Design and Implementation of a Generic Library for P2P Streaming

Luca Abeni
University of Trento
Trento, Italy
luca.abeni@unitn.it

Csaba Kiraly
University of Trento
Trento, Italy
kiraly@disi.unitn.it

Alessandro Russo
University of Trento
Trento, Italy
russo@disi.unitn.it

Marco Biazzini
University of Trento
Trento, Italy
biazzini@disi.unitn.it

Renato Lo Cigno
University of Trento
Trento, Italy
locigno@disi.unitn.it

ABSTRACT

Practical implementation of new P2P streaming systems requires a lot of coding and is often tedious and costly, slowing down the technology transfer from the research community to real users. GRAPES aims at solving this problem by providing a set of open-source components conceived as basic building blocks for building new P2P streaming applications which have in mind the savvy usage of network resources as well as the Quality of Experience of final users. GRAPES is designed to be usable in different environments and situations, to have a minimum set of pre-requisites and dependencies, and not to impose any particular constraints on applications using it. Our experience shows that GRAPES allows the rapid development of P2P streaming applications by writing a small amount of glue code to connect the desired functionalities. Some examples (including some simple P2P streamers) are also discussed as a means to show code usage.

Categories and Subject Descriptors

C.2 [**Computer-Communication Networks**]: Distributed Systems; D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*

General Terms

Design; Experimentation

Keywords

P2P Streaming, Implementation, GRAPES

1. INTRODUCTION

Peer-to-peer (P2P) technologies are becoming increasingly popular as a way to overcome the scalability limitations intrinsic in traditional network applications based on a client/server paradigm. In particular, there is a great interest in P2P streaming applications (both Video on Demand and TV-like live broadcasting systems), because they have high demands in terms of bandwidth requirements. IP-level multicast could help in reducing the network bandwidth required by an audio/video streaming system, but it is not supported on the Internet.

In the last years the scientific community produced a lot of research work on improving the P2P streaming technologies to provide better quality and to better exploit the available resources; however, commonly used P2P TV applications [2, 1, 3] do not always follow such trends, and are still based on architectures deriving from file sharing applications, whose requirements are far apart from (live)streaming. As a result, the network bandwidth is not used efficiently, and the P2P TV clients can provide a good user experience only by being very aggressive in network usage.

In the authors opinion, such a lack of technology transfer from the research community to commonly used applications can be solved through the availability of open-source P2P streaming applications providing the researchers with a working codebase that can be easily modified to experiment with and to integrate novel ideas. To target this goal, it is useful to remember how the free availability of open-source kernels as Linux or FreeBSD helped the OS research community! Unfortunately, integrating experimental techniques in an application sometimes requires drastic changes to its structure: for example, changing the local chunk selection algorithm can be easy, but changing a P2P streaming application from "epidemic style" push [4] to pull [8, 18], or more complex strategies [16] can require a complete redesign.

In other words, developing *one single open source application* can impose too many constraints on the research that can be performed on it, requiring to rewrite the application every time a new solution has to be tested (in [12] the authors had to implement 5 different P2P streaming application, writing more than 10000 lines of C++ code).

This paper proposes a set of *generic* and *reusable* components forming a codebase to develop P2P streaming applications with (almost) any structure. Such a toolkit, named GRAPES (Generic Resource Aware P2P Environment for Streaming), provides a set of building blocks that researchers can use, combine, and modify to test and compare different solutions fostering the development of new ideas as it hap-

AVSTP2P'10, October 29, 2010, Firenze, Italy.

pened in OS research [7]. To the authors' best knowledge, similar toolkits do not currently exist in the P2P community. A possibly close work [6] identified a common API for a Key-based Routing Layer that can be used as a base for various P2P services, but no implementation of such an API has been freely released to the research community.

The paper is organised as follows: Section 2 discusses the requirements for GRAPES, which drove the main design choices and influenced the GRAPES structure, described in Section 3; Section 4 describes how to use GRAPES, and presents some early experiences (showing how GRAPES can simplify the development of P2P streaming applications); finally, Section 5 presents the conclusions and describes ongoing work.

2. REQUIREMENTS

First of all, the functionalities provided by GRAPES should be usable by as many applications as possible in as many different environments as possible. This means that GRAPES should be able to run in many different platforms/operating systems, and should be accessible to many different development tools and runtime environments. For this reason, GRAPES has been implemented as a C library, since almost every development platform provides a C compiler, and it is quite easy to develop bindings to other languages such as python, java, etc... C++ programs can directly link to the GRAPES library without needing any kind of wrapper or special bindings. Moreover, C does not requires complex runtime support, or system libraries.

