. Update action set (clear actions and/or write actions instructions)
iii. Update metadata
③ Send match data and action set to next table

Figure 4.9: Openflow's tables

Group Identifier	Group Type	Counters	Action Buckets

Figure 4.10: Openflow's group table fields

switch can have multiple flow tables so to be able to implement more complex behavior through further processing the same packet, as shown in Figure 4.9.

An extra table, called group table, can be present. This table allowes to group entries and gives OpenFlow more forwarding methods, like 'select'and 'all', and let it to apply a bucket of actionsto flows belonging to a specific group. Let us briefly explain each field of this table (shown in Figure 4.8).

- `Group Identifier` is a 32-bit unsigned integer identifying the group;

- `Group Type` is a semantic information to indicate which bucket will be executed, and can assume the following values:

  `All`. All of the buckets, this value can be used for broadcasting purpose;

  `Select`. Selection of just one bucket, usually implemented by a dynamic algorithm. For example, it can select the output on the currently least congested port;

`Indirect`. Executes a defined bucket. This group supports only a single bucket. Allows multiple flow entries or groups to point to a common group identifier, supporting faster, more efficient convergence (e.g. next hops for IP forwarding);

`Fast failover`. Executes the first *live* bucket. Each action bucket is associated with a specific port and/or group that controls its liveness. The buckets are evaluated in the order defined by the group, and the first bucket which is associated with a live port/group is selected. This group type enables the switch to change forwarding without requiring a round trip to the controller. If no buckets are live, packets are dropped. This group type must implement a liveness mechanism.

- `Counters` is a field updated when packets are processed by a group;
- `Action Buckets` is an ordered list of action buckets, where each action bucket contains a set of actions to execute and associated parameters.

It's worth nothing that each rule can have some utility fields, including:

- `Timers` indicating the time validity of a rule. Specifically:

  `Idle Timer`, also known as soft timer, defines a temporary window during wich the rule is valid. If at least a matching packet is received during this wondows the timeout is reset, otherwise the rule is deleted;

  `Hard timer`. This is a fix timeout, no matter how many packets matching the rule are received, after a fixed period of time the rule will be deleted.

- `Counters`. They are used to keep trace of various statistics (number, type and so on) related to packets and are updated when they are processed;

4.3.2 Overall architecture

The protocol specification currently envisages two different types of OpenFlow switches:

- Dedicated OpenFlow switches, which just implement the OpenFlow protocol;
- OpenFlow enabled switches, traditional layer 2 or layer 3 switches supporting the OpenFlow protocol.

The first type of switch can just be used if an OpenFlow controller is present in the network. The second one instead can act as a traditional layer 2/3 switches but, in case a controller is activated in the network, it can change its behavior halting the normal operational mode and starting waiting for flow rules enforcement. Eventhough the latter is the most common and flexible type of SDN compliant switch, they both exhibit the same internal structure (Figure 4.8):

- One or more OpenFlow tables used to process incoming flows;
- A secure channel to communicate with the controller.

Let us now explain the interactions between a switch and a controller. When a controller is connected to a switch the latter stops acting as a normal switch and starts referring to internal tables which will be, initially, empty. When a packet with no matching rule is received, the switch contacts the controller asking for a flow rule and the packets is enqueued (providing there is enough available space). The controller replies with a flow rule to apply to that packet. The switch updates its flow table and processes the packet. This mechanism is summarized in Figure 4.12.

It's worth noting that, once a flow rule has been installed on the switch, every packet matching that rule will be automatically processed so limiting the overhead (due to the interaction with the controller) just to the first step. For this reason, most of OpenFlow switches implement the flow table

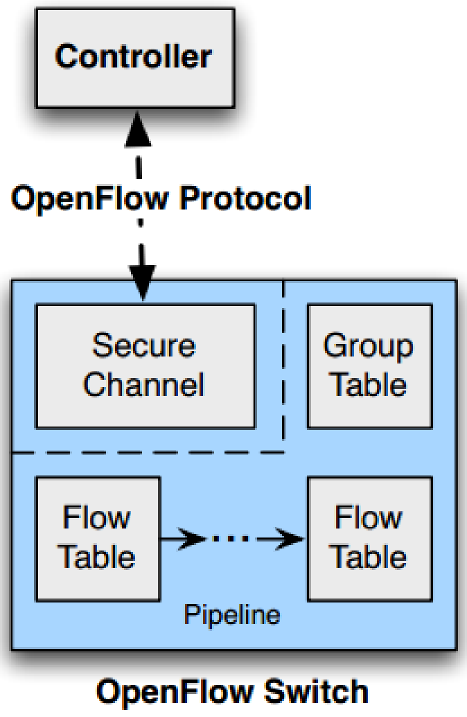


Figure 4.11: Openflow's switch architecture

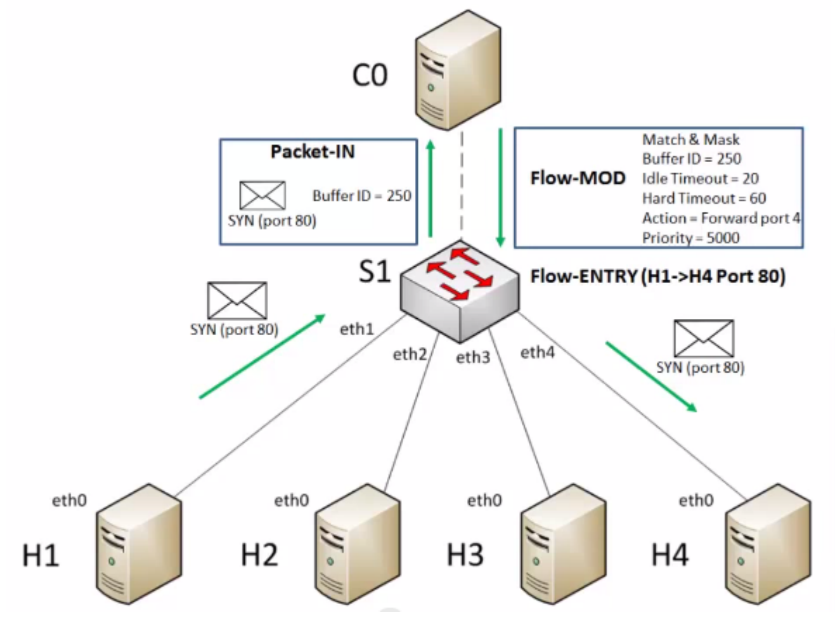


Figure 4.12: Openflow: protocol logic

in hardware, differently from the secure channel part, which is software-based. The fact that the forwarding part is done in hardware means that the efficiency of a switch, provided a flow rules exists, will be about on the same order of a traditional layer 3 switch.

Also note that a network administrator can create complex flow rules by composing several of the available protocol fields and parameters. This implies that very specific rules can be enforced. Obviously this degree of freedom lead to an increase of the overhead due to different interactions between switch and controller. A more generic rule, instead, could dramatically reduce the number of interactions but it could not be specific enough to enforce the control required by the application level business logic.

In conclusion, the OpenFlow protocol provides network administrators with huge freedom and flexibility in how the network is used, operated, and sold. The software that governs the network can be written by enterprises and service providers using ordinary software environments so promoting rapid services introduction through customization.

Chapter 5

SOLEIL *low level* network design

As already stated in the previous chapters, the SOLEIL system has been conceived to be deployed on a cloud infrastructure in order to exploit the huge advantages this technology introduces in terms of flexibility and scalability. Furthermore, in order to have full control on the way the streams of data flow from the source (the *broadcater*) to several destinations, we decided to rely on the novel Software Defined Network approach for what concerns the enabling underlying network infrastructure. In this section we present our design of the SOLEIL *low level* network describing the internal structure of each key component and motivating architectural choices.

5.1 Nodes architecture

Starting from assumptions and functional requirements introduced in the previous chapters, it's quite natural to define the general architecture of a generic node of the *SOLEIL* architecture. As it is shown in Figure 5.1, each node is composed by two main elements:

- An *OpenFlow* switch;
- An *Hypervisor* in charge of hosting and handling one or more *Virtual Machines*;

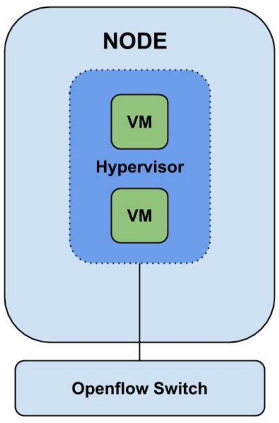


Figure 5.1: SOLEIL: generic node architecture

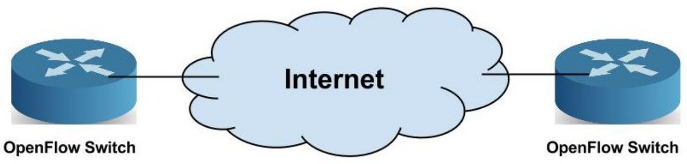


Figure 5.2: SDN nodes deployment

As we said, the hypervisor is needed in order to deploy our architecture in a cloud environment. The need for an *OpenFlow* switch attachment point is motivated by the assumption that in each node can reside multiple *VMs* and, from an architectural point of view, these VMs can be seen as real hosts. Each host can deliver a different type of service (i.e. both different streams and different formats of the same stream) and, in order to allow the proper high level differentiation and forwarding of media packets, the related media flows need to be intercepted and handled in the right way. In order to better explain the need for an OpenFlow switch in each node, it's worth noting that the *SDN* specification doesn't impose that every single router in a network topology has to be *OpenFlow-enabled*. For this reason, several different *standard* routers, could be on the path between two SDN nodes in a network (see Figure 5.2).

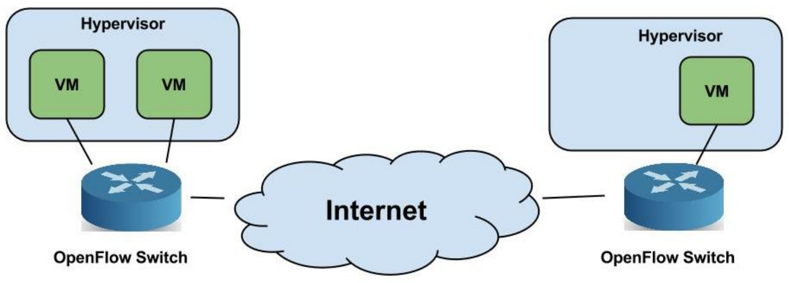


Figure 5.3: SDN nodes deployment in a real world scenario

This means that, in the most generic case, the *hypervisor* could be attached to a *not-SDN switch*. Even in this case, if each *VM* had a unique IP address the flow rules could still be established by imposing a proper *matching rule* (i.e. checking the source addresses of each packet). However, in most of the cases the *hypervisor* itself is on a LAN environment, and access the Internet via a NAT element. In such cases, the source field of each data packet originated by different *VMs* behind the NAT is replaced with the same IP address (the NAT public IP address) when leaving the LAN limits, thus causing the aforementioned differentiation technique to be useless. So, in order to correctly handle the traffic originated by each virtual machine, we impose the first hop crossed by the traffic is an OpenFlow node.

Standing these considerations, the previous general schematizations can be integrated as in Figure 5.3.

It's worth noting that the mechanism allowing an OpenFlow switches to discover each other and communicate over the Internet is an implementation depending detail and, for this reason, we can give it for granted. Just for the sake of clarity, we can say that in most implementations of OpenFlow switches this task is achieved by means of application level *tunneling/encapsulation* (Generic routing encapsulation - GRE [25]).

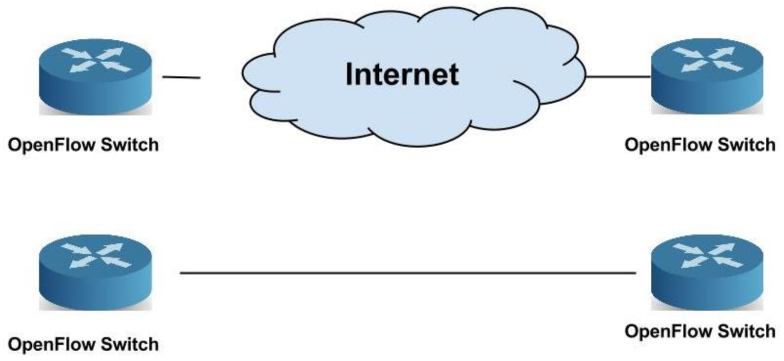


Figure 5.4: Logical link abstraction

5.2 Low level network design

Before starting the description of the SOLEIL low level design, we want here to specify that, from this moment on, we will use the term 'link' to refer to its meaning of *logical connection* between two OpenFlow switches, no matter other *non-OpenFlow* switches or network devices are present in the middle (Figure 5.4).

Given the agility and scalability requirements that characterize our system, we decided to adopt a hierarchical topology, more specifically a tree. Due to the absence of circuits, this topology allows for very simple routing and management algorithms, as just one path connecting two nodes exists. However, being just one path also means that, in case of failures, some nodes will not be reachable, conflicting with the fault tolerance requirement. For this reason, the system administrator can specify a set of redundancy links among switches, so that in case a switch chases, an alternative path will be found.

In Figure 5.5 is depicted an example of alternative paths between layer 2 and layer 3 switches. If a failure occurs on a specific link (the one connecting 2-2 to 3-2 in this case), an alternative path is available, making use of the switch 2-1. Obviously, this *fall-back* feature is not always granted as it depends on which specific redundancy links are actually present. It is worth noting that the choice of inserting or not redundancy links among network

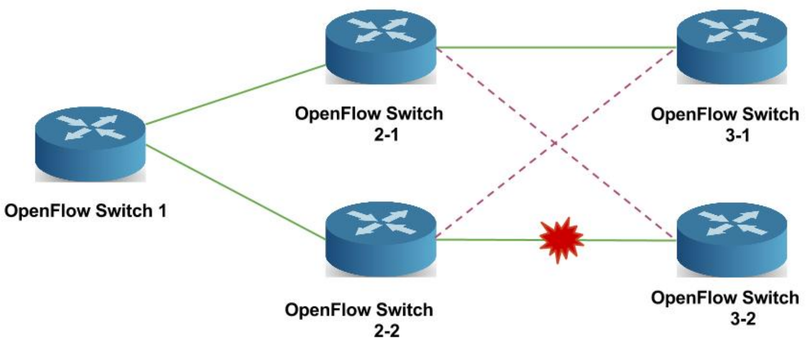


Figure 5.5: Redundancy links among OpenFlow switches

components is very specific and depends on the application level logic of the system. For this reason we decided to leave the to the system administrator. For the same reason, the creation of the whole topology, including for example the number of levels of the tree and how each node is connected to the others can be defined by the administrator during the service set-up phase. As will be explained in details in the following sections, once the user completes the system configuration, automatic routines will be executed in order to cope with possible human errors and to optimize the topology creation itself.

5.2.1 Bridging mode

As already stated, in a cloud environment virtual hosts should be considered functionally equivalent to physical machines. Although any virtualization technology allows the hypervisor to act as a NAT element and to hide the VMs to the external world, this is not feasible in our scenario. What is instead very helpful to realize our high level logic is the so-called *bridge mode* which allows to directly attach the virtual machines to the external network. This operational mode is very simple: even though there is just one network adapter on the physical host, the hypervisor creates several virtual bridging interfaces, one for each virtual machine, which actually just redirect the traffic to the *Network Interface* (NIC). Figure 5.6 gives an overview of the operational mode just introduced.

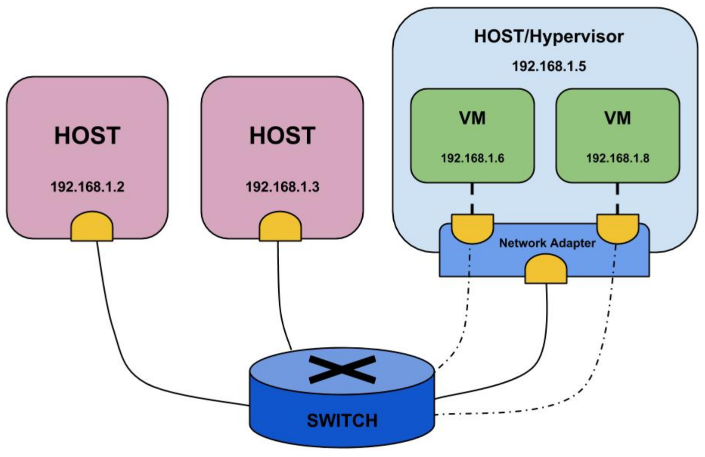


Figure 5.6: Bridge mode example

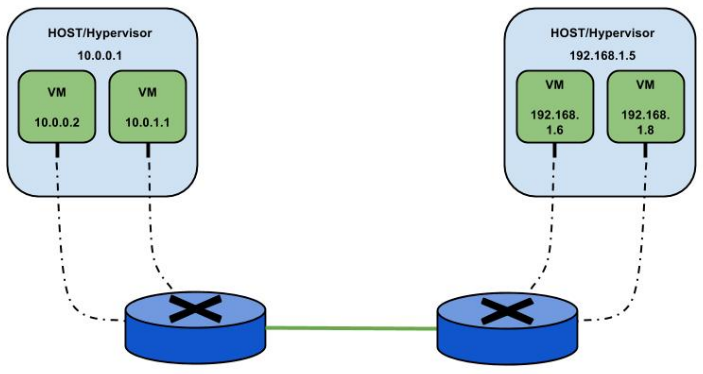


Figure 5.7: Bridge mode in a *global* scenario

Analyzing the figure, it can be noted how a specific host is not aware whether it's connected to *real* or *virtual* network adapter. Figure 5.6 also let us note that, for the same reason, both real and virtual hosts result connected to the same logic network. Some problem can arise when moving our point of view from a local to a global perspective.

