Execution of the P2PSP protocol in parallel environments

Cristobal Medina-López[1], J.A.M. Naranjo[1], Juan Pablo García-Ortiz[1], L. G. Casado[1] and Vicente González-Ruiz[1]

Resumen— **P2PSP is an application layer protocol for the real-time streaming of multimedia content over the Internet, i.e., where the users playback the stream in a synchronized way. It can be used to build a variety of live streaming services that ranges from small hangouts to large IPTV systems. Unlike in the traditional CS (Client/Server) and CDN (Content Delivery Network) based video streaming, P2P peers contribute with their upload bandwidth to the system. For this reason, in general, P2P systems are much more scalable than those based on the client/server architecture. This work focuses on actual executions (not simulations) of P2PSP networks. Preliminary results regarding buffering time suggest that the protocol scales well up to several hundreds of peers, mainly due to its simplicity.**

Palabras clave—**Peer-to-peer, multimedia streaming, distributed algorithms.**

I. Introduction

P2P video streaming technology has been an intensive research field in the last years with many new proposals appearing. Taking into account the non-commercial ones only, we find Overcast [1], PPLive [2], PPStream [3], CoolStreaming [4], ZIGZAG [5], PRIME [6], AnySee [7], HotStreaming [8], PALS [9], SplitStream [10], AnySee [7], Chainsaw [11], Chunkyspread [12], SPPM [13], DirectedPush [14] and TURINstream [15], including the PPSP IETF proposal [16] and several other approaches without a specific name or acronym. Obviously, P2PSP (Peer to Peer Straightforward Protocol) is another proposal to add to this long list of solutions. However, before describing it we highlight some of the features that make it attractive, specially for the open-source community:

1. P2PSP is not aware of the transmitted content, the bit-rate, the format, etc. Any type of multimedia stream can be transmitted without having to modify the protocol at all.
2. At least one working implementation of the P2PSP can be found in Launchpad, at `https://launchpad.net/p2psp`. Anyone can use/modify/expand it for free as long as the GNU GENERAL PUBLIC LICENSE is observed.
3. P2PSP has a layered architecture. The number of layers used depends on the final requirements of the specific instance.
4. The most basic layer (the broadcasting layer) is simple enough to run the peer process in systems with very low computing resources (for instance, running several threads or forking processes is not needed). The rest of layers add functionality to the protocol, such as parallel streaming, system integrity and information privacy. Of course, layers can be modified or new ones can be added to fulfill the requirements, always keeping the interface between them.
5. If native IP multicast is available (even locally, as happens in most of the local networks), P2PSP can use it, having the same performance as IP multicast.
6. Under unannounced peer churn, the P2PSP provides methods for error concealment in the received stream.[1]
7. Peers can be hosted in private networks, even if they are behind symmetric NATs.
8. P2PSP provides video multiresolution (both, spatially and temporally) and bandwidth-adaptive streaming services by using simulcast, scalable video coding and multiple description video coding.
9. Both, P2PSP and CS/CDN models, can be mixed in order to build massive systems.

II. The Peer-To-Peer Straightforward Protocol

A. Layers

P2PSP is organized in four layers, each one incorporating additional functionality. Figure 1 depicts them.

1. **Broadcasting Layer (BL):** This layer implements the most basic behaviour of the protocol and it has been designed to be efficient in transmitting a data-stream from a source node to the peers of the network.
2. **Integrity Layer (IL):** In some contexts, the network needs to overcome hostile peers which could produce the poisoning of the stream, a denial of service, etc. Those peers will be identified and rejected from the P2PSP overlay by using a set of rules defined in this layer.
3. **Multi-channel Layer (ML):** In contexts where there is sufficient bandwidth available, peers can decide to subscribe to more than one stream (channel) in a given time interval. A peer that implements this layer interacts with different, concurrent instances of the Broadcasting Layer.

[1]Dpto. de Informática, Univ. Almería, Agrifood Campus of International Excellence (ceiA3). Corresponding e-mail: `cristobalmedinalopez@gmail.com`.