For the same reason, the amount of dependencies for GRAPES has been reduced to the minimum (no dependencies on external libraries, etc). The result is that GRAPES can be used on many different systems, ranging from large network servers to small and not-so-powerful embedded devices.

A second requirement, more difficult to fulfil, is that GRAPES should not impose any particular structure to the applications using it, so that the library can be used applying different programming paradigms (ranging from event-based, reactive, programming to thread-based multiprogramming). In this way, GRAPES supports different programming styles (some programmers prefer thread-based multiprogramming because of its simplicity, while others are against the thread abstraction [14]). This second requirement has some serious implications on the API exported by the library, since concurrency handling and support for parallel activities have to be moved from the library to the application using it. As it will be shown in the following, this has been obtained by removing from GRAPES the code for receiving data from remote peers (and for demultiplexing the received data), and by leaving such a task to the application. The received data will then be passed to GRAPES by invoking an appropriate data parsing function.

A third requirement is modularity: GRAPES should provide all the basic functionalities needed by a generic P2P application, without forcing the application to use unneeded code. For this reason, the GRAPES functionalities have been grouped in several *modules* (that will be described in Section 3) and described by a well defined API. Each module has its own API (described by a C header file), and can be used independently from the others (if a module needs the functionalities of a different one, it will use them through the public API, so that each module can be easily replaced by a user-provided implementation). As previously mentioned, all the modules that need to interact with remote peers export a data parsing function (named `ParseData()`, plus a prefix dependent on the module name).

Applications based on GRAPES communicate through *messages*, which can contain data to be diffused or signalling information. Such messages are encapsulated in network packets and sent by using a network abstraction layer, named *network helper*, which allows to easily change the protocol used for transmitting the messages, to port GRAPES to different architectures, and so on. GRAPES modules can directly send messages (by invoking the network helper), or can simply construct them, leaving to the application the responsibility of sending the messages (still through the network helper). On the receiving side, applications are responsible for invoking the network helper to receive messages, and pass them to the correct GRAPES modules (by invoking the correct `ParseData()` function). This architecture also enables optimisations like embedding multiple messages in a single packet to reduce the network traffic.

3. DESIGN AND STRUCTURE

Since GRAPES aims at providing the basic blocks needed for building a P2P streaming application, the most important modules composing such applications have been identified. A preliminary analysis revealed that a generic P2P streaming application usually needs, in addition to the net helper module:

A *Peer Sampling* mechanism, providing each peer with continuously up-to-date random samples of the entire population of peers;

A *Chunk Trading* module, allowing to send/receive pieces of a media stream (called *chunks*);

A *Chunk Buffer*, used to store the received chunks so that they can be forwarded to the other peers;

A *Chunk ID Set* data type, that can be used to send signalling information about the received or needed chunks;

Some *Scheduling* functions, which can be used to decide which chunk to send/ask, to which peer;

A *Peer Set* data type, to store neighbours in a structure describing the overlay.

The goal of the various GRAPES modules is to hide the implementation of the mechanisms introduced above, and to make them accessible through an uniform and generic programming interface. In this way, implementation details of the various mechanisms can be easily changed without affecting the application code.

Note that although the following description of GRAPES is based on an example using all its functionalities, applications are free to use only the needed modules. For example, a streamer based on a tree-like overlay will probably not use the Peer Sampling module; other applications can use GRAPES' Peer Sampling implementing their own chunk buffer, etc...

3.1 Peer Sampling

The Peer Sampling module is used by a P2P application for joining an overlay: the application provides to the peer sampling mechanism one (or more) known peers, and can

obtain a "view" containing a random sample of all the peers currently active in the system. Such a mechanism can be implemented by using a gossipping protocol like NewsCast [9] or CYCLON [17], or some other mechanism (the use of a centralised database has been proposed in some situations [11]). Currently, a simple gossipping protocol (similar to NewsCast) has been implemented, and an implementation of CYCLON will be released soon. If a specialised peer sampling mechanism is needed, it can be easily implemented in this module, and made available to all the applications using the GRAPES API. The most important functions exported by the Peer Sampling module are:

- `Init()`, to initialise the peer sampling service, and assign a local address to it
- `ParseData()`, invoked when a peer sampling message from a remote peer is received
- `AddNeighbour()`, to provide the ID of known peers; mainly used for bootstrapping
- `RemoveNeighbour()`, to remove a peer from the local view
- `GetNeighbourhood()`, which returns a random sample of the peers (a list of the known peers)
- `GrowNeighbourhood()` and `ShrinkNeighbourhood()`, to modify the size of the local view (the set of known peers)
- `GetMetadata()` and `ChangeMetadata()`, to get and modify the metadata associated to a peer.