In the global scenario depicted in Figure 5.7, in fact, each switch is used by VMs belonging to different LANs. In this case we have a critical addressing issue since the two LANs adopt different addressing spaces. In such a situation in fact, since by means of a tunneling mechanism between switches the private local addresses can travel through the Internet the VMs running in different LANs environment are cannot communicate in a transparent way. This specific problem could be avoided imposing that all the subnets used the same private subnet addressing space. Unfortunately, this option is not always feasible since the physical infrastructures hosting the VMs could be used by other processes with their own specific network requirements. Moreover, this would not solve IP conflicts as, among dozens of LANs (each one regulated by their own DHCP server) it is likely to assign the same IP address to two or more VMs. Last but not least, another problem has to be taken into account in case a node migration is needed. In this case, in fact, in order to successfully complete the migration process, a change of address for the migrated VM could be needed, leading to an additional overhead due to reconfiguration operations on that node.

5.2.2 Overlay and low level networks configuration

To cope with such issues, we opted for an overlay network architecture. As hosts are virtual, it seems just appropriate to make the network they are plugged in virtual as well. This approach solves every problem exposed so far. Figure 5.8 shows how virtual machines have the illusion of belonging to the same network, a $10.x.x.x/8$, even though host nodes reside on different local networks. The fact VMs belong to the same network means that, for any application running on them, there is no need to worry about network details like addresses translations and port forwarding. This makes guest cooperation immediate, a very desirable requirement on a cloud computing infrastructure.

The only aspect we have to take particular car of is addressing. At the very early stage of our work, we had thought of just assigning static addresses to the VMs, as the fact that VM network is a virtual network meant that just VMs could see each other. Anyway, this approach is not optimal as it is needed to store somewhere current associations. Moreover, each time a virtual machine is plugged in, it is necessary to interact with its OS in order

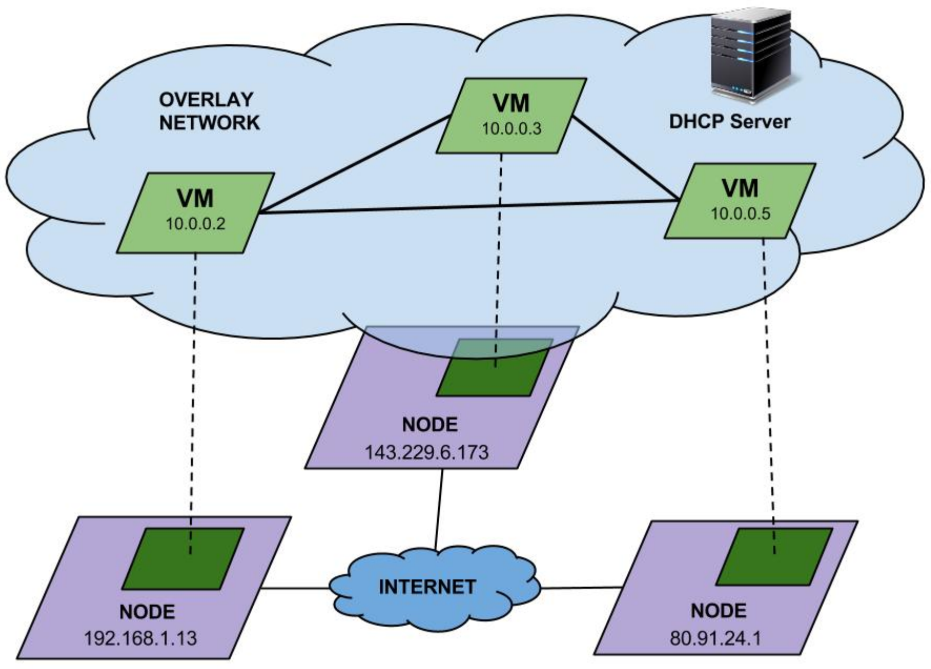


Figure 5.8: Example of overlay network configuration

to set the static IP and properly configure application network parameters. For this reason, we decided to use a *Dynamic Host Configuration Protocol* (DHCP) [23] server for the overlay network as well, so to introduce the dynamicity needed in this particular operational environment. So, in order for our system to work correctly, we set as mandatory the installation of a DHCP server on the root node of the tree. This constraint ensure the DHCP server is always reachable by new nodes *entering* the overlay.

5.2.3 Path, flows and role based tree

In Section 4.2 we stated that when a controller is present into an SDN enabled network, the traffic has to be explicitly allowed by flow rules. This means that just creating connections between switches does not automatically make a stream flow. We decided to build our network logic around this concept and we adopted the following rules for the delivery of the service: in order to make a stream flow from the root to a leaf, the end user willing to access it

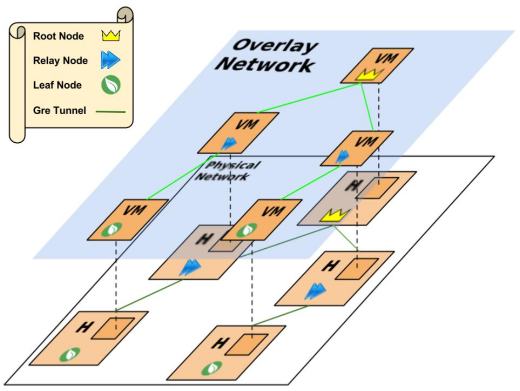


Figure 5.9: *Role-based* tree

has to explicitly ask for it. In our logic, we will define a *path* as a sequence of switches and links starting from the root and reaching a leaf. Each switch crossed by the *path* has to be instructed (by installing a proper *flow rule*) about the way it has to handle data packets. In other words, a path can be defined as such only if it exists in the system a flow rule allowing packets to travel across a well defined set of switches. Adopting the same notation used for the high-level architectural components, we can distinguish three types of nodes:

- *Root* node;
- *Relay* node;
- *Leaf* node;

A VM present on one of these nodes will become respectively a root, a relay or a leaf VM. In the aforementioned system set-up phase, the administrator is asked to mark a node with one of these roles thus defining a so-called *role-based tree* (see Figure 5.9).

According to the node type and depending on whether a VM belonging to the overlay network is present or not, the flow installation procedure will slightly change. Specifically, if a node is marked as root or does not host a VM, an application packet has to be sent straight forward to the next

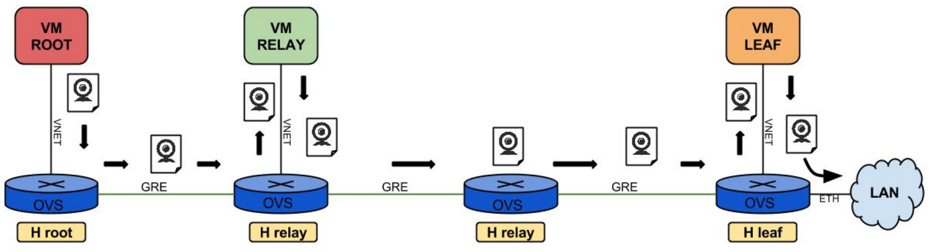


Figure 5.10: Differentiation of flow rules on different nodes of the network

hop. In fact, if the node is the *root node* of the low level tree, the only received packets will just come from the *root VM*. in case a VM is present on a *relay node*, packets will have to be forwarded first to the VM and then, from the VM, to the next hop, as exemplified in Figure 5.10. Note that by 'packets' we are here referring exclusively to application related packets. Management packets (like DHCP or ARP) are not contemplated by this forwarding algorithm.

Physical and Virtual network interaction

As we explained in the previous section, the introduction of the overlay network solves the addressing problem, but it does not fully explain how a generic client can interact with a VM running on it. Figure 5.11 shows a generic OpenFlow switch interfaces in a typical real world scenario. We can distinguish two different interfaces:

- An interface to the overlay network (i.e. the 10.0.0.x/8 IP network);
- An interface to the *real* network (i.e. the 192.168.1.0/24 IP network.

Our idea is to exploit the overlay network as a delivery network, so that any application level flow will travel across its nodes toward its final destination. During their travel packets will touch several switches on the physical network and VMs on the overlay network, but from the final user point of view the overlay network can be seen as *black-box*. In this architecture in fact, each OpenFlow switch is in charge of translating network addresses to

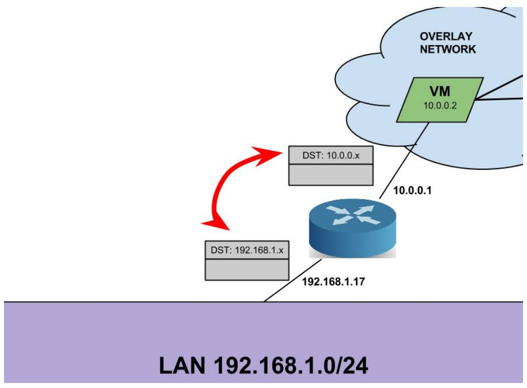


Figure 5.11: Interconnection between overlay and physical network

make possible for packets to travel in a transparent way over the two different networks. This, as we will explain in the following, will be done by applying proper flow rules on the OpenFlow switches.

Physical and Virtual network isolation

The last considerations allow us to introduce two additional functional requirements our architecture has to assure:

1. Control messages and external traffic have to be treated in a different way respect to overlay network traffic. Only packets generated by the components of our application level have to be able to flow through the overlay network. The external traffic from other hosts or traffic used for infrastructure management (flow rules installation procedures, bridge creation commands, etc.) should be confined to the physical network;

2. The overlay DHCP server has to answer only to requests from hosts belonging to the overaly network. In other words, the DHCP server has to be isolated from other hosts belonging to the same underlying physical network.

As the plug of an OpenFlow controller makes the network isolated, the readers could think that these requirements are already respected as any interaction would have to be explicitly enforced by flow rules. That is true

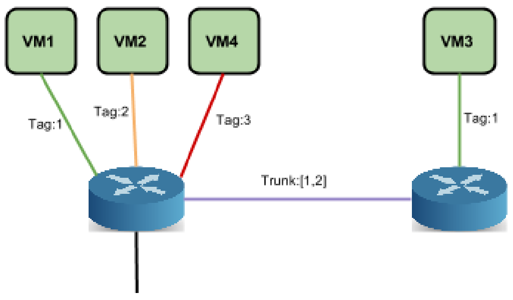


Figure 5.12: VLAN configuration example

as long as a controller is present in the network, but the problem could arise in case that component crashes or is simply shut down, especially if the OpenFlow switches are implemented to act as normal layer 2 devices when no controller is present. We found the solution to this problem thanks to the adoption of *Virtual LANs (VLANs)* [43], a mechanism implemented by almost every switchable to isolate specific segments of a LAN, leading to the creation of multiple virtual networks on the same physical network. This mechanism is implemented by means of *tags*, plain numbers identifying segments belonging to the same virtual network. In particular, segments can be identified as:

- *Tag ports.* Segments connecting an host to a switch;
- *Trunks.* Segments connecting switches. A trunk specifies a list of VLAN tags it is allowed to pass traffic from/to.

Figure 5.12 shows an example of VLAN configuration.

VM1 and VM3 have tag 1 thus belonging to the same VLAN, while VM2 and VM4 are tagged respectively 2 and 3. The link connecting the switches is a trunk for VLANs 1 and 2 so just VM1, VM3 and VM2 traffic can flow on this link, while VM4 has to find another way. Once on the rightmost switch, just VM1 traffic can reach VM3 because of them being on the same VLAN, while packets from VM4 will not be visible by VM3. So, we have been able to cope with the aforementioned issue, just marking overlay network links with a specific VLAN tag. This completely solves our problem as:

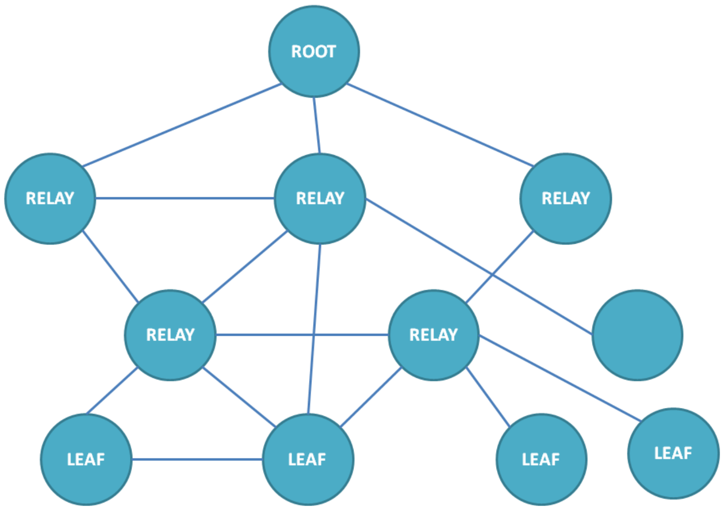


Figure 5.13: Administrator system high level configuration

- When a controller is present, VLAN tags are ignored and the isolation is enforced by flow rules;
- When there is no controller, isolation is still guaranteed by the VLAN mechanism.

5.3 Automatic configuration of the low level tree topology

As it will be detailed in Section 6.5.2, the creation and configuration of the tree topology which allows to deliver the service to end users can be defined by the system administrator by means of a web panel designed and created *ad-hoc*. As we anticipated, automatic routines are executed by the system in order to cope with possible human errors and to optimize the topology creation process itself. In this section we present the algorithm we implemented to achieve this complex task. The system administration can exploit graphic tools to *draw* and configure the nodes to be deployed. The outcome of this operation is an high level definition of the system components that can be schematized as in Figure 5.13.

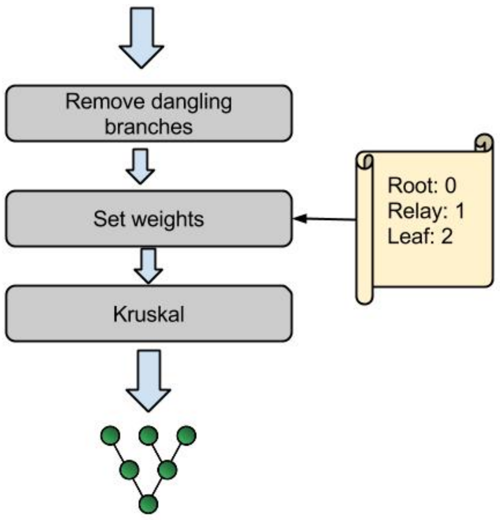


Figure 5.14: Tree minimization algorithm

With reference to the figure, we can note the administrator wants to create a distribution network composed by eleven nodes. The administrator can assign a specific role to each node in order to deploy the overlay network according to the peculiar features of the SOLEIL architecture. Furthermore, it's possible draw connection links among nodes (having the possibility to specify redundancy links that can be used in case of problems, as it has been explained in Section 5.2). In order the system can find the minimum tree by starting from this user level description of the topology we opted to rely on a consolidated minimization algorithm, namely the *Kruskal's Minimum Spanning Tree* algorithm, combined with some application logic, as it is shown in Figure 5.14.

In fact, the algorithm in its original form is obviously unaware of the role-based nature of the SOLEIL tree and, by just giving it a generic topology, we would have no guarantees that the role of a certain node is respected when computing the minimum tree. Moreover, as we anticipated, the user could have made errors (e.g. relay nodes either not connected to any leaf or to the root node) during the topology definition phase. For this reason, before feeding the algorithm with the topology, the system removes any dangling

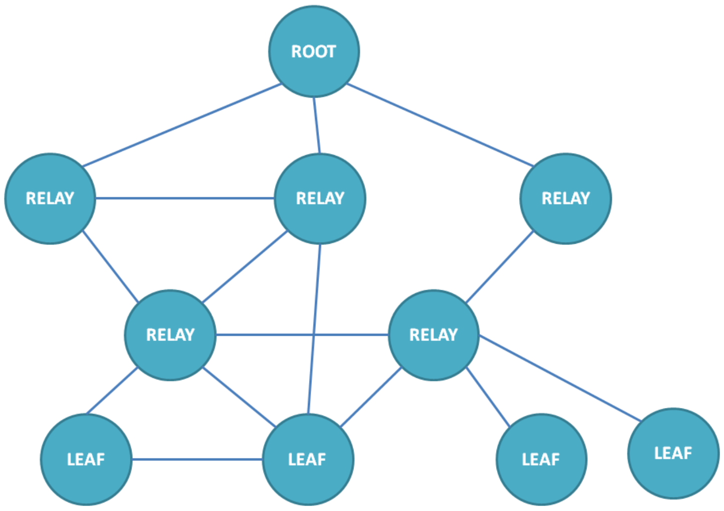


Figure 5.15: Tree topology creation: pruning of *dandling* branches

branches, which are branches ending with a relay or a node whose role was not set, attaining the new configuration shown in Figure 5.15.

Then, to make sure that the computed tree is the one the user wanted, we appropriately set link weights according to the type of its endpoints. In particular, each node is associated to a weight value according to the following mapping:

- Root: 0
- Relay:1
- Leaf: 2

and a link weight is set calculating the sum of the values associated to the endpoints the link connects (see Figure 5.16).

