Department of
Information Engineering
and Computer Science
DISI

UNIVERSITY
OF TRENTO - Italy

DISI - Via Sommarive, 14 - 38123 POVO, Trento - Italy
http://disi.unitn.it

GRAPES: a Generic Environment for P2P Streaming

Luca Abeni, Csaba Kiraly, Alessandro Russo, Marco Biazzini, Renato Lo Cigno

June 2010

Technical Report # DISI-10-038
Version 1.0

This work is supported by the European Commission through the NAPA-WINE Project (Network-Aware P2P-TV Application over Wise Network – www.napa-wine.eu), ICT Call 1 FP7-ICT-2007-1, 1.5 Networked Media, grant No. 214412

GRAPES: a Generic Environment for P2P Streaming

Luca Abeni, Csaba Kiraly,
Alessandro Russo, Marco Biazzini, Renato Lo Cigno
University of Trento, Trento - - Italy
luca.abeni@unitn.it, kiraly@disi.unitn.it,
russo@disi.unitn.it, biazzini@disi.unitn.it,
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 relieving this burden, and speed-up application development, by providing a set of open-source components conceived as basic building blocks for building new P2P streaming applications. GRAPES is designed to be usable in different environments and situations, to have a minimal set of pre-requisites and dependencies, and not to impose any particular constraints on applications using it. The development of streamers in the Napa-Wine project [1] has shown that using GRAPES P2P streaming applications with different characteristics can be developed by only writing a small amount of *glue* code that connects the desired functionalities and dictates the architecture of the streamer.

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, Video on Demand and TV-like systems, because they have high demands in terms of bandwidth requirements and IP-level multicast 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, 3, 4] not always follow such trends, and are still based on a architectures deriving from file sharing. 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 researchers with a working codebase that can be easily modified to experiment and integrate novel ideas. In this sense, remember how the free availability of open-source kernels such as Linux or BSD 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 (the so-called "chunk scheduler") can be easy, but changing a P2P streaming application from "epidemic style" push [5] to pull [6, 7], or more complex strategies [8] can require a complete redesign. Hence, developing *one single open source application* risks to 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 (to look at real numbers, in [9] the authors had to implement 5 different P2P streaming application, writing more than 10000 lines of C++ code).

For this reason, within the NAPA-WINE project, a set of *generic* and *reusable* components has been designed and developed, to provide a codebase for building P2P streaming applications with (almost) any structure. The resulting 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 ideas fostering the development of new ideas as it happened in OS research [10].

This report is organised as follows: Sect. 2 discusses the requirements for GRAPES, which drove the main design choices and influenced the GRAPES structure, described in Sect. 3; Sect. 4 describes how to use GRAPES, and presents some early experiences (showing how GRAPES can simplify the

development of P2P streaming applications); Sect. 5 is a snapshot of the status of GRAPES at the time of this publication; and Sect. 6 finally 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). This supports different programming styles (some programmers prefer thread-based multiprogramming because of its simplicity, while others are against the thread abstraction [11]). This second requirement has some serious implications on the API exported by the library, since concurrency handling and parallelism 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 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).

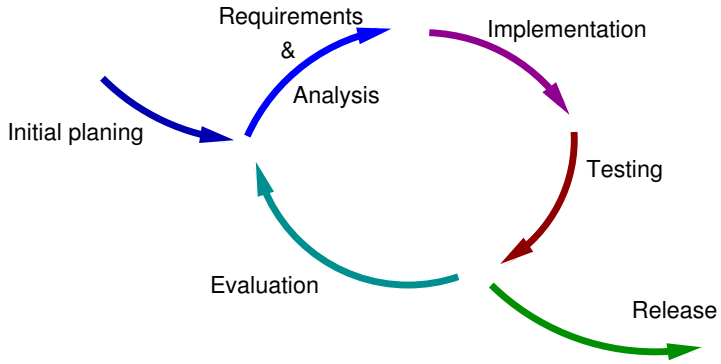

Figure 1: IID software development model

To ensure that all the previously mentioned requirements are properly satisfied, GRAPES development is inspired to an agile, incremental, iterative development model (IID for short) depicted in Fig. 1. The "initial planning" are the basic functionalities (e.g., chunks must be transferred from one peer to another); "Requirements & analysis" is a refinement step functional to design a simple implementation of the general planning; "Implementation" and "Testing" require no comments, while the "Release" step indicates the pulication of specific (tested and verified) versions of the software from time to time.