Moreover, some *metadata* can be associated to each peer, to describe its attributes (for example, upload bandwidth, etc...). The metadata are application-specific, and are handled transparently by GRAPES.

3.2 Signalling and Chunk Trading

Once an application has a list of some of the peers participating to the P2P system, it can exchange signalling messages (telling which chunks it needs and/or which chunks it can provide to the other peers) and chunks with the other peers.

GRAPES provides powerful and generic signalling protocol primitives, which allow to implement a large set of different chunk trading mechanisms. Analysing the signalling messages required by various chunk trading protocols found in the literature, it is clear that most signalling messages send a set of chunk IDs, but with different semantics. A buffermap, for example, is a set of chunk IDs encoded normally as a bitmap, just as a chunk request is usually a set with only one element.

Therefore, GRAPES provides a Chunk ID Set datatype that can be used for storing the IDs of the chunks owned (or needed, or offered, or accepted, ..., depending on the required semantics) by a peer. The signalling functionalities also provide a low-level API with a generic `encodeChunkSignaling()` function, that takes a Chunk ID Set as a parameter, transforming it in a message to be sent on the network through the network helper. Metadata explaining the type and the semantic of the message (e.g., whether the chunks are needed or offered) and information about the chunk trading protocol (e.g. a transaction ID) can also be encoded by `encodeChunkSignaling()` and added to the message. Note that the current implementation supports different encoding schemes, e.g., Chunk ID Sets can be encoded both as bitmaps (useful for large dense sets), as well as lists (useful for sparse sets or for debugging purposes, optionally assigning priorities and other information to the various chunk instances. The different encoding schemes support different trade-offs between bandwidth usage and represented information.

Like the Chunk ID Sets, chunks can be transmitted by using appropriate `encode()` (`encodeChunk()`, for transforming a chunk or a Chunk ID Set in a message) and `decode()` functions (`decodeChunk()`, for transforming a message in a chunk).

Based on the above low-level function, an API consisting of "high-level" functions has also been built. Thanks to such a high-level API, GRAPES implements a large set of signalling protocols, which allow the composition of very different chunk trading mechanisms. The following signalling functionalities have been implemented based on `encodeChunkSignaling()`:

- buffermap message (to inform another peer about the status of the chunk buffer)
- chunk offer (to offer a set of chunks to another peer)
- chunk accept (the response to an offer message)
- chunk request (ask for one or more chunks)
- chunk deliver (response to a request)

Various chunk trading logics can easily be obtained from the above primitives: A simple "useful" push protocol (used frequently in papers analysing epidemic streaming) [4] uses only buffermap messages; a pull protocol [8, 18] will use request and deliver messages; and a more complex one that uses bufferstate information to send pull requests to selected peers will use all the buffermap, request, and deliver messages. More complex trading protocols, such as an offer-accept protocol are also supported: the Technical Report [15] reports examples of streamers built using GRAPES.

At the implementation level, the above messages are rather similar and therefore easily parseable. Signalling information are encapsulated in messages by a message type, by using the Chunk ID Set datatype, and some other information (encoded as *metadata*) such as a transaction ID and the maximum number of chunks to be delivered.

3.3 The Chunk Buffer

Chunks received by an application are generally stored in a Chunk Buffer, from which they are taken for forwarding the stream to other peers. GRAPES provides a Chunk Buffer API which enables to store the received chunks, and to get a list of the currently stored chunks. The application does not have to care about the data structure's internals, and GRAPES is responsible for ordering the chunks, removing the duplicates, discarding chunks that are too old, etc ....

Different buffer management policies are possible:

1. the buffer discards chunks when a maximum size has been reached;
2. chunks are discarded when the difference between their playback time and the current time is too large.

Other, more advanced, policies can be designed and added to the library without affecting its interface, or the user code (for example, storing all the chunks for a larger time, which can be more useful for VoD systems). Many parameters, like the buffer size, are configurable through the initialisation call.