Finally, we give this topology to the real algorithm. According to how the Kruskal's algorithm is implemented [46], this process will get us the minimum tree respecting SOLEIL roles (Figure 5.17).

Only links which belong to the tree will be actually translated into actual tunnels, because they are the only links needed for our traffic to flow. Redundant links will be stored and used whenever necessary. We decided to store

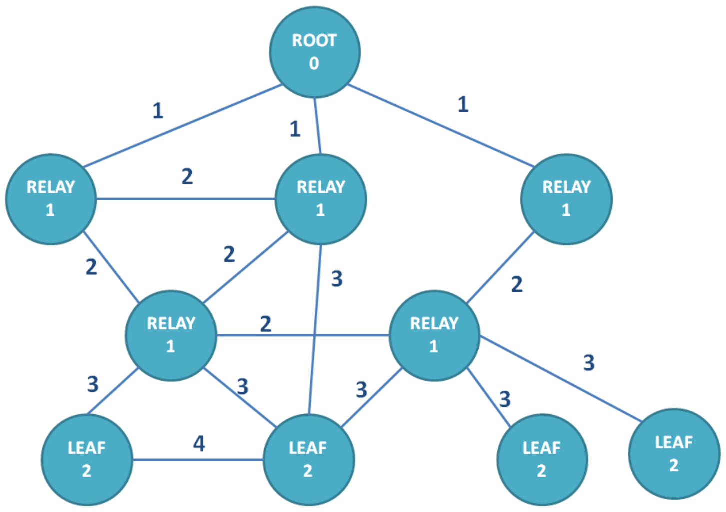


Figure 5.16: Weights assigned to nodes and links

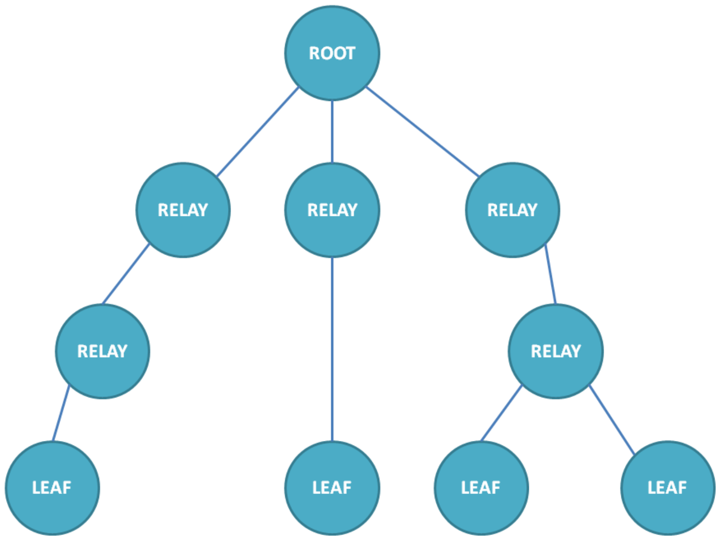


Figure 5.17: Computed optimal tree topology

the tree topology by using an XML representation, as the high level framework we use for the system administration panel natively provides methods to parse a *topology object* stored into a file. This feature allows to save and reuse an already computed topology in case the system is shut down and restarted. Of course, the overall status of the system could change while the system is down. For this reason before accepting a previously stored topology, a check is performed on the proposed topology. If no specific problems occur, the stored topology can be reused as it is, otherwise some reconfiguration operations are to be performed by the system administrator.

As we already mentioned, one of the most important feature our architecture provides is the ability of automatically react to both overload and over-provisioning conditions by migrating nodes or even entire subtree of the overlay network. The mechanism we decided to adopt in order to minimize the service downtime and consequently to maximize the quality of service perceived by end users is detailed in the following section.

5.4 Migration of SOLEIL nodes

Migrating a VM from a hypervisor to another can be a very difficult operation in the context of our complex infrastructure. This is especially true when the node to be migrated is contributing to delivering the service being on the path between the source node and a final client or even another VM. In such a situation, in fact, by migrating the VM the communication (and the service delivery) could be interrupted. The previous consideration let us understand how complex can be migrating a node in different use case scenarios, because of the several variables to be taken into the account. Nevertheless, the operational constraints we just presented can be relaxed if we assume the high level application can tolerate some packet loss and/or is able to implement some sort of ACK-based mechanism to ask for the retransmission of lost packets. This condition is widely satisfied as streaming protocols, although typically built on UDP, always take into account the possibility of losing some packets and, in some cases, allow to ask for retransmission. As

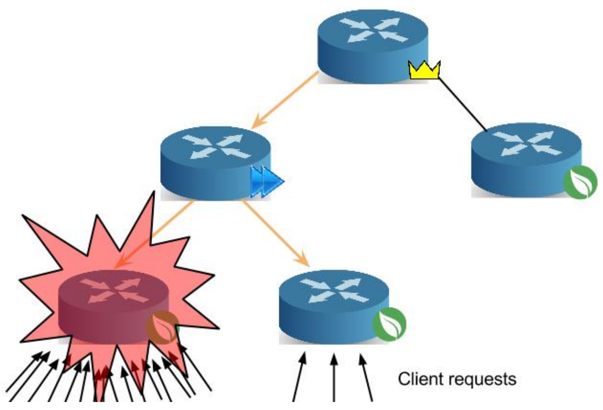


Figure 5.18: *Relay node* workload

a more strict constraint, we impose is to allow to migrate only leaf VMs. This constraint directly derives from intrinsic nature of the overall architecture design. In fact, if we think to the roles the different nodes of the tree play in the overall architecture we find out that the load on a generic relay VM is quite constant. Figure 5.18 help us to better understand this concept: in case the number of clients interested to a specific stream increases, the only VMs affected by this event would be the leaf VMs directly serving those requests.

This can be furthermore explained by the consideration that since the delivered resource is in our case represented by just a single stream, a generic relay VM has to handle just packets belonging to that single stream, no matter of the number of clients asking for it to the leaf node. Even in case we decided to add a multi-stream support, due to the architecture hierarchical design, it is very unlikely that a *relay VM* could be so much stressed to need to be migrated. In any case, even in such a circumstance, the problem could be avoided (or even prevented) by introducing a new tree layer when a certain load threshold is passed, avoiding in this way the overhead introduced by the migration process.

Intuitively, the side-effects introduced by the migration process, could be easily solved if we were able to move the VMs without disconnecting them by the network. This is the concept on the base of the so-called *LIve*

Migration of Ensembles (LIME) approach whose details are explained in [24]. This approach illustrates in detail what the best way to perform this operation is, by preliminarily *cloning* the network instead of just moving it. This approach leads to the best performances as the VM, once migrated, can start its services immediately, as packets are already flowing to the node's new location.

For this reason, when a node migration is needed, we adopt the following algorithm:

1. First of all the system checks whether or not the VM to be migrated is on the path is contributing to providing the stream to end users;

2. If this is the case, the system checks whether or not the destination switch which the VM has to be attached to is on the path of the same stream (i.e. the system checks if the new switch is already reached by packets belonging to the same media stream);

   In the first case (stream packets already reach the switch) nothing special is needed and the system is ready to migrate the node;

   In the other case, the system computes the path needed to reach the node like the VM were already been moved to the new location and installs flow rules on the corresponding switches in order to implement the route;

3. At this point the migration process can start.

It's worth noting that, during the migration process, packets will be replicated on the two paths (or cloned, if we want to use the term envisaged by the LIME approach). Once migration is successfully completed, the old path can be removed. Migrated VM will receive data right away it is waked up because the packets were already flowing to the new location during the migration process. This mechanism allows to minimize latency, downtime and packet loss.

Chapter 6

System implementation

The last period of the doctoral activity this document is related to has been devoted to the realization of a prototype implementation of the architecture we described in the previous sections. In this chapter we will hence present, motivating our choices, a set of components that helped us to deploy the platform on single testbed machine emulating a distributed network environment.

6.1 Virtualization environment and network emulation

In the previous chapters we introduced some virtualization solutions and briefly described their main features. We based our implementation of the virtualization environment on the Kernel-based Virtual Machine (KVM) framework and used the Quick Emulator (Qemu) for the network emulation component.

The main motivations behind the choice of KVM as the foundation of our implemantation can be found its an open source nature. Thanks to this reason the KVM framework is already present in almost any Linux distribution. Having a platform completely based on open-source components is indeed a very appealing feature for the academic nature of our project paving the way to the possibility to release the product of our work as an open-source platform as well. Furthermore, the fact that KVM it is natively included in many Linux distributions makes the deployment of our infrastructure on existing servers faster and easier. As a drawback we have to say that KVM exists for Linux hosts only but, standing the server side nature of the architecture we conceived, this can be smoothly considered a minor issue. A more technical reason that guided our choice can instead be related to the hardware requirements introduced by the migration process envisaged by our system in case critical situations occur. In this case, in fact, the node to migrate could be migrated. for example, from a host machine equipped with an Intel processor to a new node based on an AMD architecture. Such kind of issues can be easily coped with KVM provides an environment very robust to hardware heterogeneity and extremely tolerant when it comes to migration. Furthermore the module natively supports full hardware virtualization that, as we said in Section 5.2, is required to the guest to act as a *real* machine. KVM is also supported by OpenStack, our first-choice cloud platform for a real world working deployment of the SOLEIL system.

As KVM is just for the virtualization part, we used Qemu to emulate the network and storages components of our testbed. Qemu was chosen as it is the de facto user-level application for KVM. Figure 6.1 shows an overview of the KVM/Qemu architecture.

It is worth noting that, network emulation is needed to implement the overlay network we talked about in the previous chapter (we will present our OpenFlow-based implementation of such a network) while storage emulation is required to make available to multiple VMs a shared ISCSI disk as a local disk in order to facilitate and speed up the migration process.

Even though the couple KVM and Qemu seems to perfectly suit our requirements, we decided to furthermore abstract our implementation by the underlying technology by using the *Libvirt* component[9]. Libvirt, whose architecture is depicted in Figure 6.2, is neither a virtualization nor an emulation technology. It is a universal API developed by the RedHat Corporation that allows applications to interface with a lot of different virtualization solu-

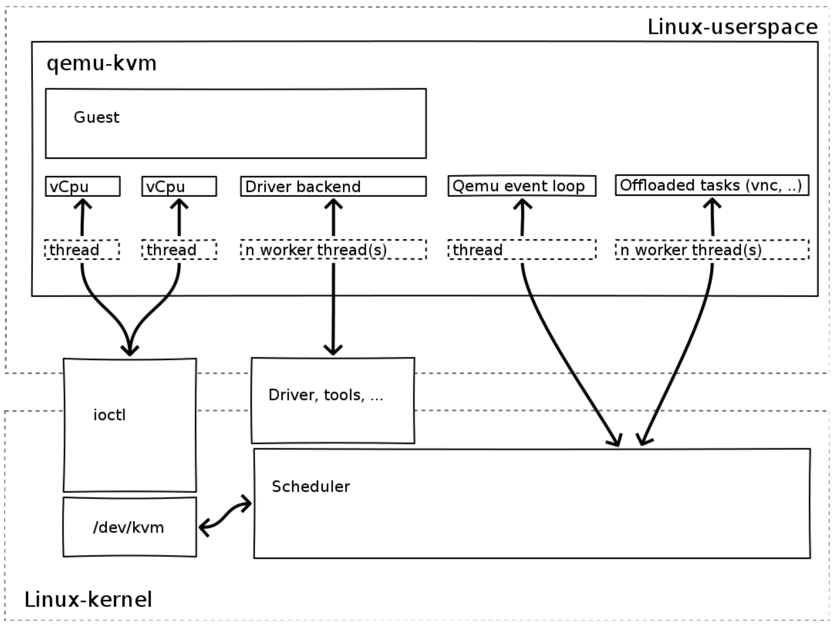


Figure 6.1: Virtualization environment overview: KVM and Qemu

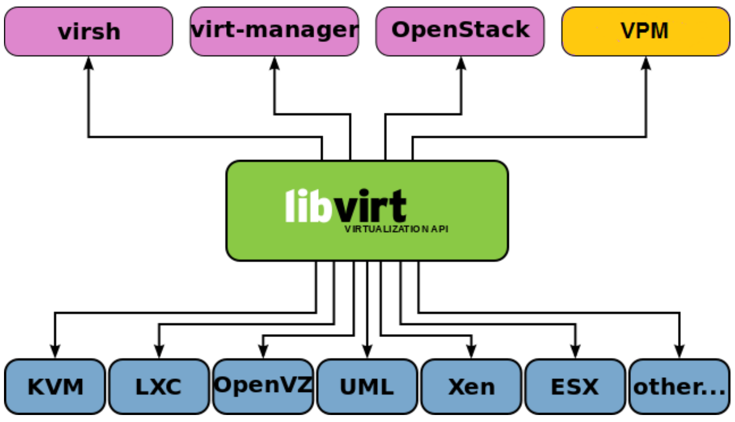


Figure 6.2: The Libvirt API

tions, also providing monitoring tools, to check both guests and hosts status. It is implemented as a daemon and, although its core is written in C, bindings exist for almost every programming language. The main reason behind the adoption of this further layer can be found in this very high-level API which provide a very flexible and easy to use access point to perform a variegated set of complex operations. Moreover, as we anticipated, should we ever need to change the underlying virtualization technology, we can reuse exactly the same application no needing to modify the code at all. Last but not least, Libvirt is also used by Openstack, so we can apply similar considerations as for KVM.

6.2 Disk sharing

As we anticipated, in order to be able to migrate a VM, a disk sharing technology is needed. We also identified ISCSI as the most suitable type of SAN and block-level disk-sharing technique. The first motivation leading us to choose this kind of technology is related to the better performances achievable by adopting a block-level technique. Furthermore, we didn't want to be limited to use a specific hardware infrastructure like required, for example, by the FiberChannel solution. Additionally, our infrastructure is designed to be run on the Internet and ISCSI is specifically designed for this purpose.

6.2.1 ISCSI components and SOLEIL setup

An ISCSI architecture, represented in Figure 6.3 is composed by two elements: (i) a target and (ii) an initiator. The target is a server holding real disk resources. These resources can be either existing disks or image files, or even complex disks structures such as RAID.

The target abstracts them as one or more disk resources and presents them to the outside world with an *ISCSI Qualified Name* (IQN). A IQN is a dotted separated string composed by the following elements:

- An `iqn` prefix;

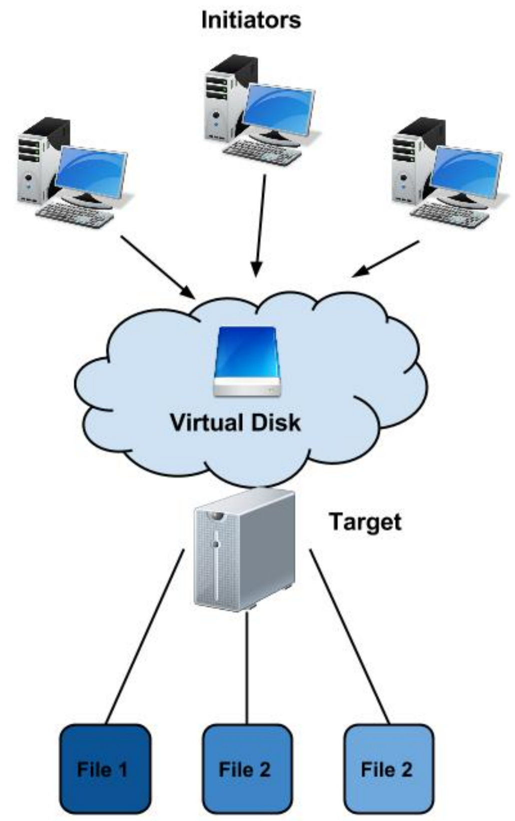


Figure 6.3: The ISCSI architecture

- a Date formatted according to the `yyyy-mm` notation and representing the year and month in which the authority took ownership of the domain;
- The reverse domain name of the authority.

The IQN can end with a `:targetName`, identifying a particular target. An example of IQN is:

```
iqn.2014-02.com.example:storage.disk2.sys1.xyz
```

the virtual storage can be accessed by using the IQN clients, called initiators. How the storage has to be seen by the outside is decided by the target itself. If multiple disks have to be abstracted as they were just a unique disk, a common configuration envisages to identify each single physical disk as a *Logical Unit Number* (LUN) of a unique virtual disk. This is the configuration we used for our implementation as well, where physical resources, in this case, are VM image files.

We then properly created the virtual HDs for the VMs we wanted to deploy on our testbed and then indicated them as LUNs of a specific ISCSI target. In order to allow the dynamic creation of a VM the virtual disks created have been over provisioned. In our testbed in fact, we deployed 5 VMs and 6 virtual disks so to be able to boot up a new VM if needed.

Once created the VMs we devoted our effort to the implementation of the SDN-enabled infrastructure. We relayed on the Openvswitch implementation of an Openflow switch.