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

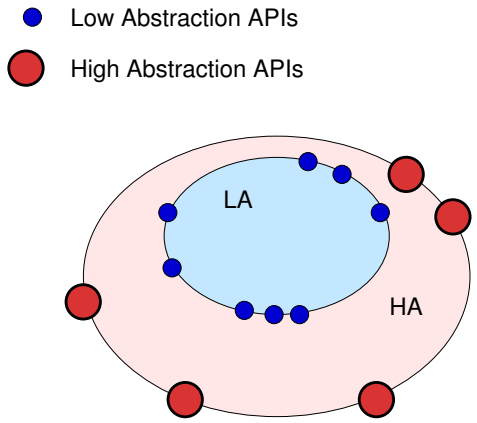

Figure 2: Software abstraction levels and APIs available in GRAPES

GRAPES is structured according to a two-tier abstraction level, as described in Fig. 2. The Low Abstraction (LA) level contains elementary (not necessarily simple) functions, which expose Application Programming Interfaces (APIs) that are concerned with *how* the functions are implemented, including, for instance, details about the data structures, the presence of metadata in these structure (which imply that the calling function/application has knowledge about this metadata and its meaning) and so on. The High Abstraction (HA) level contains instead composite software functions (not necessarily complex) that hide from the library user and the calling applications all the details of how functions are implemented, and concentrates solely on *what* the function must do, and not with the details of how to do this.

Both HA and LA APIs are public and exposed, separated and clearly identified in GRAPES, leaving the application designer the freedom of using the HA or LA functions as he deems fit.

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 Set* data type, to store information about the nodes connected to a specific peer in the overlay (the so called *neighbours*);

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;

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

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 API. In this way, changing implementation details will be very easy, and will have no impact on the application code.

Note that although this 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 one (or more) known peers, and can obtain a "view" containing a random sample of the peers currently active in the

system. Such a mechanism can be implemented by using a gossipping protocol like NewsCast [12] or CYCLON [13], or some other mechanism (the use of a centralised database has been proposed in some situations [14]). A simple gossipping protocol 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), and `GetNeighbourhood()` (which returns a list of the known peers). 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 empowers the implementation of 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 signalling provides a low-level API with a generic `encodeChunkSignaling()` function, that takes a Chunk ID Set and some metadata as parameters, transforming them in a message to be sent on the network through the network helper. The Chunk ID Set datatype is used for storing the IDs of the chunks owned by a peer (or offered, or accepted, ..., depending on the required semantics). Metadata explaining the type 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 be added to the message. Like the Chunk ID Sets, chunks can be transmitted by using appropriate `encode()` (for transform-

ing a chunk or a Chunk ID Set in a message) and `decode()` functions (for transforming a message in a chunk, or in a Chunk ID Set)[1].

Based on the above low-level function, an API consisting of "high-level" functions was also built. 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) and 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) [5] uses only buffermap messages; a pull protocol [6, 7] 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 push/pull protocols [15, 8], are also supported: the Technical Report [16] reports examples of streamers built using GRAPES.

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: i) the buffer discards chunks when a maximum size has been reached; ii) chunks are discarded when the difference between their playback time and the current time is too large; and others can be added, like 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.

[1]The 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)

The most important 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), and `cb_get_chunks()` (returning an ordered list of the chunks which are currently stored in the buffer). Additional functions for clearing and destroying (freeing) a buffer are also provided.

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;
- 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;
- more complex protocols 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 schedulers provide 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 [17], so that schedulers can be easily moved from the simulator to real applications and vice-versa[2].

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

3.5 Other Modules

Other modules are currently under development and will be available in the next releases of the software. For example, the development of a new module that contain *topology management* algorithms such as TMan [18] that allow to build a more structured overlay based on the random view provided by the Peer Sampling module has just been released.