The functions exported by the Chunk Buffer module are:

- `cb_init()`, to initialise a chunk buffer, setting its size and some important parameters
- `cb_add_chunk()`, to insert a new chunk in the buffer
- `cb_get_chunks()`, returning an ordered list of all the chunks which are currently stored in the buffer
- `cb_get_chunk()`, returning a specified chunk from a buffer
- `cb_clear()`, to remove all the chunks from a buffer
- `cb_destroy()`, to destroy a buffer, freeing all the resources used by it

3.4 Scheduling

During the streaming, an application often has to select chunks to send / receive, or remote peers to contact for chunk trading. All of these decisions are performed by the *peers and chunks scheduler*. Note that the scheduling decisions to be taken depend on the chunk trading protocol that the application implements. For example, when using an epidemic streaming approach an application periodically sends a chunk to a neighbour. Hence, it needs a chunk scheduler to select the chunk to be sent and a peer scheduler to select the target peer to which the chunk will be sent. In alternative if the application is based on a *pull* protocol, it has to select a set of chunks to be requested to a neighbour, and a neighbour to which the chunks are requested: Two scheduling functions are still needed, but they work in a different way respect to the "push" schedulers. Finally, more complex protocols (such as an offer/trade protocol) can be used, but any peer has still to take scheduling decisions about the chunks to offer, and about the offers that it wants to accept.

The GRAPES scheduler provides fundamental scheduling functions that can be used in the situations described above, and are, in the authors' opinion, generic enough to be used in many other situations. Furthermore a *scheduling framework* that can be used to implement new and more specialised schedulers is provided. The final goal is to have a scheduling API which is compatible with the one used by SSSim [5], so that schedulers can be easily moved from the simulator to real applications and vice-versa[1].

3.5 Other Modules

Other modules are currently under development and will be available in the next releases of the software. For example, a new module will contain some *topology management* algorithms such as TMan [10] that allow to build a more structured overlay based on the random view provided by the Peer Sampling module.

[1]This feature has not been implemented yet, but will be available in future releases.

Another important GRAPES module which has not been fully yet allows to connect a P2P application with the `libavcodec` and `libavformat` libraries[2], to encode/decode audio and video, and to handle multimedia formats. Such a module can be used in the input and output parts of a P2P streaming application, to implement *media aware streaming* (for example, inserting an integer number of frames in each chunk, or assigning different importance to different chunks based on the presence of reference frames in them). The functionalities of this module have already been implemented, and [13] shows how to use such functionalities together with a simulator, but they have also been used in a real streamer. The code is available at `http://imedia.disi.unitn.it/QoE`, but it still has to be integrated in the library exporting a powerful-but-generic enough API.

4. USAGE AND EARLY EXPERIENCES

Applications based on GRAPES can use the library's public interface (exported through some C header files) to initialise the network helper and the various GRAPES modules, to send/receive messages, and to pass them to the appropriate `ParseData()` function. The typical application will

1. Initialise the various components
2. Enter a loop in which it:
   (a) Receive messages
   (b) Demultiplex them and pass them to the appropriate module
   (c) Eventually send back messages (this can be done by the module itself

Figure 1 shows how to do this in a single-threaded application. The `wait4data()` function, exported by the network helper, allows the application to block waiting for a message or for a timeout to fire. If a message arrives before the timeout fires, the message is received (`recv_from_peer()` and is passed to the proper `ParseData()` function, selected through a `is*()` function (in this example, only the handling of the Peer Sampling messages - identified by `isTopology()` - is shown; if the application uses more GRAPES modules, other `is*()` and `*ParseData()` functions will be invoked - in place of the "else check if the message goes to other GRAPES modules" comment).

Based on the structure described above, a simple application which builds a P2P overlay by using the Peer Sampling service has been written with about 100 lines of C code. Such a program compiles in an executable large about 10 kB, which requires less than 2 kB of data to execute.

As previously explained, GRAPES does not force any particular application structure, so it can also be used in a multi-thread environment, as shown in Figure 2. Note that in this case the application is responsible of ensuring mutual exclusion on the GRAPES functionalities and data structures (by using appropriate mutexes), so GRAPES does not depend on any specific threading library. Alternative implementations of the network helper which allow to use GRAPES in event-based programs have been developed and will be integrated in the main codebase soon.