6.3 Openvswitch

Openvswitch [40] is a software implementation of an OpenFlow switch and it is, at the moment of writing, the only one to seem mature enough to be used in a real project. It is released under the Apache 2.0 license (open source) and is available for all Linux distributions as a kernel module. In particular, from kernel versions 3.3+, it is already included as part of the kernel and

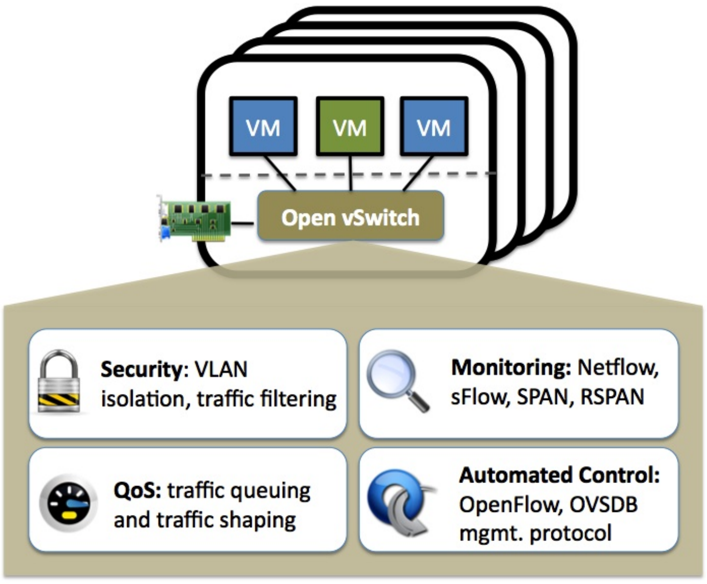

VM
VM
VM
Open vSwitch
Security: VLAN isolation, traffic filtering
Monitoring: Netflow, sFlow, SPAN, RSPAN
QoS: traffic queuing and traffic shaping
Automated Control: OpenFlow, OVSDB mgmt. protocol

Figure 6.4: Openvswitch overview

most of distributions provide user space utilities. Moreover, it is supported by KVM, thus allowing us to connect it to the overlay system's VMs without any additional workaround. Efforts are being devoted to port this same implementation on hardware chipsets so to obtain a dedicated OpenFlow device. This fact would grant not only higher speeds, but the integration with bare metal hypervisors that, as we stated in Section 2.2.2, run directly on top of existing hardware.

In Figure 6.4 are highlighted its features. It's worth noting that they all fit perfectly into our project making this component the ideal candidate for our prototype implementation. About security, Openvswitch provides extensions to be used along with TLS and it natively envisages VLAN support, which is a needed feature to keep the overlay and the real network isolated.

The monitoring aspect don't need further explanation: SOLEIL has to provide the administer the means to monitor the state of the network. Openvswitch supports multiple monitoring protocols, and as it is very adaptable, it also allows to export network monitoring data to external applications without any data conversion, in most cases. Finally, it supports OpenFlow, of course, but implements its own standard management protocol as well (Open vSwitch Database Management Protocol [6]). By management protocol we mean that Openvswitch provides the user a way to interact with its internal structure, thus allowing to create switches, ports and links between switches. This feature has nothing to do with OpenFlow, but it is needed as, according to the project requirements, we need to provide the system administrator a way to create networks by connecting switches.

6.3.1 Bridges and connections

As we explained in Section 5.2.1 a fundamental aspect of the overlay-physical network interconnection is related to the ability of configure the VMs to operate in the so called *bridged mode*. Openvswitch allows the creation of multiple bridge interfaces. In particular, in the Openvswitch syntax, a bridge is a virtual switch instance on which is possible to create new network interfaces or attach existing ones. Attachment points are called ports. Thanks to this feature an Openvswitch instance can have multiple bridges, and each one of them can have multiple ports. Bridges are normally isolated, but they can be connected by ports to form an extended network. In particular, connections between bridges can be local or remote:

- *Local connections* represent links between two bridges belonging to the same Openvswitch instance. Relative attachment points are also called *patch ports*;

- *Remote connections* join together remote switches. They are implemented with tunneling protocols, the user can choose between GRE and VLANx. Thanks to tunnels, it is possible to connect two LANs across the Internet, making their hosts communicate by using local IPs. Of course, it is necessary to ensure that local IPs respect the constraints imposed in LAN environments (i.e. same subnetwork, two hosts cannot have the same IP).

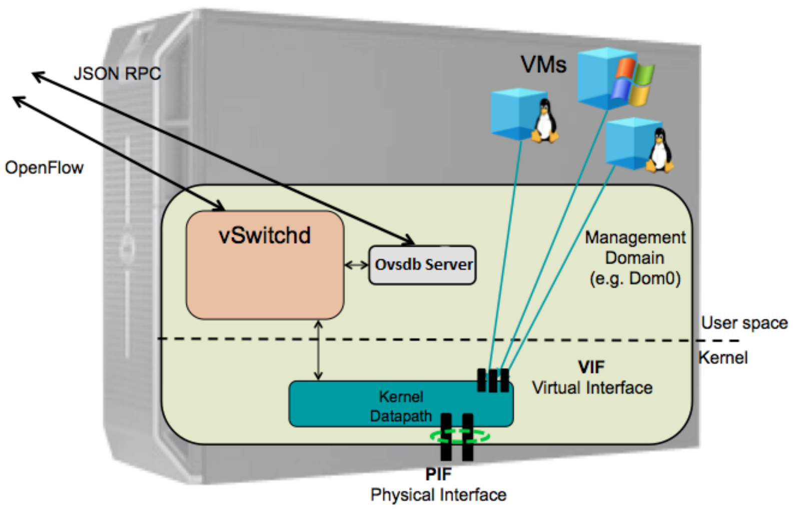


Figure 6.5: Openvswitch architecture

Remote connections have been massive used in our implementation in order to create the overlay network on top of the underlying SDN-enabled real network. In order to better explain our implementation of the network it is worth presenting the key components of the Openvswitch internal architecture .

6.3.2 Openvswitch architecture

The Figure 6.5 give us an high level perspective of the internal architecture of Openvswitch (further details can be found in [53]). The *Vswitchd* module highlighted in the figure is a daemon directly interfacing with the kernel module, taking instructions by user applications and communicating them to the module to actually be executed. It is also the element implementing the OpenFlow protocol and logic, directly talking to the controller. Openvswitch data (i.e., bridges, ports and connections) are stored in a database by using a json notation. The database is directly managed by a server called OvsdbServer. Users can directly interface to this server by using remote calls, according to the already mentioned JSON-RPC control protocol [6]. The Openvswitch bundle also provides user space scripts to perform these operations.

6.3.3 Openvswitch traffic patterns

Openvswitch rely upon the Linux network stack implementation. This means that even though the user thinks that Openvswitch is a bridge on its own, behind the scenes the kernel module just forwards Openvswitch traffic to the OS. [58] gives a good overview of all possible traffic patterns, so we will here just make present a summary of what is directly related to our infrastructure implementation. As we said, it is possible to attach real interfaces to an existing bridge or to create virtual interfaces as well (like in the case of a VM plugged on an existing bridge).

Imagine to have a host equipped with two NICs, `eth0` and `eth1`, and an Openvswitch bridge to which a VM is attached by means of a virtual interface `vnet0`. The bridge is also connected to another remote bridge via a GRE tunnel, whose port is called `gre0`. We will analyze two cases of interest. To simplify the example, we will suppose no controller presence so that Openvswitch acts as a simple *layer 2 switch*, thus broadcasting any packet it receives to all its active ports.

- Case 1, `eth0` is attached to the bridge. The Opevswitch is configured as in the following listing:

```
Bridge "br0"
Port "br0"
      Interface "br0"
            type: internal
Port "vnet0"
      Interface "vnet0"
Port "eth0"
      Interface "eth0"
Port "gre0"
      Interface "gre0"
            type: gre
            options: {remote_ip="192.168.1.100"}
```

In this case, the eth0 interface is isolated, because it belongs to the bridge. VMs can communicate to the external world, as they are attached to the bridge as well. Their packets are forwarded to eth0 (and to other interfaces as well). From the Linux host perspective this means that any sent packet will be passed to eth1, the only *visible* interface. In case the eth1 is not available, the host could not communicate with the external world. Such kind of problem can be solved by exploiting the internal interface whose is assigned the bridge name (i.e. br0. This interface, in fact, directly connects the host to the bridge and it is present in the network interface list as it was a real interface. By specifying the br0 interface as the default gateway, the host can inject its packets into the switch that, acting as a layer 2 one, would forward them to every interface, thus including eth0.

- Case 2, eth0 is not attached to the bridge. In this case the Openvswitch is configured as in the following:

```
Bridge "br0"
Port "br0"
      Interface "br0"
            type: internal
Port "vnet0"
      Interface "vnet0"
Port "gre0"
      Interface "gre0"
            type: gre
      options: {remote_ip="192.168.1.100"}
```

In this scenario, the host keeps to work as usual. VMs cannot interface to the external network because the eth0 interface is not attached to the bridge. However, a GRE tunnel port is present, so the traffic can flow at least to next switch. What is interesting is that GRE tunnel traffic implicitly flows through a NIC, specifically the one acting as gateway for packets directed to 192.168.1.x/24 network, the remote IP specified.

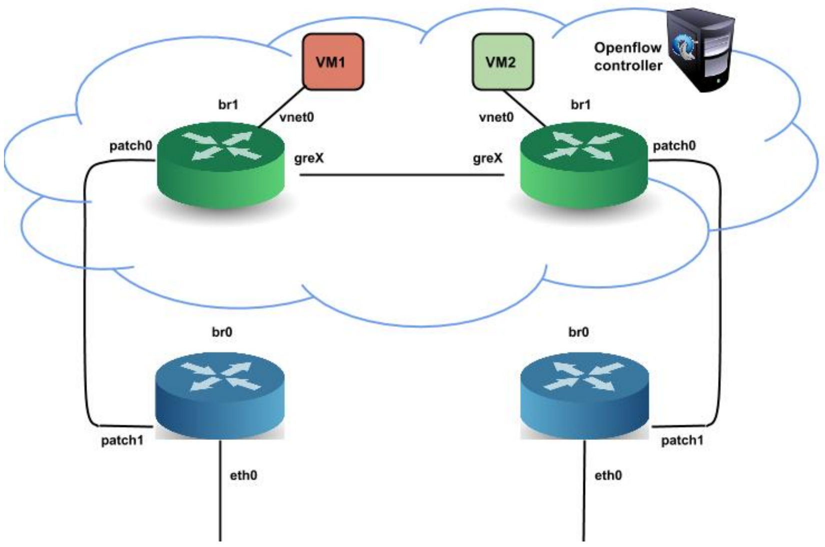


Figure 6.6: Openvswitch setup

6.3.4 SOLEIL Openvswitch setup

In order to implement the isolation between the overlay and the physical network introduced in Section 5.2.3, we relied on the Openvswitch capability of creating multiple bridges. Our setup, shown in Figure 6.6, is based on the second configuration example (case 2) we presented in the previous section. We also specified `br0` internal interface as default gateway in order to allow create the GRE tunnel enabling the communication with external network through the `eth0`, otherwise not reachable.

With reference to Figure 6.6, we can note that each node has two local bridges, `br0` and `br1`, connected by means of patch ports. While `br0` has a working NIC attached (`eth0`) and can exchange packets with the external network, `br1` is isolated, and the GRE tunnels are its only way to communicate with external components. Opting for this setup allow us to separate the traffic generated by our application from all the other traffic generated or directed to the VM. Bridges `br1` are the only ones belonging to the overlay network and, for this reason, they are also the only ones connected to the OpenFlow controller. In this way we make sure to examine and handle just packets regarding our application, drastically reducing the overhead. Patch ports will be used on the leaf nodes of the distribution tree, when a packet will need to jump from the overlay network to the physical network in order to be delivered to the end user. Just in this specific case, a special flow rule will be applied, allowing that packet to flow across the patch port, thus arriving to the bridge `br0` and then reaching the `eth0` interface. It's worth noting that links belonging to the overlay network, `vnet0` and GRE tunnels, are all marked as belonging to the same VLAN so that, in case the controller is not present in the network, virtual traffic will not affect the underlying physical network. There is no need to also mark patch ports, as this traffic will never reach them. This setup, commonly known as isolated bridge, is widely used in production environments.

6.4 The Controller element

As usual, we based our implementation of the Controller element envisaged by the SDN architecture by modifying a robust and flexible open source software, namely *Floodlight*

6.4.1 Floodlight

Floodlight [13] is an Apache licensed Java-based OpenFlow controller, supported by an active community of developers including a number of engineers from Big Switch Networks, one of the leader companies in the field of Software-Defined Networking . Although its open source nature, it was natively designed to work in production environments and to grant very high performance levels and to be able to interaction with both OpenFlow and some non Openflow enabled networks. Moreover, it is fully extensible thanks to a plugin based architecture. This specific feature paved the ground for the implementation of the controller role in the SOLEIL architecture.

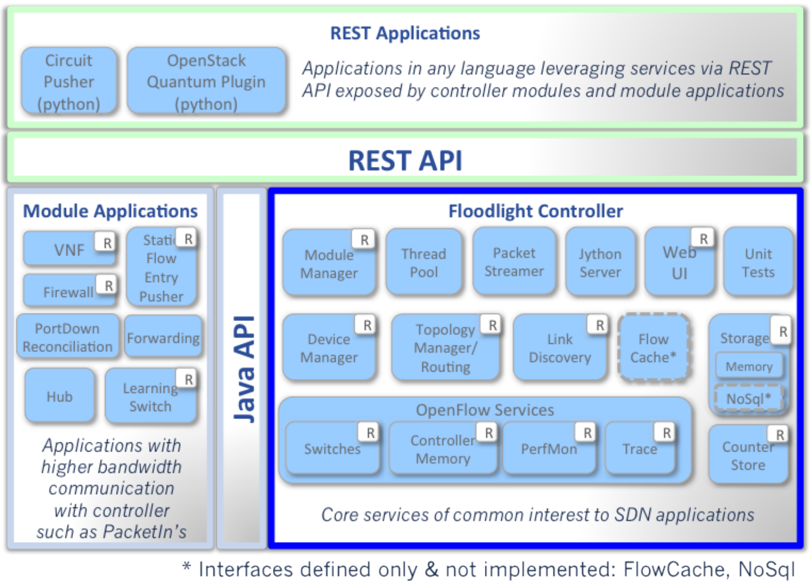


Figure 6.7: Floodlight architecture

Floodlight internal architecture

Figure 6.7 shows the modular architecture of the Floodlight implementation. For the sake of clarity, it implement both the OpenFlow controller role and a wide set of applications built on top of it. Applications can be distinguished in *module* and *REST-based* applications. The two kinds of elements differs mainly for their location in the overall architecture. In fact, while *module applications* reside inside the Floodlight controller core and they can make use of its plugin system, *REST-based* ones are external components and just communicate with the controller by means of REST calls. Module applications are more powerful, as they can make use of every Floodlight feature by directly interfacing with its components. Anyway, REST APIs are a way to easily interact with the controller and are preferred when no specific internal feature is required. Existing module applications, indeed, already provide basic functionalities like, for example, *flow pushing*.

6.4.2 SOLEIL controller setup

We implemented our own Floodlight module to perform the management operations needed to implement our application logic. We tried to limit them as much as possible, because putting too much logic into Floodlight would mean tying the overall system to this specific implementation. Nevertheless, in order to increase the overall system performance we opted to implement some classes of function exploiting the tools provided by Floodlight. The first one class is related to the exchange and handling of `ARP`, `DHCP` and `ICMP` packet. VMs need in fact to frequently exchange this kind of packets and so ad-hoc flow rules have to be installed in order the packets can reach their destination (it's worth remembering that, in presence of a controller, the network we configured is isolated). If we thought to create in a *proactive way* a new rule every time a VM is booted up it would be highly inefficient and it would lead to an unreasonable increase of the rule tables on the switches even in case no one of those packets is directed to the brand new VM. For this reason we implemented our plugin for simply waiting for these packets to be requested and reactively install flow rules to allow the proper communications, when needed. An idle timeout on the rule will allow to delete the rule in case of inactivity. The second class of traffic we decided to handle by means of the Floodlight plugin concerns the notification of network events, like for example the attachment/detachment of a VM or the fall of a link, to the management system. In fact, implementing a polling system directly on the management application, would have generated useless overhead and traffic on the network.

Our communication protocol makes use of the *Observer* design pattern and can be summarized as in the following steps:

- The management application asks the controller to be notified of events and communicates it the callback URL it wants to be contacted at for receiving events;
- Each time a significant event happens, the plugin sends an HTTP

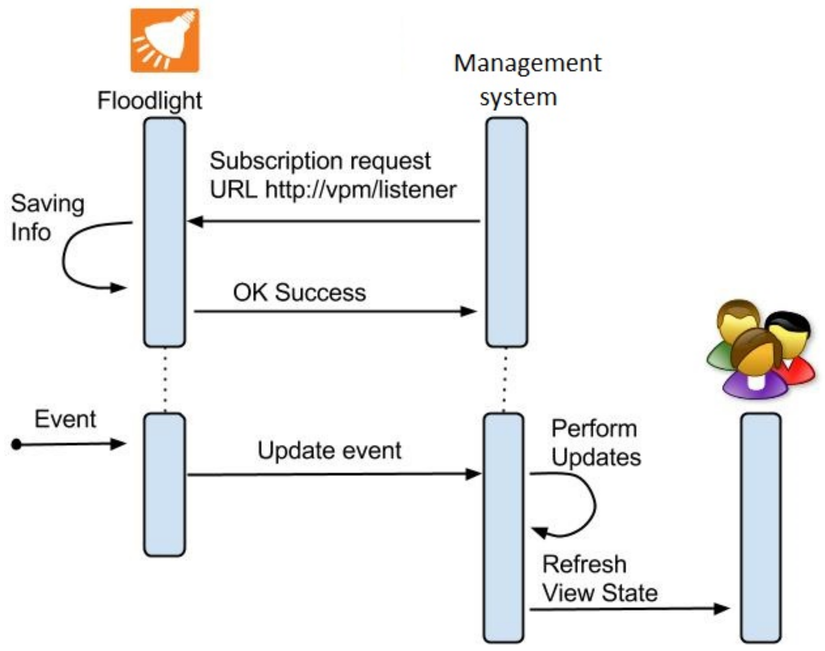


Figure 6.8: SOLEIL notification system

POST request to the specified URL containing data about that particular event.