Another important GRAPES module which has not been fully developed yet allows connecting a P2P application to the `libavcodec` and `libavformat` libraries[3], 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[4], but the code 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 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) Send back messages if needed (this can be done by the module itself).

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

[4] `http://imedia.disi.unitn.it/QoE` shows how to use such functionalities together with a simulator, but they have also been used in a real streamer.

Figure 3 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 $10KB$, which requires less than $2KB$ 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 4. 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.

To test the portability of the library, some tests have been cross-compiled for an embedded platform (an ARM-based board) and succesfully tested on it. This proves that the library's dependencies are reasonable, and that GRAPES-based application 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 simply uses 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 [9] (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 1 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 kbytes

```
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 3: Single-threaded usage of GRAPES.

(about 21 kbytes of code), and it needs less than 2 kbytes 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.

```
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 4: Multi-threaded usage of GRAPES.

5 Software Status

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[5].

6 Conclusions and Future Work

This paper described GRAPES, a toolkit for easily and rapidly developing P2P streaming applications. The idea behind the development of GRAPES is that P2P streaming applications are lagging behind their potential impact because of the inherent effort and difficulties in developing the required code.

GRAPES represent a powerful codabase containing basic functionalities encoded with portability, efficiency, and performance in mind, that can be used to reduce the effort for the development of a P2P streaming client by orders of magnitude. Initial testing and development of applications within the Napa-Wine project hint that GRAPES is fulfilling its goals and ambitions, fostering the development team to sustain the effort and improve and enrich the libraries.

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.

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

Acknowledgements

Besides recognizing the support of the Napa-Wine project itself as a source of inspiration and funding, we wish to thank all the Napa-Wine team for the fruitful discussions, the constructive criticism, and for providing additional software modules that can be used in conjunction with GRAPES to develop P2P applications. In particular the teams from Politecnico di Torino, NEC, NetVisor, and Lightcomm efforts for the joint development of Network-Aware, Media-Aware and Network-Wise applications are highly appreciated.

A short version of this Technical Report has been submitted for publication at the 2010 ACM Multimedia Workshop on "Advanced video streaming techniques for peer-to-peer networks and social networking", while a DEMO featuring GRAPES based P2P applications has been accepted for presentation at ACM 2010 Mutimedia International Conference.

References

[1] The NAPA-WINE Project. The napa-wine project home page. `http://www.napa-wine.eu/`.

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

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

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

[5] Luca Abeni, Csaba Kiraly, and Renato Lo Cigno. On the optimal scheduling of streaming applications in unstructured meshes. In *Networking 09*, Aachen, DE, May 2009. Springer.

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

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

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

[9] Chao Liang, Yang Guo, and Yong Liu. Is random scheduling sufficient in p2p video streaming? In *Proceedings of ICDCS 2008*, pages 53–60, Los Alamitos, CA, USA, June 2008. IEEE Computer Society.

[10] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux oskit: a substrate for kernel and language research. In *Proceedings of SOSP '97*, pages 38–51, Saint Malo, France, 1997. ACM.

[11] Robbert Van Renesse. Goal-oriented programming, or composition using events, or threads considered harmful. In *Proceedings of ACM SIGOPS EW98 Support for Composing Distributed Applications*, pages 82–87, Sintra, Portugal, 1998. ACM.

[12] Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In Hans-Arno Jacobsen, editor, *Middleware 2004*, volume 3231 of *LNCS*. Springer-Verlag, 2004.

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

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

[15] Renato Lo Cigno, Alessandro Russo, and Damiano Carra. On some fundamental properties of p2p push/pull protocols. In *Second International Conference on Communications and Electronics, ICCE 2008*, pages 67–73, HoiAn, Vietnam, June 4–6 2008.

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

[17] Luca Abeni, Csaba Kiraly, and Renato Lo Cigno. SSSim: a simple and scalable simulator for p2p streaming systems. In *Proceedings of IEEE CAMAD '09*, Pisa, Italy, June 2009.

[18] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-man: Gossip-based fast overlay topology construction. *Comput. Netw.*, 53(13):2321–2339, 2009.