[1]Unannounced/non-scheduled peer churn produces a loss of data in the received stream that is spread along time. Fortunately, error concealment techniques can be used easily.

4. **Privacy Layer (PL):** Finally, there is a collection of rules ensuring that the transmitted stream is played only by authorized peers. This layer can be useful to implement, for example, Pay-Per-View (PPV) streaming systems.

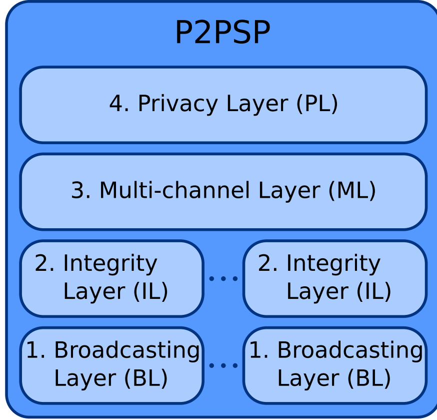

Fig. 1
P2PSP layer diagram.

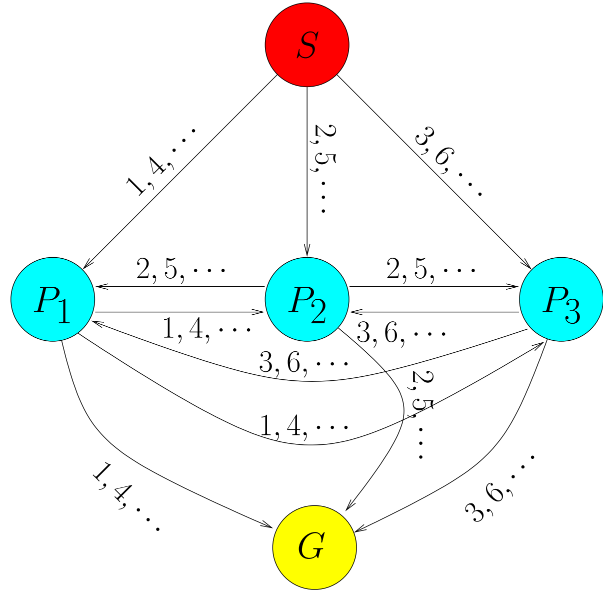

Fig. 2
A P2PSP cluster. Arrows and their labels indicate the transmission of blocks. S sends one different block to one different peer which is selected using a Round Robin model. Peers send each block received from S to each other peer in the cluster.

B. The Broadcasting Layer

This layer defines the algorithms and communication methods governing the entities of a P2PSP network. In order to describe them, it is convenient to make some preliminary definitions.

In the Broadcasting Layer, data can be transmitted in two different states:

1. As a **stream**, i.e., as an endless sequence of data that transports some kind of information. Streams are always transmitted over the TCP.
2. As a collection of **blocks**, being a block a piece of stream. All blocks have the same size. Blocks are always transmitted over the UDP. A small block minimizes the average latency of the transmission but also increments the underlying protocols (UDP/IP/data-link) overhead, and vice versa. In any case, the block size should not exceed the MTU (Maximum Transfer Unit) of the transmission links in order to avoid the overload produced by IP fragmentation.

We also define the following types of entities and groups of entities:

1. **Source (O):** It is the producer of the stream which is transmitted over the P2PSP network. Typically, it is a streaming server using the HTTP protocol such as Icecast [17].
2. **Player (L):** Players request and consume (decode and play) the stream. Usually, several players can retrieve the stream from the source, in parallel.
3. **Splitter (S):** This entity receives the stream from the source, splits it into blocks of the same size and sends the blocks to peers.
4. **Peer (P):** Receives blocks from the splitter and from other peers, ensembles the stream and sends it to a player. Some blocks are also sent to other peers (see below). Notice also that the player and the peer usually run on the same machine.
5. **Gatherer (G):** The gatherer is a special peer that sends blocks neither to other peers nor to the splitter. It only receives a sequence of blocks and (optionally) resends them as a stream to a player. Its function is explained below.
6. **Repeater (R):** Repeaters receive a stream and simply re-transmit it.
7. **Cluster (C):** A cluster is formed by a splitter, a gatherer and zero or more peers (see Figure 2).
8. **Overlay:** An overlay is the union of one or more clusters. These clusters can be connected by repeaters.