This solution, schematized in Figure 6.8, decouples the management application from the plugin as the latter does not need to contain any static reference to the first.

Data is a JSON formatted text string containing a type indicating the element the event is related to (this field can assume one of the values in the set `switch`, `link` or `VM`), a subtype (`add`, `delete`) and specific details which vary according to the event class (for example, in the case of a switch, they will contain its IP and DPID).

In order to implement our module, we used the following Floodlight plugins:

- *TopologyListener* plugin, which implements an event system notifying about changes to the topology. We registered our module as a listener for this plugin;

- *TopologyManager* plugin, which provided means to find specific nodes or switches inside the topology, along with some embedded algorithms to compute the shortest path. We used these methods to compute paths between nodes, so to create proper flow rules to handle `ARP`, `DHCP` and `ICMP` packets and to install them on those specific switches.

6.5 Management application

We developed the component controlling the system as a Java application could in such way rely on several existing frameworks and libraries. Specifically, we implemented the management application of the SOLEIL system as a web application based on the Servlets paradigm and deployed it on the Apache Tomcat servlet container. Servlets are a very powerful mean to implement session and authentication mechanisms. Moreover, their security model has proven, through years, to be very solid. In fact, as each servlet runs inside a *Java Virtual Machine*, any crash, malicious or not, is contained inside of the VM sendbox, thus not affecting the underlying physical host.

6.5.1 Existing modules and APIs

In the following we will present the main Java APIs we adopted to realize our implementation specifying some custom improvement and modifications needed to fit the requirements of our architecture.

Libvirt 0.5 mod

As we already explained in 6.1, *Libvirt* provides bindings for almost each programming language. However, since the binding functionality are developed as separate branches of the main project it can happen they are not promptly updated once the corresponding technology evolves. The *Libvirt Java* binding had exactly this issue and in its current version don't implement some key monitoring function bindings. In addition, the code architecture resulted not optimized requiring to the developer to write an improper number of line of

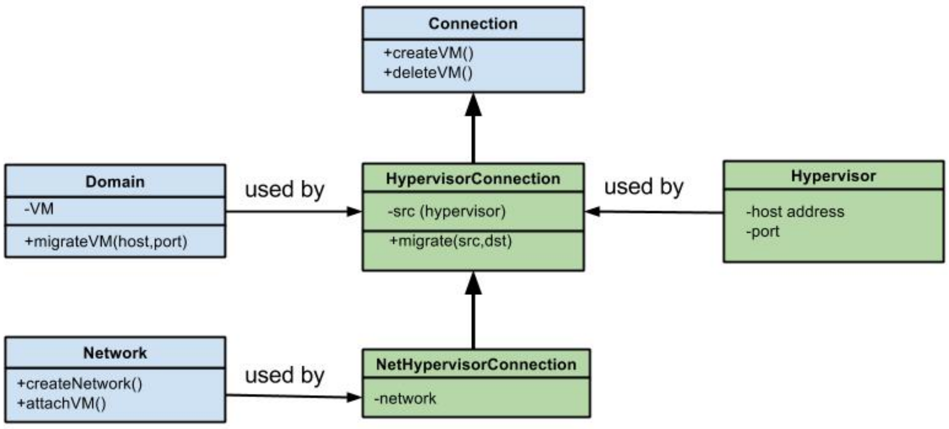


Figure 6.9: Example of extension of native Libvirt binding function

code even to perform very simple operations. Standing these considerations, we decided to customize the library introducing the missing bindings in the existing code and reorganizing the overall code structure exploiting the *inheritance* properties of the Java language to extend and enrich existing classes application logic. In this way, we can use new powerful classes while the old ones are still available, granting in this way backward compatibility with old applications. As en example in Figure 6.9 is shown a UML diagram of our `NetHypervisorConnection` and `HypervisorConnection` classes design and their relationship with the old existing classes.

As you can see, the original `Connection`, `Network` and `Domain` classes were kept as they were, but only to be either subclassed or used y our brand new implementation. In particular, the `HypervisorConnection` class represents a connection to a particular hypervisor and it inherits the methods to migrate a specific VM directly from the `Domain` class. `NetHypervisorConnection` is a further abstraction identifying a hypervisor connection whose VMs are bound to a particular network, in our case the overlay network. Starting or creating a VM with this class will automatically attach it to the aforementioned network, drastically reducing the code complexity.

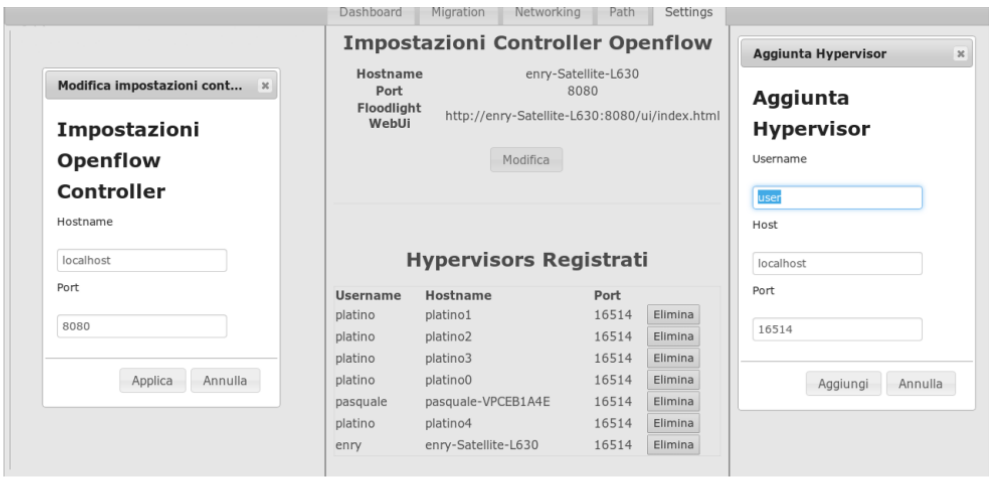
Opendaylight-Ovsdb library mod

In order for our management application to e able to create links between two Openvswitches, we needed a Java implementation of the Opevswitch Data Base (OVSDB) management protocol (see Section 6.3). We based our implementation on the one developed by the *Opendaylight* group. Opendaylight[31] is a collaborative SDN project providing, among others, a Java-based implementation of an OpenFlow controller coming whit a rich set of plugins and extensions. One of them, in particular, enable the interaction with a remote OVSDB. We still preferred Floodlight mainly because of its high performances and mature development status so we decided to start working on this plugin to create a stand-alone library version of it by removing all the dependencies from the controller part thus taking just the protocol implementation component. For the sake of completeness we have to say that, at the time of this writing, the *opendaylight-ovsdb group* is developing their own standalone version of the library.

6.5.2 System set up and management: top-down guided tour

As it should be clear at this point, the management application is the core of the whole system and, for this reason, it is very complex to clearly explain its structure. Nevertheless, as we already presented the most of the components and technologies underlying the front end provided to the administrator, our job will be much easier. In order to provide the reader with a description as close as possible to the administrator user experience, we will in the following present the application according to its GUI organization. The web interface is in fact divided in tabs associated to the different functionality it provides. The presentation of the application will then comply to this graphical organization and we will in the following introduce separately functionalities it offers along with some implementation details.

The user interface of the SOLEIL management application is a mix of HTML and JavaScript controls (based on the JQuery and JQuery UI li-



Modifica impostazioni cont...

Impostazioni Openflow Controller

Hostname: localhost

Port: 8080

Impostazioni Controller Openflow

Hostname	enry-Satellite-L630
Port	8080
Floodlight WebUi	http://enry-Satellite-L630:8080/ui/index.html

Hypervisors Registrati

Username	Hostname	Port	
platino	platino1	16514	Elimina
platino	platino2	16514	Elimina
platino	platino3	16514	Elimina
platino	platino0	16514	Elimina
pasquale	pasquale-VPCEB1A4E	16514	Elimina
platino	platino4	16514	Elimina
enry	enry-Satellite-L630	16514	Elimina

Aggiunta Hypervisor

Aggiunta Hypervisor

Username: user

Host: localhost

Port: 16514

Figure 6.10: SOLEIL management application: the settings tab

braries) while the backend, as we already mentioned, is implemented in Java, through the servlets approach. All the exchanged messages are formatted as JSON objects and embedded in the body of HTTP requests/responses.

Settings panel

The entry point of the application is represented in Figure 6.11.

This panel allows the administrator to add hypervisors and ISCSI targets to the SOLEIL database. Floodlight controller IP address and port can be specified as well. In the following we will refer to these configurations as runtime properties. Since their status can change while the system is running, the user can dynamically edit them accessing the Settings tab. As the underlying technology, we implemented the database exploiting *SQLite*, a software library implementing a server less, self-contained, zero-configuration database. We opted for this implementation because of its simplicity, as our tables, and interactions among them, are really essential and we didn't want to increase the overall system complexity by adopting more structured database solutions. Nevertheless, database details are wrapped by a `Database` class whose methods allow the abstraction of any interaction with the database itself. The actual communication with the database is re-

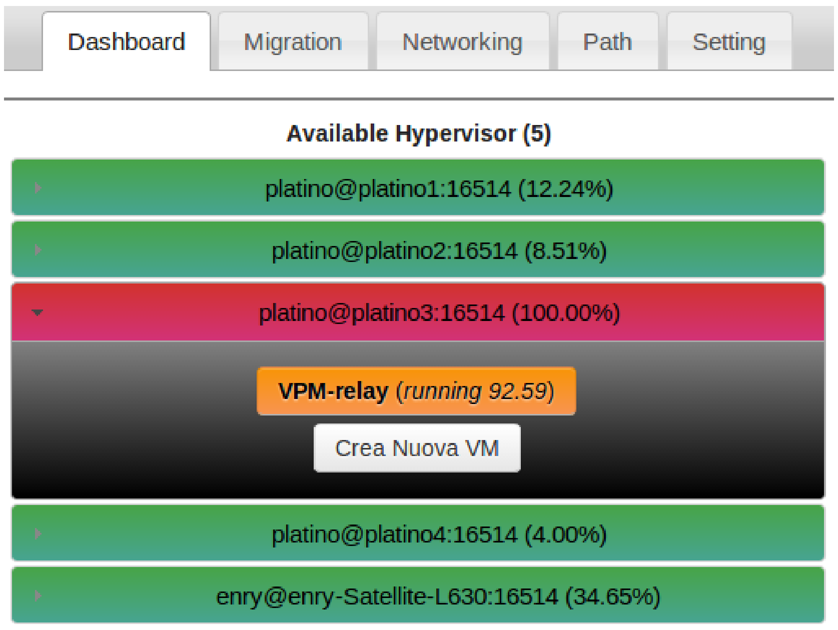

Dashboard
Migration
Networking
Path
Setting
Available Hypervisor (5)
platino@platino1:16514 (12.24%)
platino@platino2:16514 (8.51%)
platino@platino3:16514 (100.00%)
VPM-relay (running 92.59)
Crea Nuova VM
platino@platino4:16514 (4.00%)
enry@enry-Satellite-L630:16514 (34.65%)

Figure 6.11: SOLEIL management application: dashboard tab

alized by means of *Java DataBase Connectivity* (JDBC) [28], an API for the Java programming language that defines how a client may access a database and furthermore abstract the business logic by the underlying database. A `Database` class instance is initialized at the system startup and it is used as a singleton by the other class of the project. Some static properties of the system (e.g., threshold values used to decide if an hypervisor is overloaded) are stored in an external XML file loaded by the application during the start up phase and can be modified by the users before starting the system or even at runtime.

Dashboard panel

The Dashboard is aimed to give the user an overview of the network nodes. With reference to Figure 6.11 network nodes are represented just for what concerns the hypervisor part (a different tab is instead devoted to the network configuration of the system)

A list of registered hypervisors is shown, offline ones are marked gray while online ones can be marked with different colors according to their current CPU percentage usage, specifically:

- Green if CPU usage is $<$ `WARNING THRESHOLD`;
- Orange if `WARNING THRESHOLD` $<$ CPU usage $<$ `DANGER THRESHOLD`;
- Red if CPU usage is $>$ `DANGER THRESHOLD`.

Thresholds are set by the user as static property in the aforementioned XML property file. Ajax requests are periodically done to keep the view updated. By expanding a hypervisor row, the user gets a list of the virtual machines on that hypervisor and can start, stop and delete them. It is also possible to create a new VM, by specifying the ISCSI target to use as virtual HD instance. Note that VMs belonging to any application built on top of our system must be created by using this web interface. The creation process, in fact, will make sure to properly configure the virtual machine by plugging it on the overlay network and by assigning it an available virtual HD.

A *VPMConnectionManager* class instance (Figure 6.12) is shared by the existing servlets via *Servlet Context* and can be used to provide client with hypervisors related data. This class contains references to registered hypervisor connections and it is initialized by using data stored inside the SQLite database. It also registers for DB events by implementing a custom DatabaseListener interface. In such a way, every time a new hypervisor is added or removed to the database, the internal connection list can be updated. A background Task checks the state of the connections and notify the Connection Manager if something changes.

As the `HypervisorConnection` class directly inherits from the Libvirt `Connection` class, it has means to obtain the % CPU utilization of a Vm, so the GetHypervisors servlet, once obtained a list of active connections from the manager, can obtain this information by simply invoking the API. About VM management (create, start, delete, etc.) we use Libvirt methods as well to perform these operations, combined with our Database to keep the LUN

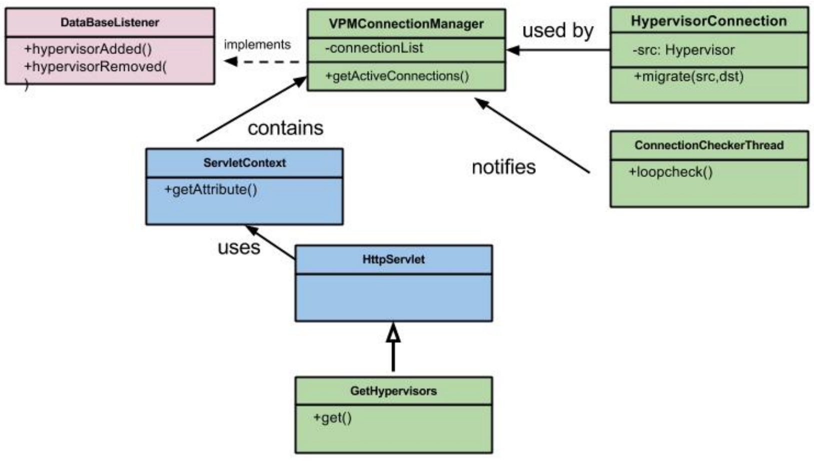


Figure 6.12: The Connection Manager component

allocation status for an ISCSI target consistent. Libvirt, in fact, cannot know if a generic virtual HD is used by a VM belonging to another hypervisor, as every hypervisor has a local view of allocated resources. So, each time a creation request is received, the database is queried to find an available LUN. If a LUN is available, the VM is created and a success message is sent back to the client. In the other case, client will be notified of the impossibility of creating the VM.

Network panel

As we anticipated in Section 5.3, the *network* tab allows the administrator to connect switches in order to create a tree network topology of the overlay network. The user can also specify a set of redundancy links to be used in case of crashes of network elements. In order to allow users to draw the topology in a fast and easy way, we made use of an external library called *mxGraph*, available as an academic and a professional licensed product. It consists of a JavaScript client-side library allowing topology drawing inside web browser, and a server-side component designed to receive and process the graphs sent by the client. In Section 5.3 we explained our delivery network is based on the concept of role-based tree. This means the user has to explicitly assign to every node a specific role. As we can see in Figure 6.13, the network tab

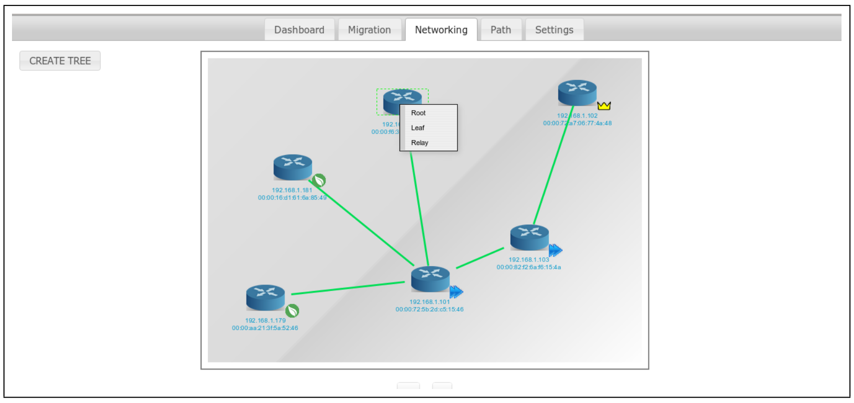


Figure 6.13: SOLEIL management application: network tab

show every switch currently attached to the controller. This elements can be used by the administrator to create the tree topology which will allow the service delivery to end users. I's worth noting that, being a node both a hypervisor and a switch, the graphical elements in the figures represent hypervisors as well and in fact, by hovering the mouse on a generic device, a list of installed VMs will appear.