[2]`http://www.ffmpeg.org`

```
struct nodeID *my_id;

my_id = net_helper_init(my_addr, my_port);
topInit(myID, NULL, 0, "");

while(!done) {
  new_msg = wait4data(s, &timeout, NULL);
  if (new_msg) {
    len = recv_from_peer(s, &remote,
                         buff, BUFFSIZE);
    if (isTopology(buff)) {
      topParseData(buff, len);
    } /* else check if the message
         goes to other GRAPES modules */
    nodeid_free(remote);
  } else {
    /* Invoke Parse functions with NULL
       argument, to check for timeouts */
    topParseData(NULL, 0);
    /* Other modules' Parse() */
  }
}
```

Figure 1: Single-threaded usage of GRAPES.

To test the portability of the library, some tests have been cross-compiled for an embedded platform (an ARM-based board) and successfully tested on it. This proves that the library's dependencies are minimal, and that GRAPES-based applications can be used in resource-constrained environments.

By using the GRAPES library, a simple but functional P2P video streamer (based on epidemic streaming techniques) has been written with about 900 lines of C code. Since it is based on the GRAPES API, it is quite simple to change the chunk or peer scheduling algorithms, the chunk buffer implementation, the peer sampling protocol, or other algorithms without large changes in the streamer's code. If compared with some previous works [12] (where more than 10000 lines of code had to be written), these results represent a considerable improvement, and enable easier experimentation with novel P2P streaming approaches. The streamer program has been developed, debugged, and tested in less than one day, and only depends on GRAPES (additional dependencies on audio/video libraries can be added to use advanced chunkisation strategies - see below); the executable size is about 26 kB (about 21 kB of code), and it needs less than 2 kB of memory for data to execute.

The generation and the playback of chunks is based on `libavcodec` and `libavformat` (as explained in Section 3.5, the corresponding code will be moved into GRAPES in the next releases, and these functionalities will be exported through a generic API), and can be easily modified to experiment with new media-aware chunkisation techniques (for example, using 1 GoP per chunk, or grouping frames into chunks according to their types, or using more advanced temporal scalability approaches). Moreover, it is very simple to change the video codec, or the encoding parameters, to verify which codecs/parameters are more suitable for P2P streaming applications. If the dependencies on `libavcodec` and `libavformat` are removed (by disabling support for advanced chunkisation techniques in the input and output modules), the streamer is still able to receive and forward chunks, and

```
void *ps_thread(void *arg)
{
  topInit(myID, NULL, 0, "");
  while(!done) {
    pthread_mutex_lock(&topology_mutex);
    topParseData(buff, len);
    pthread_mutex_unlock(&topology_mutex);
    usleep(gossiping_period);
  }

  return NULL;
}
/* Thread bodies for other GRAPES modules */

void *recv_thread(void *arg)
{
  while(!done) {
    len = recv_from_peer(s, &remote,
                         buff, BUFFSIZE);
    if (isTopology(buff)) {
      pthread_mutex_lock(&topology_mutex);
      topParseData(buff, len);
      pthread_mutex_unlock(&topology_mutex);
    } /* else check if the message goes
         to other GRAPES modules */
    nodeid_free(remote);
  }

  return NULL;
}

int main(int argc, char *argv[])
{
  my_id = net_helper_init(my_addr, my_port);

  pthread_create(&id1, NULL,
                 recv_thread, NULL);
  pthread_create(&id2, NULL,
                 ps_thread, NULL);
  pthread_create(...);  /* Create threads
    for the other GRAPES modules ... */
```

Figure 2: Multi-threaded usage of GRAPES.

can be used as a *superpeer* or to improve the available upload bandwidth in a P2P system.

Thanks to the flexibility of the GRAPES API, the streamer has been modified to implement different chunk trading protocols [15], allowing to run experiments to compare such protocols through real tests on the internet (and not through simulations).

5. CONCLUSIONS AND FUTURE WORK

This paper described GRAPES, a toolkit for easily and rapidly developing P2P streaming applications. A first release of GRAPES is available at `http://imedia.disi.unitn.it/P2PStreamers/grapes.html`. This first release provides a net helper for using the UDP protocol on POSIX systems (it has been tested on GNU/Linux, some BSDs, MacOS X, and some other POSIX compliant systems), a simple but functional implementation of the modules described in Section 3, and some examples and tests. An additional peer sampling algorithm (CYCLON) has already been implemented, but is not included in the first release, and some additional modules (such as a topology manager) are under development and will be included in the next release.