B.1 BL rules

1. **Block scheduling:** When the cluster is empty, all blocks are transmitted from the splitter to the gatherer. Otherwise, all blocks are transmitted from the splitter to the peer(s), then among peers and finally to the gatherer (see Figure 2). The splitter sends the n-th block to the peer P_i if
$$(i+n) \bmod |C| = 0, \quad (1)$$
where $|C|$ is the number of peers in the cluster. Next, P_i must forward this block to the rest of peers of the cluster and the gatherer. Blocks received from other peers are not re-transmitted.
2. **Content unawareness:** The P2PSP does not known the nature of the stream. However, at the beginning of the transmission, a stream header

can be transmitted over the TCP in case the player needs it.

3. **The list of peers:** Every node of the cluster (except for the gatherer) knows the endpoint (IP address and port) of the rest of nodes in the cluster. A list is built with this information, which is used by S to send the blocks to the peers. Peers use this list to forward the received blocks to other peers and the gatherer.
4. **IP multicast mode:** If native IP multicast is available, the cluster can be configured to use it. In this case, S uses an IP multicast address and port where all peers of the cluster and the gatherer wait to receive blocks. In this mode, the list of peers maintained by every peer is always empty and stream blocks are not forwarded by them.
5. **Free-riding control in peers:** Each peer assigns a counter to each other peer of the cluster. When a block is sent to a peer, its counter is incremented and when a block is received from it, its counter is reset. If a counter associated to a peer P_i reaches a given threshold, P_i is deleted from the list and it will not be served any more. Notice that the gatherer does not need this mechanism because it does not forward blocks.
6. **Peer arrival:** An incoming peer P_i must contact with the splitter in order to join the cluster. After that, the following steps are carried out:
   (a) The splitter appends P_i to the end of his list.
   (b) The splitter sends to P_i, using the TCP, his list of peers. P_i starts sending to the rest of peers a message (an empty block) in parallel with the reception of the list of peers, in order to introduce P_i to the cluster as soon as possible.
   (c) A peer P_j inserts the peer P_i in his list as soon it receives a message from P_i.
7. **Peer departure:** Peers are required to send a "goodbye" message to S when they leave the cluster, so S can stop sending blocks to them in a natural way. If a peer P_i leaves without notification no more blocks will be received from it. This should trigger the following succession of events:
   (a) The free-riding control mechanism will remove P_i from the list of peers at P_j, $i \neq j$.
   (b) P_j, $i \neq j$, will complain to S about blocks that likely S has sent to P_i, because they have not been received them (the complaint method is explained below).
   (c) S will delete P_i from his list (see rule 11 below).
8. **Buffering:** Blocks in transit can suffer different transmission delays due to jitter[2]. Moreover, they can arrive out of order. For this reason, S numbers every block (see Figure 3) with a 16-bit incrementing positive index. Peers and the gatherer store the received blocks in a buffer.

[2]Variations in network latency.

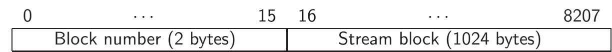

Fig. 3
A BL block. The first field (16 bits) is the block number and the second field, the stream block (in this example is 1024 bytes long).