In order to create a connection between nodes, the administrator can simply drag the mouse from a switch to another, while to assign the role to one of the nodes he can right click the node and chose the proper value by a context menu. We decided to let the user draw a generic topology, be it a tree or not, anyway when the `send tree` button is pressed the server side algorithm presented in Section 5.3 will take care of founding the minimum topology respecting the roles assigned by the administrator. Unused links will be identified and saved as redundancy connections to be used in case of problems. About the network topology representation, we decided to store it in an XML file, as *mxGraph* already provides methods to parse a topology object reading it by this kind of file.

6.5.3 Path tab and plugin system

Once the topology has been created, the user can establish a path connecting the root and the leaf nodes by means of the tool provided by the *path tab*

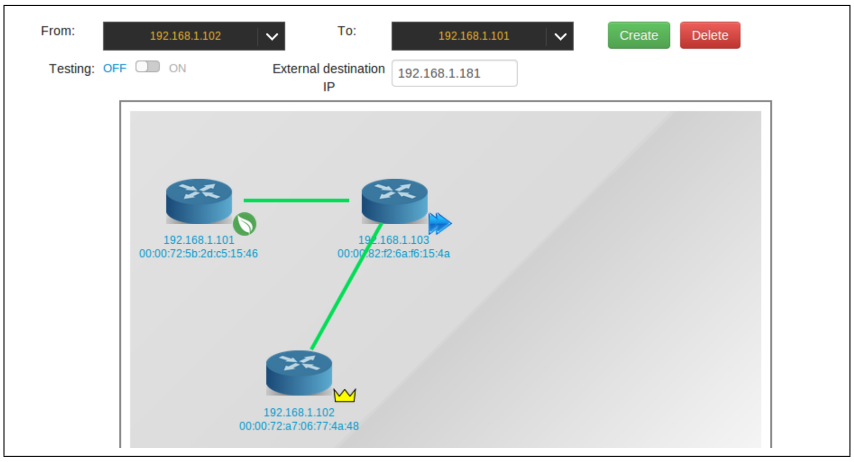


Figure 6.14: SOLEIL management application: path tab

(Figure 6.14).

Selecting a root node, the user is allowed to chose a destination leaf. Any possible path connecting the two nodes will be shown on the topology. In order to create a path the user is also required to chose an external IP address corresponding to the entry point from the physical network on the leaf node. This field is required in order to make the packets belonging to the flow to *jump* from the overlay to the physical network.

On the back-end side, a `DefaultVPMPathManager` class instance is shared among servlets instances. This class holds information about existing paths and application-related flow rules currently installed on them. By application-related we mean that, as we explained, the flows related to DHCP, ARP and ICMP protocols will not be included in this category. When a new path is asked to be built, the servlet directly contacts the `DefaultVPMPathManager`, which will install specific flow rules on selected switches along the path (or return any existing one). The rules are composed by a *basic* and a *specific* part.

The *basic* part implements the behavior we presented in Section 5.2.3:

- If the switch is a root node or a a plain relay switch (namely not having any VM on it) the packets are sent to the next switch on the path;

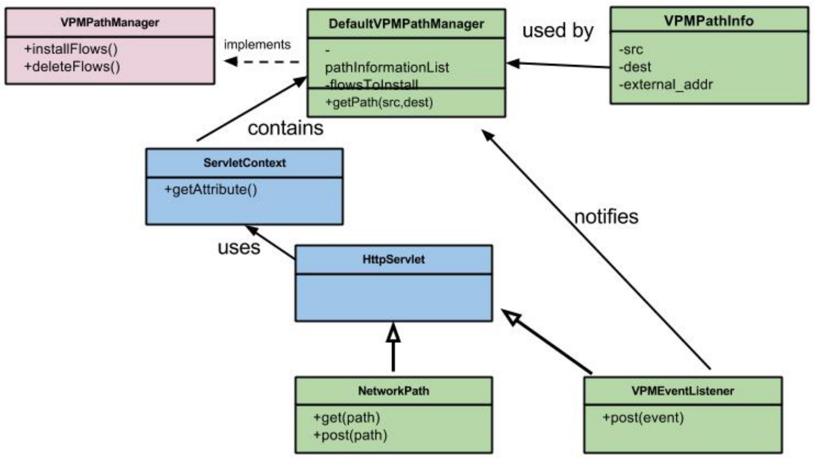


Figure 6.15: Path Manager diagram

- If the switch is a relay with a VM on it, two flow rules will be installed

    to forward this packet to the VM

    to forward any packet from the VM to the next hop

These rules will thus allow to pass packets to relay VMs where the application level will use the packet to implement the high level business logic.

The *specific* part is taken by an external JSON file during the initialization phase. This file, configurable by the user, contains specific matches for the flow rules, like source IP address, source MAC address, TCP source port and so on. By just using the basic part, indeed, it is not possible to fully identify a generic flow, so we need to add the specific matching rule to properly forward different flows. The name we assigned to the class in charge of implementing this logic (whose diagram is depicted in Figure 6.15), namely `DefaultVPMPathManager`, reflect the fact that this is our own implementation of the `VPMPathManager` interface.

We chose this design pattern to allow the creation of different `PathManager` class siuted for other use cases. In our designed scenario, in fact, a generic resource such as a live stream has to flow from a root node to a leaf node, elaborated by any VM present on a switch. There could be applications in which this behavior is not the best or needs to be made more complex. To

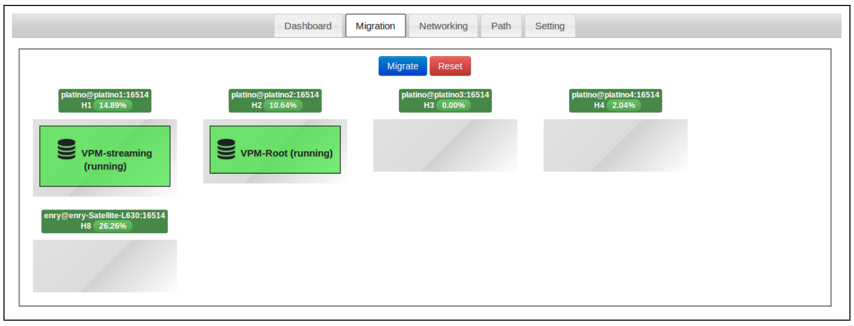


Figure 6.16: SOLEIL management application: migration tab

address these kind of scenarios, we adopted this full customizable design pattern. Specifically, a `path manager instance` field contained in the XML property file, allows the users to specify their the implementation class they want to be used. Furthermore, a `VPMEventListener` servlet is designed to accept POST requests from the SDN controller module. These requests, as we already explained, contain updates about the network and VM status and can be forwarded the `DefaultVPMPathManager` in order to act on the system by modifying flow rules or even invalidate them. For example, if a VM is shut down the flow rule on that switch would have to change to make packets directly jump to the next hop, to avoid to send packets to a *dead* VM.

Migration Tab

This is the section of the administration panel reserved to the migration of system's nodes. The frontend panel is depicted in Figure 6.16.

Since we imposed that only leaf nodes were allowed to be migrated, this panel shows only this kind of nodes. In order to schedule a migration operation, the user can simply drag and drop the desired VM from the source to the destination hypervisor. This simple operation is converted in a complex set of server side operations, as we explained in Section 5.4. Libvirt APIs are used to perform migration process itself, while the `PathManager` is in charge of implementing the network cloning algorithm inspired to the LIME migration approach detailed in Section 5.4. With reference to Figure 6.17,

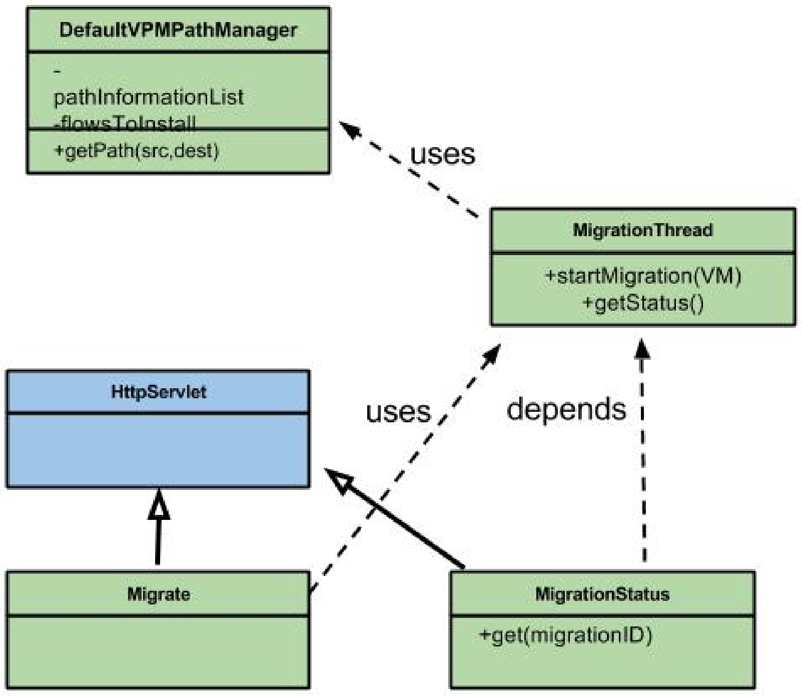


Figure 6.17: Migration process class diagram

we can point out that a specific servlet (`GetMigrationProgress`) can be periodically polled by the client side to obtain updated information about the status of a specific migration operation.

Chapter 7

Functional testing

In this section we will present the result of a functional testing campaign involving both the streaming application and the real-time content distribution system. After presenting the testing environment, we will describe the demo scenario and we present an overview of the final results. It's worth clarify that, since the system is still in an prototype state and the available testbed is composed by a single micro server (see details below), we decided to conduct only functional tests. Quantitative analysis, involving throughput and latency measurements, will be the objective of our future work.

7.1 Testing environment and testbed setup

The testbed we realized for the testing campaign objective of this section is realized using a portable *HP Proliant N54L* $708245 - 425$ *micro server* with the following configuration:

- CPU: AMD Turion II Neo N54L dual core processor
- RAM: 8 GB
- Network: 2 network interface cards
- Hard disk: 2 HDs, 80 and 160 GB respectively

In order to achieve the best performances, we installed a bare metal hypervisor, *VMWare vSphere ESXi 5.5*, to simulate physical hosts through

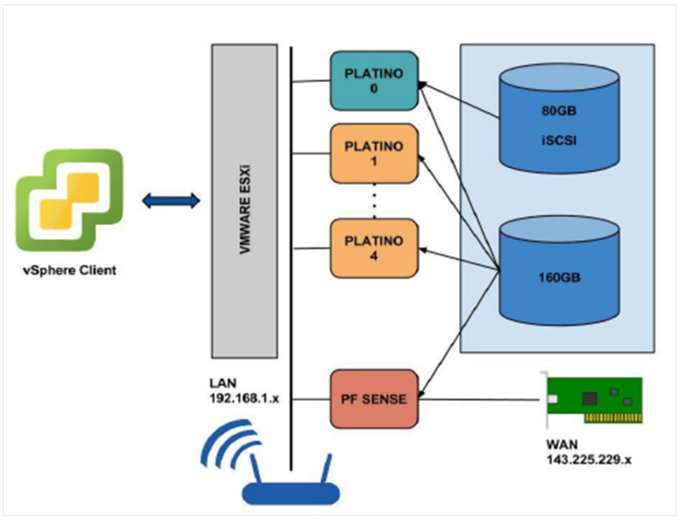


Figure 7.1: Testbed overview

VMs. Not providing any local user interface, this hypervisor can only be remote controlled by its dedicated client software. This client-server architecture, which is common to most *type I hypervisors*, is necessary to make the virtualization as more efficient as possible.

Figure 7.1 shows an overview of our testbed we realized.

We set up the two hard disks to be shared by the virtual machines according to the following configuration:

- 160 GB are shared across VMs and used to simulate their virtual hard disks;
- 80 GB are allocated to a specific VM with a raw device mapping technique. As this VM will represent our ISCSI target, it needs dedicated storage.

We deployed five virtual machines (named `platino0` - `platino4`) with the following configuration:

- 1 GB RAM
- 20 GB Hard Disk

The `platino0` machine is designed to be an ISCSI target and it mounts a dedicated 160 GB HD as an external peripheral.

The testbed includes an additional VM mounting an open source firewall (*pfSense*), which provides NAT and DHCP functions as well. This VM creates a 192.168.1.x/24 LAN while providing external access through its physical WAN interface. A second NIC, not displayed in figure, is also allocated to *pfSense*. This NIC is connected to an external router in order to become part of the virtualized LAN as well, along with any physical host currently attached to it.

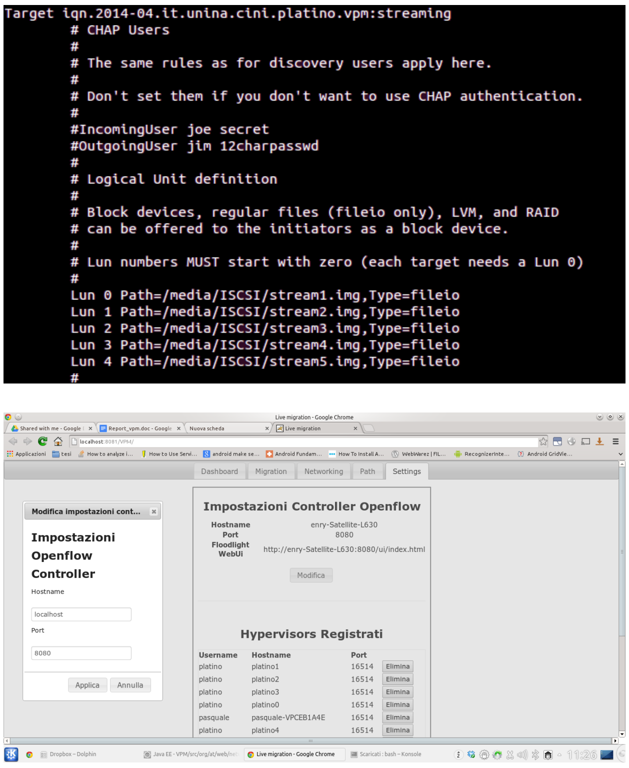
Thanks to this configuration we are able to implement complex scenarios involving multiple external devices as clients of the emulated server infrastructure.

7.1.1 Setup of system's nodes

Each virtualized node is a VMs equipped with an Ubuntu server 13.10 (Saucy Salamander) operating system. This distribution has minimum requirements and already come with all the software we need in our testing scenarios. Two bridges, `br0` and `br1`, are set up according to the description presented in Section 6.3.3. In order to create the overlay network in the context of the Qemu instance, the following XML configuration is passed to the *libvirt API*:

```
<network>
      <name>test-network</name>
      <forward mode="bridge"/>
      <bridge name"br1"/>
      <virtualport type="openvswitch"/>
      <portgroup name="vlan-00" default="yes">
            <vlan>
                  <tag id="2"/>
            </vlan>
      </portgroup>
</network>
```

This step is automatically passed to the *libvirt API* by our management application every time a new VM is created and booted up.



```
Target iqn.2014-04.it.unina.cini.platino.vpm:streaming
        # CHAP Users
        #
        # The same rules as for discovery users apply here.
        #
        # Don't set them if you don't want to use CHAP authentication.
        #
        #IncomingUser joe secret
        #OutgoingUser jim 12charpasswd
        #
        # Logical Unit definition
        #
        # Block devices, regular files (fileio only), LVM, and RAID
        # can be offered to the initiators as a block device.
        #
        # Lun numbers MUST start with zero (each target needs a Lun 0)
        #
        Lun 0 Path=/media/ISCSI/stream1.img,Type=fileio
        Lun 1 Path=/media/ISCSI/stream2.img,Type=fileio
        Lun 2 Path=/media/ISCSI/stream3.img,Type=fileio
        Lun 3 Path=/media/ISCSI/stream4.img,Type=fileio
        Lun 4 Path=/media/ISCSI/stream5.img,Type=fileio
        #
```

Modifica impostazioni cont...

Impostazioni Openflow Controller

Hostname: localhost

Port: 8080

Impostazioni Controller Openflow

Hostname	enry-Satellite-L630
Port	8080
Floodlight WebUi	http://enry-Satellite-L630:8080/ui/index.html

Hypervisors Registrati

Username	Hostname	Port	
platino	platino1	16514	Elimina
platino	platino2	16514	Elimina
platino	platino3	16514	Elimina
platino	platino0	16514	Elimina
pasquale	pasquale-VPCEB1A4E	16514	Elimina
platino	platino4	16514	Elimina

Figure 7.2: ISCSI Target and Initiator

'test-network'is the name we assigned to our network, `br1` is the overlay network bridge and the `id=2` is the used to identify the VLAN used to connect the different components of the overlay network.