This first release of GRAPES has already been used as

a base for building some experimental P2P video streaming software[3].

As a future work, more modules will be integrated in GRAPES, and some GRAPES-base applications will be used for performance measurements in experimental P2P streaming systems.

6. ACKNOWLEDGEMENTS

This work is partly supported by the European Commission through the NAPA-WINE Project (Network-Aware P2P-TV Application over Wise Network[4]), ICT Call 1 FP7-ICT-2007-1, 1.5 Networked Media, grant No. 214412.

The authors are also deeply in debt to all the people involved in the NAPA-WINE project for the fruitful discussions and for patiently using the GRAPES code for the development of applications following the NAPA-WINE approach and architecture.

7. REFERENCES

[1] Coolstreaming. `http://live.coolstreaming.us`.

[2] Pplive. `http://pplive.com`.

[3] Sopcast. `http://www.sopcast.com`.

[4] Abeni, L., Kiraly, C., and Lo Cigno, R. On the optimal scheduling of streaming applications in unstructured meshes. In *Networking 09* (Aachen, DE, May 2009), Springer.

[5] Abeni, L., Kiraly, C., and Lo Cigno, R. SSSim: a simple and scalable simulator for p2p streaming systems. In *Proceedings of IEEE CAMAD '09* (Pisa, Italy, June 2009).

[6] Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., and Stoica, I. Towards a common API for structured peer-to-peer overlays. *Peer-to-Peer Systems II* (2003), 33–44.

[7] Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., and Shivers, O. The flux oskit: a substrate for kernel and language research. In *Proceedings of SOSP '97* (Saint Malo, France, 1997), ACM, pp. 38–51.

[8] Hei, X., Liu, Y., and Ross, K. Iptv over p2p streaming networks: the mesh-pull approach. *Communications Magazine, IEEE 46*, 2 (february 2008), 86 –92.

[9] Jelasity, M., Guerraoui, R., Kermarrec, A.-M., and van Steen, M. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In *Middleware 2004* (2004), H.-A. Jacobsen, Ed., vol. 3231 of *LNCS*, Springer-Verlag.

[10] Jelasity, M., Montresor, A., and Babaoglu, O. T-man: Gossip-based fast overlay topology construction. *Comput. Netw. 53*, 13 (2009), 2321–2339.

[11] Li, H., Clement, A., Marchetti, M., Kapritsos, E., Robison, L., and Dahlin, M. Flightpath: Obedience vs. choice in cooperative services. In *Proceedings of OSDI '08* (San Diego, CA, December 2008).

[12] Liang, C., Guo, Y., and Liu, Y. Is random scheduling sufficient in p2p video streaming? In *Proceedings of ICDCS 2008* (Los Alamitos, CA, USA, June 2008), IEEE Computer Society, pp. 53–60.

[13] Kiraly, C., and Lo Cigno, R., and Abeni, L. Deadline-based Differentiation in P2P Streaming In *Proceedings of IEEE Globecom '10* Miami, FL, USA, Dec. 2010.

[14] Renesse, R. V. Goal-oriented programming, or composition using events, or threads considered harmful. In *Proceedings of ACM SIGOPS EW98 Support for Composing Distributed Applications* (Sintra, Portugal, 1998), ACM, pp. 82–87.

[15] Russo, A., Biazzini, M., Kiraly, C., Abeni, L., and Lo Cigno, R. Implementing Streamers with GRAPES: Initial Experience and Results. Tech. rep., TR-DISI-10-039, University of Trento, 2010. http://disi.unitn.it/locigno/preprints/TR-DISI-10-039.pdf.

[16] Russo, A., and Lo Cigno, R. Delay-Aware Push/Pull Protocols for Live Video Streaming in P2P Systems. In *IEEE ICC 2010* (Cape Town, South Africa, May 2010).

[17] Voulgaris, S., Gavidia, D., and Van Steen, M. CYCLON: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management 13*, 2 (2005), 197–217.

[18] Zhang, M., Zhang, Q., Sun, L., and Yang, S. Understanding the power of pull-based streaming protocol: Can we do better? *Selected Areas in Communications, IEEE Journal on 25*, 9 (december 2007), 1678 –1694.

[3] `http://imedia.disi.unitn.it/P2PStreamers`

[4] `http://www.napa-wine.eu`