   The size in blocks of the buffer b is the same in each peer.
9. **Relation between buffer size b and block index upper bound m:** Due to practical reasons, there is an upper bound to the block index (preferably a power of two). In order to minimize the probability of receiving two or more blocks with the same number (remember that blocks can be reordered in transit), m must be a multiple of b verifying $m > b$.
10. **Block tracking:** S remembers the block index sent to each peer among the last b blocks. This is necessary to identify unsupportive peers (free-riding). On the other hand, peers can guess a lost block's index by counting buffer slots since the closest received block in the buffer.
11. **Free-riding control in S:** In an ideal scenario, every peer (and also the gatherer) should receive all blocks sent by S. However, in real P2PSP overlays, blocks can be lost due to a number of causes: noise and physical failures in the networking infrastructure, peers leaving the cluster without notification and what it is more likely, peers suffering a temporary reduction of their upload bandwidth [3]. In any of those cases, the number of lost blocks should be kept as small as possible.
    Peers become aware of a block loss at the time of sending it to the player. When a block is missing, peers send to S a message for specifying its index (obviously, G does the same). Using this information, S finds out unsupportive peers. When peer P_i accumulates more than $(|C|)/2$ complaints then P_i is deleted from the list of peers of S and it is not served any other block. This implies recalculating (1).
12. **Blocks re-transmission:** As it was explained in the previous rule, blocks sent to unsupportive peers will be most likely lost and a large part of the cluster will complain about that. S will detect this problem because it will receive a large number of messages with the same (lost) block index. Under this situation, if enough bandwidth is available, S may resend to peers, using again the Round-Robing pattern, the missing blocks just as if they were new. Note that, due to the fact that peers may be slightly asynchronized, some peers could receive these lost blocks on time and therefore would not complain.

[3]The Internet is a shared medium and therefore the load of the network influences communication capabilities of peers.

13. **Congestion control:** If IP multicast is not available, each peer sends blocks using a constant bit-rate strategy to minimize the congestion of its uploading link. The rate of blocks that arrive to a peer is a good metric for performing this control in networks with a reasonable low packet loss ratio. If multicasting is available, peers do not send blocks to the multicast channel (the splitter does all the work).

C. The Integrity Layer

The IL (Integrity Layer) is responsible for fighting against possible custom implementations of peers that might be specifically designed to attack a P2PSP network. More specifically, this layer serves as a barrier against Denial of Service attacks. We identify and neutralize two main attacks that might end in a service interruption situation.

C.1 DoS by starvation, DoS by report

An attacker (or a pool of attackers) might try to induce a denial of service situation if it joins the cluster but does not send any block. If this happened, all nodes of the cluster would complain about the lost blocks after a given period of time, that depends on the buffer size and the bitrate of the stream. Under this situation, the splitter will remove the malicious peer from the cluster and will reject further connections from the same endpoint.

Note that, when adopting such a report-based system, a complementary attack might be developed too: a pool of attackers might report a well-deliberated peer in order to expel it from the network. To prevent this, the splitter should only accept to take actions against reported peers if more than a fixed number of peers have complained (i.e. establishing a report-honesty threshold). If this threshold is set to a high value (for example, 75% of peers in the cluster), then an attacker should need to virtually control the cluster in order to expel any given victim (see Section II-B, point 11).

With the aim to make these types of attacks more difficult, the splitter allows only one connection from the same IP address. Therefore, an attacker that is behind a NAT will have only one shot (if the NAT does not renegotiate its public IP address). Obviously, this provokes that two or more well-meaning peers that are in the same private network can not connect to the same cluster. An efficient solution to this problem is to create a private cluster for all peers behind a given NAT.

C.2 DoS by stream spoiling

Another possible attack consists on the injection of fake information into the cluster. This can be caused by a custom peer sending poisoned blocks [4]. A way of tackling this problem is by inserting (and removing after a given period of time) one o more "trusted-peers". These peers are authenticated as trusted to the splitter, but not to the rest of peers (the behavior of a trusted-peer is identical to any other peer making impossible for a poisonous peer to know these special peers). The source sends to trusted-peers the hash code of randomly chosen blocks using a reliable transmission protocol (note that the poisonous peer is not aware of which blocks were selected). If a poisonous peer sends an altered block to a trusted-peer, it will detect the change using the hash code and will notify the source, which will eject the poisonous peer from the cluster.