Each node is thus configured to mount the ISCSI target situated on `platino0`, so 5 LUNs, each one representing an already configured VM, will be available (as shown in Figure 7.2).

7.1.2 Guest setup

This is an extract of the XML file that has been used as the template for creating new virtual machines:

```
<domain type="kvm">
<name>VPM-Streaming</name>
<uuid>2476b3d2-4ccf-4ca9-46d1-f6610ff55529</uuid>
<memory unit="KiB">524288</memory>
```

```
...
<devices>
      <emulator>/usr/bin/qemu-system-x86_64</emulator>
      <disk type="block" device="disk">
            <driver name="qemu" type="raw" cache="none
               "/>
            <source dev="/dev/disk/by-path/ip
               -192.168.1.100:3260-iscsi-iqn.2014-04.
               it.comics.vpm:streaming-lun-4"/>
            <target dev="vda" bus="virtio"/>
            <address type="pci" domain="0x0000" bus="0
               x00" slot="0x05" function="0x0"/>
      </disk>
      <interface type="network">
            <mac address="52:54:00:d3:4c:c2"/>
            <source network="ovs-network" portgroup="
               vlan-stream"/>
            <address type="pci" domain="0x0000" bus="0
               x00" slot="0x03" function="0x0"/>
      </interface>
...
</devices>
</domain>
```

Part of this file is automatically configured by system when a new VM is created. For example, the UUID is randomly chosen and the network attribute is added each time the virtual machine is booted up as well.

The VMs come with Ubuntu server 13.10 OS installed as well. As we explained in Section 3 the streaming application is built on top of the Gstreamer technology. In our specific use case, the application is used to create a live stream of a sample `.mov` video file. Two bash scripts have been created for the purpose, the first one to be used on the streaming VM:

```
#!/bin/sh
#
FILE=$1
DESTINATION=$2
PORT=$3
gst-launch-1.0 -v filesrc location=$FILE \ ! decodebin
    ! x264enc ! rtph264pay ! udpsink host=$DESTINATION
```

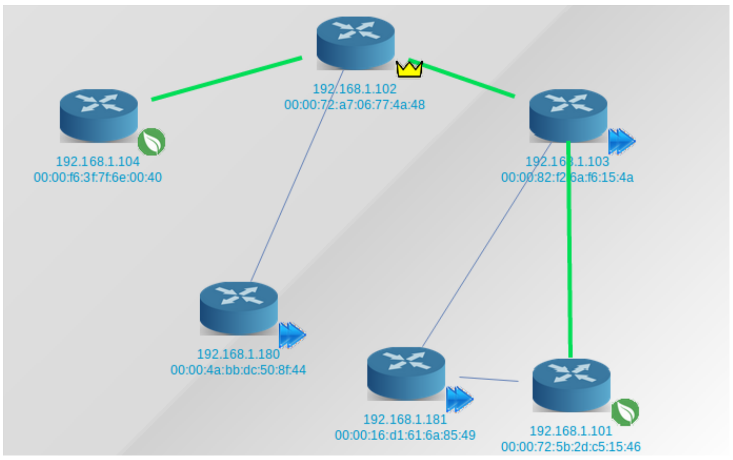


Figure 7.3: Creation of the network topology

```
    \ port=$PORT sync=false
```

the second one installed on both internal and leaf nodes:

```
#!/bin/sh
#
PORT_IN=$1
DESTINATION=$2
PORT_OUT=$3
VIDEO_CAPS="application/x-rtp, media=(string)video,
   clock-rate=(int)90000, encoding-name=(string)H264"
RELAY_SINK="udpsink host=$DESTINATION port=$PORT_OUT
ts-offset=0 name=vrtpsink sync=false"
LATENCY=200
gst-launch-1.0 -v udpsrc caps=$VIDEO_CAPS port=
   $PORT_IN ! $RELAY_SINK
```

The latter script is used on the leaf VMs too, as they act as relays between the overlay network and the physical network.

7.1.3 Network Setup

By using the management application web GUI, we configured the topology shown in Figure 7.3.

With reference to the figure, it is possible to note we defined some redundancy links (represented by the blue lines). Furthermore, we assigned roles to the nodes. Two leaves are specified, one directly connected to the root node (`platino2`), the other connected via a relay node. Being `platino2` the root node of the tree, as we explained in it is mandatory to install on it a DHCP server. We used the Ubuntu default DHCP daemon (`dhcpd`), with the following network specification:

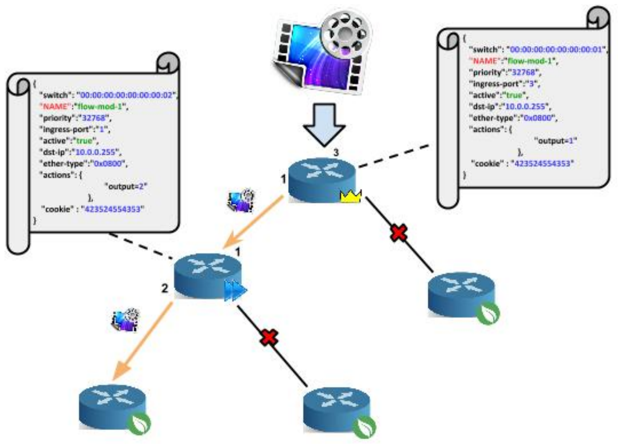
```
subnet 10.0.0.0 netmask 255.255.255.0 {
      range 10.0.0.2 10.0.0.254;
}
```

This server sits on the br1 internal interface, whose static address is 10.0.0.1 and is marked by VLAN with `tag=2`. In this way, it acts as a default DHCP server for our overlay network assigning addresses belonging to the 10.0.0.x/24 subnet. The VLAN tag assures it will not be any conflict with an existing server running to the physical network.

7.2 Sink or Swim selective forwarding technique

In order to test the several mechanisms and components involved in the overall complex asrchitecture, we decided to exploit the streaming application to create an audio video flow to send in broadcast to the distributed delivery network (i.e. by using the 10.0.0.255 as destination address). Under normal circumstances, this would imply the flooding of the entire network and would make the relay nodes just useless steps on the root-destination path, because the stream would just naturally flow to every switch ignoring any halfway entity. For our testing purposes this is just ideal, because without explicitly applying flow rules on the network components we expect these packets would not be delivered at all. In fact, what we the system implements is a logic that we called *sink* or *swim*, which can be described by the following definitions:

- *Sink.* In case no flow rules associated to a stream is present in the flow table of a switch, the corresponding packets will be dropped and the flow will be stopped at this hop;



```
{
  "switch": "00:00:00:00:00:00:00:02",
  "NAME":"flow-mod-1",
  "priority":"32768",
  "ingress-port":"1",
  "active":"true",
  "dst-ip":"10.0.0.255",
  "ether-type":"0x0800",
  "actions": {
                "output=2"
          },
  "cookie" : "423524554353"
}
```

```
{
  "switch": "00:00:00:00:00:00:00:01",
  "NAME":"flow-mod-1",
  "priority":"32768",
  "ingress-port":"3",
  "active":"true",
  "dst-ip":"10.0.0.255",
  "ether-type":"0x0800",
  "actions": {
                "output=1"
          },
  "cookie" : "423524554353"
}
```

Figure 7.4: Sink and swim technique example

- *Swim*. In case a flow rule is present the corresponding output action will be applied letting the flow travel only across the proper interfaces (even though the packets contain a broadcast destination address).

In Figure 7.4 is shown a graphical representation of the aforementioned selective forwarding technique.

Once realized the aforementioned testbed setup and applied the configurations exposed in the previous sections, we could start the actual testing phase.

7.3 Testing flow

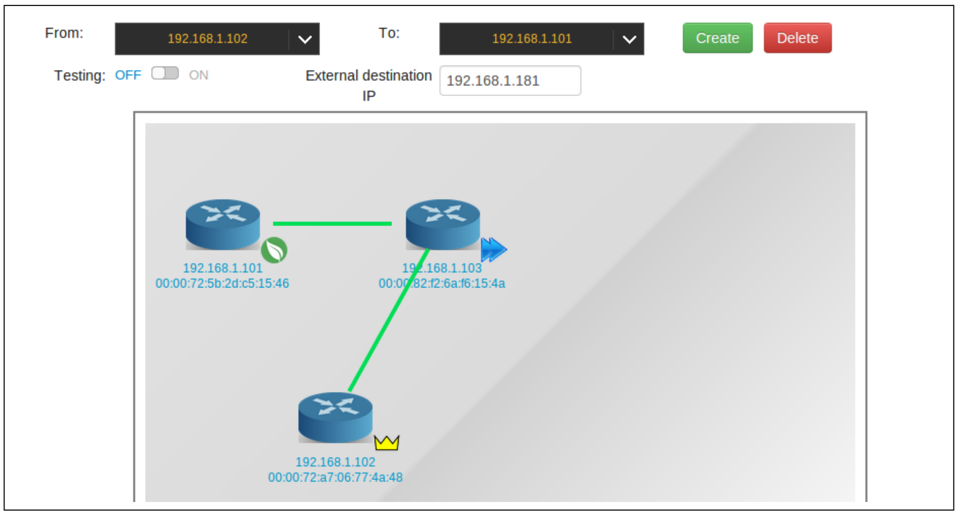
First of all, we booted up the virtual machines and made sure they had been assigned an IP address letting them to be part of the overlay network. We tested the reachability of the machines by means of `ICMP ping` messages. The success of this test also confirmed us that our Floodlight module were working correctly, as automatic reactive rules for DHCP, ARP and ICMP were successfully installed on the network. As for the proactive rules, in the setup phase we needed to specify our custom `extended` part as a JSON file:

```
"vpm-extend-match": {
        "dataLayerType": "0x0800",
        "networkTypeOfService": 0,
        "transportDestination": 0,
        "dataLayerVirtualLanPriorityCodePoint": 0,
        "transportSource": 0,
        "networkSource": "0.0.0.0",
        "dataLayerDestination": "00:00:00:00:00:00",
        "networkSourceMaskLen": 0,
        "dataLayerVirtualLan": -1, "networkDestination":
            "10.0.0.255",
        "networkProtocol": 0,
        "networkDestinationMaskLen": 32,
        "dataLayerSource": "00:00:00:00:00:00",
}
```

As our scenario envisages a very simple application, we had to simply customize the destination IP address (i.e. the 10.0.0.255 broadcast address) and the `ether-type` field (0x800 = IPv4), relying on default values for all the other fields. Such a king of configuration allows to capture every broadcast IPv4 packet, in other words all the packets sent by our streaming application. As a first testing scenario we decided to boot only VMs installed on the root and leaf nodes of the topology. This means that relay nodes act as a 'pass-through'elements, by just sending packets to the next hop in the path according to the mechanism described in Section 5.2.3. We hence started a stream and, after verifying that no external user could receive it, we activated a path toward a leaf node, specifying 192.168.1.181 as external destination IP address (see Figure 7.5).

This is an extract of the pass-through rule automatically installed on the relay switches:

```
"data": [
        ...
        {
        ...
        "type": "FLOW_MOD",
        "hardTimeout": 0,
        "lengthU": 80,
```


From: 192.168.1.102 To: 192.168.1.101 Create Delete

Testing: OFF ON External destination IP 192.168.1.181

Figure 7.5: First testing scenario involving only root and leaf nodes

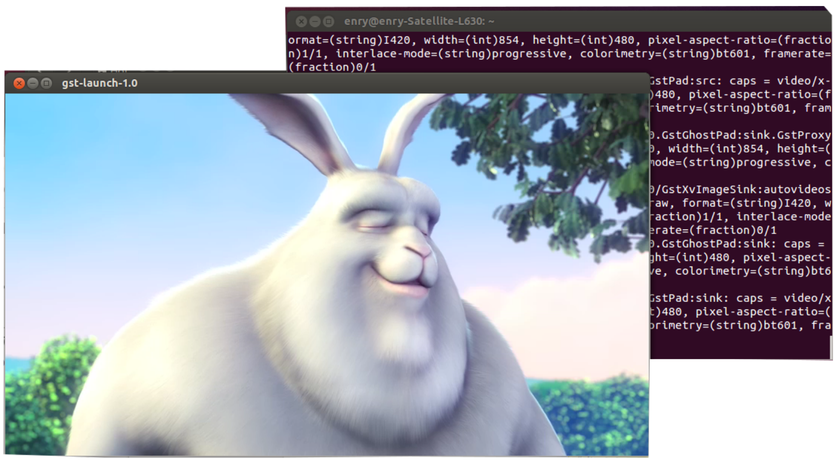
```
"version": 1,
"cookie": 45035996524483576,
"priority": 100,
"name": "RTP_PASSTHROUGH000082f26af6154a",
"idleTimeout": 0,
"length": 80,
"command": 0,
"match": {
      "wildcards": 3162095,
      "dataLayerType": "0x0800",
      ...
      "networkDestination": "10.0.0.255",
      "inputPort": 0
},
"actions": [
      {
      "port": 13,
      "maxLength": 32767,
      "length": 8,
      "type": "OUTPUT",
      "lengthU": 8
      }
],
...
```

```
}
```

It's quite clear that this configuration simply recall the aforementioned `extended` one, the variable network related information (e.g., the port numbers) are inserted automatically by the system.

Since a leaf node hosts a VM too, we had to apply two different rules: (i) the first for sending packets to the VM and (ii) the other for forwarding them from the VM to the next hop (i.e. the real network).

```
"data": [
       ...
       {
       ...
       "type": "FLOW_MOD",
       "priority": 100,
       "name": "RTP_TOVNETS000082f26af6154a",
       ...
       ,
       "actions": [
              { "port": 3,
              "maxLength": 32767,
              "length": 8, "type": "OUTPUT",
              "lengthU": 8
              }
       ],
       {
       ...
       "type": "FLOW_MOD",
       "name": "RTP_FROMVNET7_0000725b2dc51546", "
          ingress-port": "3",
       ...
       ,
       "actions": [
              {
              "networkAddress": "192.168.1.181",
              "length": 8,
              "type": "SET_NW_DST",
              "lengthU": 8
              },
```



```
ormat=(string)I420, width=(int)854, height=(int)480, pixel-aspect-ratio=(fractio
n)1/1, interlace-mode=(string)progressive, colorimetry=(string)bt601, framerate=
(fraction)0/1
GstPad:src: caps = video/x-
)480, pixel-aspect-ratio=(f
rimetry=(string)bt601, fram

0.GstGhostPad:sink.GstProxy
0, width=(int)854, height=(
mode=(string)progressive, c

0/GstXvImageSink:autovideos
raw, format=(string)I420, w
raction)1/1, interlace-mode
erate=(fraction)0/1
0.GstGhostPad:sink: caps =
ght=(int)480, pixel-aspect-
ve, colorimetry=(string)bt6

GstPad:sink: caps = video/x
t)480, pixel-aspect-ratio=(
orimetry=(string)bt601, fra
```

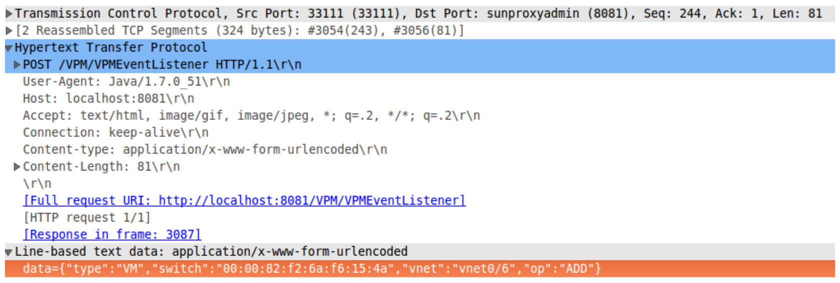
Figure 7.6: Simulated live stream received by the destination

```
                {
                "port": 4,
                "maxLength": 32767,
                "length": 8,
                "type": "OUTPUT",
                "lengthU": 8
                }
        ],
        }...
}]
```

It's worth noting how the *ingress-port* field of the second rule corresponds to the `output` field of the first rule action set, representing the port that the VM is attached to. Furthermore, it is important to highlight the `action` field in the last flow rule, used to impose the rewriting of the destination IP address using the 192.168.1.181 value before sending the packet to the physical network.

Adopting this configuration, we can verify that the stream is correctly received by the host 192.168.1.181, as is represented in Figure 7.6.

As a second use case, we involved relay VMs in the delivery process, in order to verify the efficiency of the system to reconfigure the flow rules and to verify the Notification System worked properly as well. In fact, booting the relay VM up generates an `add VM` event and triggers an alert message



```
▶Transmission Control Protocol, Src Port: 33111 (33111), Dst Port: sunproxyadmin (8081), Seq: 244, Ack: 1, Len: 81
▶[2 Reassembled TCP Segments (324 bytes): #3054(243), #3056(81)]
▼Hypertext Transfer Protocol
 ▶POST /VPM/VPMEventListener HTTP/1.1\r\n
  User-Agent: Java/1.7.0_51\r\n
  Host: localhost:8081\r\n
  Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2\r\n
  Connection: keep-alive\r\n
  Content-type: application/x-www-form-urlencoded\r\n
 ▶Content-Length: 81\r\n
  \r\n
  [Full request URI: http://localhost:8081/VPM/VPMEventListener]
  [HTTP request 1/1]
  [Response in frame: 3087]
▼Line-based text data: application/x-www-form-urlencoded
  data={"type":"VM","switch":"00:00:82:f2:6a:f6:15:4a","vnet":"vnet0/6","op":"ADD"}
```

Figure 7.7: The `add VM` event sent to the event listener servlet

for the system monitoring module, as demonstrated by the *Wireshark* trace shown in Figure 7.7.