[4] A poisoned block is a block that seems to be OK, but which the sender has changed in such a way that when played, introduces no information (for example, a block filled with zeroes) or even wrong information.

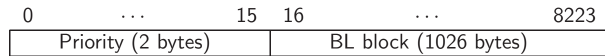

Fig. 4
A ML block is a concatenation of a BL block and a 16-bit priority number.

D. The Multi-channel Layer

P2PSP may broadcast different channels (streams) over distinct (unconnected) clusters. If the user's machine supports multiprocessing then it can run several peers in parallel, having one peer per subscribed channel. To enable this, no extra functionality needs to be added to the BL: the only essential requirement is the network providing sufficient bandwidth.

However, this condition may not be always true. If this happens, one or more (multi-channel) peers could be ejected from a cluster because they are identified as unsupportive peers. In order to minimize this drawback, the ML (Multi-channel Layer) introduces a new encapsulation scheme (see Figure 4) and a new type of node, the Multichannel Scheduler (M).

Peers that implement the ML must label each BL block. This label is a 16-bit positive integer number that represents a priority, being zero the highest one. ML blocks are sent to M which basically implements a FIFO priority queue of blocks. Each time a new ML block is received by M, it sorts them by priority and next, by block number. Thus, if there is not enough bandwidth to transmit all blocks, the user stops receiving those channels that have been assigned a lower priority.

E. The Privacy Layer

The top layer deals with privacy-related issues. Many content providers offer pay-per-view channels as part of their services. From a technical point of view, this implies having a Key Server that ciphers the stream with a symmetric encryption key and delivers such key to authorized members only. However, this is not enough: it is crucial that the Key Server renews the encryption key after the expiration of a peer's authorization period so the stream can not be decrypted any more by the peer (this feature is called *forward secrecy*). In addition, if we want to

play on the safe side then the Key Server should renew the encryption key after a peer purchases an authorization period (if the key remained the same then the peer might decrypt previously captured stream packets for a later playback): this feature is called *backward secrecy*. The associated renewal process is not trivial and is carried out by a *secure multicast protocol*. In order to alleviate the overhead incurred by avalanches of peers entering and leaving the authorized group (for example, at the beginning of a high interest event such as The Olympics) key renewal can be performed on a batch manner, i.e. renewing the key at a given fixed frequency rather than on a per arrival/exit basis. Finally, key renewal messages should be authenticated by means of a digital signature or other alternative methods [18].

Many secure multicast protocols exist in the literature, for example [19], [20], [21], [22] (see [23] for a survey). Here we suggest the implementation of a protocol by Naranjo et al [24]. In it, every authorized peer receives a large prime number from the Key Server at the beginning of its authorization period (this communication is done under a secure channel, for example SSL/TLS). For every renewal, the Key Server generates a message containing the new key to be used by means of algebraic operations: all the authorized primes are involved in this message generation process, and the key can only be extracted from the message by a peer with a valid prime. This protocol is efficient and suits P2PSP architecture in a natural way: every splitter can act as a Key Server for its own cluster. Hence, the stream would be first transmitted among splitters (possibly encrypted with a different key, shared by the splitters). Within each cluster, its corresponding splitter would control the encryption and key renewal process.

III. Experiments

We are currently in the process of testing our aforementioned implementation of P2PSP at Launchpad. The first experiments are being done in one Intel Core i5 CPU 660 @ 3.33GHz × 4 with 4 GB RAM. A single cluster is run, including a splitter, a gatherer and a group of peers, each one being a different thread within the same machine. For now, we are testing the average buffering time of a peer under *flash crowd* conditions (i.e. a large number of peers joining the cluster in a small period of time). By buffering we mean filling half the peer's buffer of stream blocks. Checking this metric for increasing cluster sizes can give an idea of the protocol's scalability. Figure 5 shows the average buffering time for different cluster sizes and different buffer lengths. Buffer lengths are measured in stream blocks, each block being 1024 bytes long. Even though real network latencies do not occur in our experiments, some properties of the protocol can be appreciated. For example, the plot makes clear that peers' buffering time is unaffected by flash crowds and growing audiences, therefore suggesting that our protocol scales well.