We also verified that the system had successfully changed the flow rules installed on the involved relay nodes. The result of this second test use case was more than satisfactory, in fact the system applied the correct configuration on every node of the topology allowing the stream were delivered properly to the destination. It's worth noting that on the relay node was applied exactly the same configuration applied to leaf node except for the recipient address translation action, of course.

7.4 Migration use case

As for the test of the migration capability provided by the system, we tried to migrate the VM situated on the rightmost leaf of the topology (see Figure 7.3) to the rightmost one, while the node to migrate were contributing to deliver the streaming service to an end user. We used the graphical tool provided by the administration application introduced in Section 6.5.3 and shown in Figure 7.8.

The migration mechanism worked properly and, looking at the involved switch flows, we noticed that flows were not present anymore on the right path and a new path, toward the new leaf, had been automatically created. We repeated this test again, but this time paths were specified for both leaves. Results were the same: when migration completed, the left path

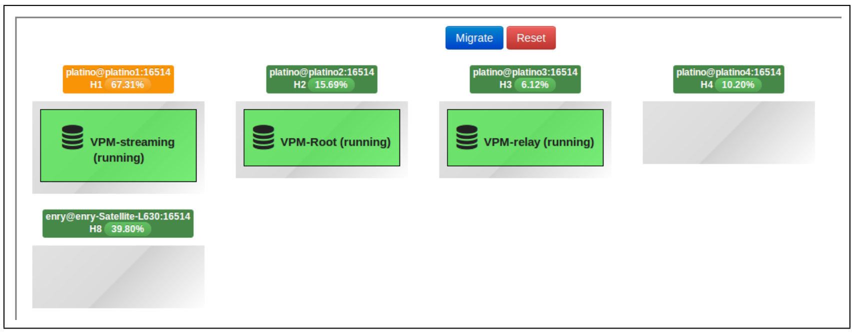
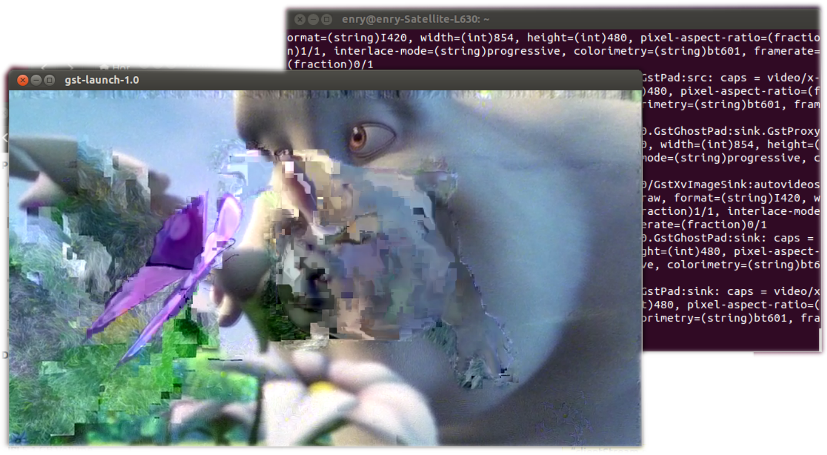


Figure 7.8: Administration panel: the migration tool



```
ormat=(string)I420, width=(int)854, height=(int)480, pixel-aspect-ratio=(fractio
n)1/1, interlace-mode=(string)progressive, colorimetry=(string)bt601, framerate=
(fraction)0/1
GstPad:src: caps = video/x-
)480, pixel-aspect-ratio=(f
rimetry=(string)bt601, fram

0.GstGhostPad:sink.GstProxy
0, width=(int)854, height=(
mode=(string)progressive, c

0/GstXvImageSink:autovideos
raw, format=(string)I420, w
raction)1/1, interlace-mode
erate=(fraction)0/1
0.GstGhostPad:sink: caps =
ght=(int)480, pixel-aspect-
ve, colorimetry=(string)bt6

GstPad:sink: caps = video/x
t)480, pixel-aspect-ratio=(
orimetry=(string)bt601, fra
```

Figure 7.9: Packet loss effect on a video during the VM migration process

disappeared, while right path was left untouched. About downtime, it was estimated, mediating results on 100 migration attempts involving the same video source, to be about 2 seconds. This result is perfectly reasonable in the context of a live streaming application especially if the migration grant a better overall quality of experience for the end users. In fact, only during the 2 seconds need to the physical migration of th VM, the user experienced some frame loss leading to just insignificant side-effects in the video played out (as it's shown in Figure 7.9) but without any interruption of the service.

Chapter 8

Conclusions and Future Work

In this thesis we present a comprehensive study on architectures devoted to support real-time multimedia applications over the Internet. Our contribution includes, functional requirements evaluation, design and implementation tasks, as well as the definition of new protocols and algorithms for system monitoring, control and network optimization.

More specifically our efforts have been devoted to designing and implementing a novel architecture aimed to support the aforementioned class of applications, with special regard to the streaming of real-time, live generated, multimedia contents use case. In this context, the most challenging goal for operators is to provide users with services that are comparable, in terms of perceived quality, to the traditional TV broadcasting system. As it has been explained, at the time of this writing the whole portfolio of real time multimedia streaming services is characterized by some common issues. First of all the need for users to install third party software or *plugins* for web browsers in order to be able to playout multimedia streams. Furthermore, existing streaming services are afflicted by high transmission delay. This fact can be very annoying when users are provided with a feedback channel to interact with the source of the streaming or when the type of event to be streamed etails real world social interferences. In case of sport events, for example, someone in the neighbourhood watching the same event on television could spoil the surprise for the crucial actions of the match by screaming and joying highly in advance.

To cope with such issues, we have presented a novel architecture aimed to provide a web-based real-time multimedia streaming service capable to serve a very large number of users. The framework we propose, called SOLEIL (Streaming Of Large scale Events over Internet cLouds), copes with the scalability requirement leveraging techniques that have been successfully exploited in the Content Delivery Networks (CDN), Cloud Computing and Virtualization fields. Our platform, in fact, envisages the deployment in a cloud environment of several elements cooperating to deliver the service to end users. More specifically, the server side of our architecture is based on an overlay network of virtual machines configured to create a self-organized tree topology. Each node of the tree plays one of the following roles: (i) *root* (ii) *relay* (iii) *leaf*. The source of the stream, i.e. the broadcaster, sends multimedia flows to the *root* node which trans-codes them in order to generate different quality replicas of the original flows. When a new user try to access the stream (by simply opening a web page) it is directed by the framework to one of the leaves of the tree depending on the overall status of the network and/or on particular client side constraints (type of device, available bandwidth, etc.).

System administrators can exploit graphic tools to draw and configure the nodes to be deployed, can assign them a specific role and draw connection links among nodes. In order the system could find the minimum tree by starting from this user level description of the topology, we relied on a consolidated minimization algorithm, namely the Kruskal's Minimum Spanning Tree algorithm, combined with some application logic needed to cope with possible human errors and to respect the role based nature of the distribution tree. In particular, dangling nodes are removed and a weight is associated to each node (on the base of its role) and to every link (summing the weights of the nodes it connects). According to how the Kruskal's algorithm is implemented, this process guarantees the minimum tree respects SOLEIL roles.

A distributed algorithm is used to provide the root node with information about the overall load status of the network and to allow it to dynamically manage the content distribution chain (by adding, removing or even migrating nodes). For this reason, we needed to provide the entity in charge of monitoring the system both information about the overall status of the entire topology and tools to query specific nodes about their current load and resource consumption status in a proactive way. This goal has been obtained by means of a protocol we designed for the purpose an we called SOLEIL Protocol. By using SOLEIL protocol messages, each node receives the statistic information from its direct child nodes by means of a RTCP ReceiverReport message and calculates an aggregate statistic to send to its direct predecessor in the tree topology by assigning to the received statistics a weight depending to the number of heirs of each sender node. This kind of information allows to properly monitor the overall status of the system. Once an alert about the status of a subtree or of a single node of the system arises, the root node can investigate, by means of direct messages, the specific load condition of that part of the system.

As for the real time requirement, communication among broadcaster, internal nodes and final users relies entirely upon the upcoming WebRTC/RtcWeb (Real-Time Communication in Web-browsers) technology, which guarantees pure web based access to multimedia streams, as well as support for real time delivery of multimedia contents. The general architecture of the platform has been furthermore optimized by relying on the innovation introduced by the so-called SDN technologies. The acronym SDN stands for Software Defined Networks and it represents an innovative approach to networking, attempting to keep pace with their great diffusion and utilization we are experiencing in latest years. By developing an application on the top of an SDN, a network administrator can obtain a substantial degree of scalability and reconfiguration that they would not be able to reach in any other way. We have hence proposed a framework for the performance optimization of the above mentioned streaming platform, as well as for the simplification of its management and administration functions. Specifically, SDN network support has allowed to apply proper forwarding rules to packets belonging to specific media flows and to control and modify such rules in a programmatic way directly from the high level application business logic. SDN has also led significant simplification and performance improvement to the node migration process letting us to implement an ad-hoc migration mechanisms based on the so called LIve Migration of Ensembles (LIME) approach. In fact, when a node migration is needed, the system checks whether or not the destination branch of the tree is already reached by the same flow crossing the node to migrate. If this is not the case, during the migration process, packets are replicated on the two paths (or cloned, if we want to use the term envisaged by the LIME approach). Once migration is successfully completed, the old path is removed. Migrated VM receive data right away it is waked up because the packets were already flowing to the new location during the migration process. This mechanism allows to minimize latency, downtime and packet loss. A prototype implementation of the system has been realized in a emulated network environment and used for realizing a functional testing campaign. Every functionality of the platform has been successfully tested so paving the way to the system industrialization and commercialization.

List of Figures

Bibliography

[1] Apple corporation. parallels desktop. http://store.apple.com/it/product/HD445ZM.

[2] Citrix. xen server. http://www.citrix.com/products/xenserver/overview.html.

[3] Kernel based virtual machine. http://www.linux-kvm.org/.

[4] Microsoft corporation. hyper-v. http://technet.microsoft.com/it-it/windowsserver/dd448604.aspx.

[5] Openflow experiment in real-time internet edutainment. http://www.ofertie.org/.

[6] Openvswitch. the openvswitch database managment protocol. http://tools.ietf.org/html/rfc7047.

[7] Oracle corporation. virtualbox virtualization technlogy. https://www.virtualbox.org/.

[8] Qemu. open source machine emulator and virtualizer. http://wiki.qemu.org/.

[9] Redhat corporation: Libvirt, the virtualization api. http://libvirt.org/.

[10] Snia. nas and iscsi technology overview. http://www.snia.org/.

[11] Third generation partnership project. http://www.3gpp.org/.

[12] Vmware. virtualization technlogies. http://www.vmware.com/.

[13] Floodlight. http://floodlight.openflowhub.org/, 2011.

[14] Alessandro Amirante, Tobia Castaldi, Lorenzo Miniero, and Simon Pietro Romano. Separation of Responsibilities between Application Servers and Media Servers in NGNs: A Practical Approach. In *Next Generation Teletraffic and Wired/Wireless Advanced Networking*, pages 199–211. Springer, 2008.

[15] Alessandro Amirante, Tobia Castaldi, Lorenzo Miniero, and Simon Pietro Romano. On the seamless interaction between webrtc browsers and sip-based conferencing systems. *Communications Magazine, IEEE*, 51(4):42–47, 2013.

[16] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. *Scalable application layer multicast*, volume 32. ACM, 2002.

[17] Jim Bankoski, Paul Wilkins, and Yaowu Xu. Technical overview of vp8, an open source video codec for the web. In *ICME*, pages 1–6. Citeseer, 2011.

[18] Mark Baugher, D McGrew, M Naslund, E Carrara, and Karl Norrman. The secure real-time transport protocol (srtp), 2004.

[19] Scott Bradner. The internet engineering task force. 1999.

[20] Tim Bray. The javascript object notation (json) data interchange format. 2014.

[21] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, page 16, 1998.

[22] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[23] Ralph Droms. Dynamic host configuration protocol. 1997.

[24] Eric,Arora,Dushyvant,Perez,Botero,Diego, Rexford, Jennifer Keller. Live Migration Of An Entire Network (and its hosts). Technical report, 2012.

[25] Stan Hanks, David Meyer, Dino Farinacci, and Paul Traina. Generic routing encapsulation (gre). 2000.

[26] Lawrence Harte. *Introduction to data multicasting.* Althos Publishing, 2008.

[27] T Hoff. Gone fishin': Justin. tv's live video broadcasting architecture. *High Scalability blog*, 2012.

[28] API Java and SQL Send. Java database connectivity.

[29] Jie Huang, Andrew P. Black, Jonathan Walpole, Calton Pu. Infopipes an Abstraction for Information Flow. *Computer Science Faculty Publications and Presentations*, 2001.

[30] Yao Lin, Jizhong Han, Jinjun Gao, and Xubin He. ustream: a user-level stream protocol over infiniband. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 65–71. IEEE, 2009.

[31] Linux Foundation Operative Project. Opendaylight Project. http://www.opendaylight.org/.

[32] Salvatore Loreto and Simon Pietro Romano. Real-time communications in the web: Issues, achievements, and ongoing standardization efforts. *IEEE Internet Computing*, 16(5):68–73, 2012.

[33] Scott Ludwig, Joe Beda, Peter Saint-Andre, Robert McQueen, Sean Egan, and Joe Hildebrand. Xep-0166: Jingle. *XMPP Standards Foundation*, 2009.

[34] Granberg Opshal Jan Magnus. *Open Source Virtualization*. PhD thesis, University of Oslo, 05 2013.

[35] Rohan Mahy, Philip Matthews, and Jonathan Rosenberg. Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun). *Internet Request for Comments*, 2010.

[36] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[37] Bill Nowicki. Nfs: Network file system protocol specification. 1989.

[38] Erik Nygren, Ramesh K Sitaraman, and Jennifer Sun. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[39] Open Networking Foundation. SDN Whitepaper. https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf, April 2012.

[40] OpenVSwitch. Production Quality, Multilayer Open Virtual Switch. http://openvswitch.org/.

[41] H Parmar and M Thornburg. Adobe's real time messaging protocol, 2014.

[42] Andrea Passarella. A survey on content-centric technologies for the current internet: Cdn and p2p solutions. *Computer Communications*, 35(1):1–32, 2012.

[43] David Passmore and John Freeman. The virtual lan technology report. *3COM White Paper*, 1996.

[44] Fabio Picconi and Laurent Massoulié. Is there a future for mesh-based live video streaming? In *Peer-to-Peer Computing, 2008. P2P'08. Eighth International Conference on*, pages 289–298. IEEE, 2008.

[45] Popek, Gerald J.; Goldberg, Robert P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 05 1974.

[46] Robert E. Tarjan. Data structures and network algorithms, 1983.

[47] Alfonso V Romero. *VirtualBox 3.1: Beginner's Guide*. Packt Publishing Ltd, 2010.

[48] Robert Rose. Survey of system virtualization techniques. 2004.

[49] Jonathan Rosenberg and Ari Keranen. Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols. 2013.

[50] Jonathan Rosenberg, Rohan Mahy, Philip Matthews, and Dan Wing. Session traversal utilities for nat (stun). Technical report, RFC 5389 (Proposed Standard), 2008.

[51] Jonathan Rosenberg, Henning Schulzrinne, Gonzalo Camarillo, Alan Johnston, Jon Peterson, Robert Sparks, Mark Handley, Eve Schooler, et al. Sip: session initiation protocol, 2002.

[52] Mendel Rosenblum. Vmware's virtual platform. In *Proceedings of hot chips*, volume 1999, pages 185–196, 1999.

[53] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W Moore. Oflops: An open framework for openflow switch evaluation. In *Passive and Active Measurement*, pages 85–95. Springer, 2012.

[54] Henning Schulzrinne. Rtp: A transport protocol for real-time applications. 1996.

[55] Henning Schulzrinne. Real time streaming protocol (rtsp). 1998.

[56] Henning Schulzrinne, S Casner, R Frederick, and Van Jacobson. Real-time transport protocol. *RFC1899*, 1996.

[57] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the h. 264/avc standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 17(9):1103–1120, 2007.

[58] Scott Lowe. Examining Openvswitch traffic patterns. http://blog.scottlowe.org/2013/05/15/examining-open-vswitch-traffic-patterns/, 05 2013.

[59] Randall Stewart. Stream control transmission protocol. 2007.

[60] Thomas Stockhammer. Dynamic adaptive streaming over http–: standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 133–144. ACM, 2011.

[61] George Tsirtsis. Network address translation-protocol translation (nat-pt). *Network*, 2000.

[62] Athena Vakali and George Pallis. Content delivery networks: Status and trends. *Internet Computing, IEEE*, 7(6):68–74, 2003.

[63] Anthony Velte and Toby Velte. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[64] W. Taymans, S. Baker, A. Wingo, R. Bultje, and S. Kost. Gstreamer application development manual. Technical report, 2001.

[65] Stephan Wenger. H. 264/avc over ip. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):645–656, 2003.