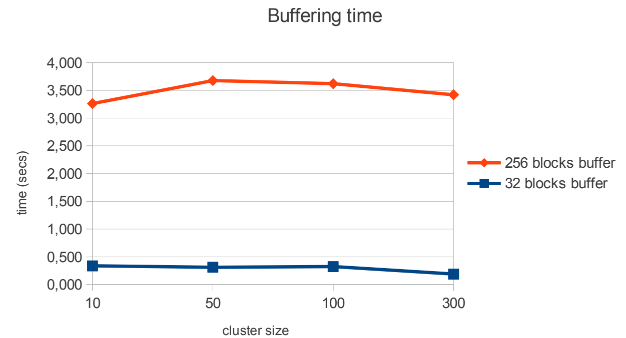

Fig. 5
Buffering time for 32 and 256 blocks buffers.

Our intention is to measure different properties of the protocol, like buffering time, presence of jitter, and quality of service (QoS) under a variety of circumstances (in stable configurations, in flash crowds, with some churn, while splitting and merging clusters, etc). Larger experiments will be made in our computer clusters at the High Performance Computing - Algorithms research group.

IV. Conclusions

Here we presented the P2PSP peer-to-peer streaming protocol. It is composed of a set of roles and rules arranged into four layers, each one with its own different purpose. Undoubtedly, the most important one is the *Broadcast Layer*, since it is at the core of the protocol and defines the rules for the most basic type of communication: one-to-many. The other layers allow to establish many-to-many communications, set different kinds of cluster configuration, provide different types of service (e.g. multilayer encoding for scalable quality, video-on-demand, etc.), assure the integrity of the streamed content and, finally, allow to establish access control rules for privacy-aware scenarios or pay-per-view services. We are in the process of testing the protocol, and show preliminary experimental results regarding the average buffering time of peers with different cluster sizes and buffer lengths. These results suggest that the protocol is scalable. Our goal is to extend our experiments to a high-performance multi-core computer in order to verify the protocol's behaviour in larger clusters.

Acknowledgements

This work has been funded by grants from the Spanish Ministry of Science and Innovation (TIN2008-01117, TIN2012-37483-C03-03, TEC2010-11776-E) and Junta de Andalucía (P10-TIC-6548 and P11-TIC7176), in part financed by the European Regional Development Fund (ERDF).

Referencias

[1] J. Jannotti, D.K. Gifford, K.L. Johnson, M.F. Kaashoek, and Jr.J.W. OŤoole, "Overcast: Reliable multicasting

with an overlay network," in *Operating Systems Design and Implementation*, 2000, pp. 197–212.
[2] "PPLive," http://www.pplive.com.
[3] "Ppstream," http://www.pps.tv.
[4] X. Zhang, J. Liu, B. Li, and Y.-S. P. Yum, "Coolstreaming/donet: A data-driven overlay network for peer-to-peer live media streaming," *Proceedings IEEE INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 3, pp. 2102–2111 vol. 3, March 2005.
[5] D.A. Tran, K.A. Hua, and T. Do, "Zigzag: An efficient peer-to-peer scheme for media streaming," in *IEEE INFOCOM*, April 2003, vol. 2, pp. 1283–1292.
[6] N. Magharei and R. Rejaie, "Prime: Peer-to-peer receiver-driven mesh-based streaming," in *INFOCOM*, May 2007, pp. 1415–1423.
[7] X. Liao, H. Jin, Y. Liu, L.M. Ni, and D. Deng, "Anysee: Peer-to-peer live streaming," in *INFOCOM*, April 2006, pp. 1–10.
[8] J.C. Wu, K.J. Peng, M.T. Lu, C.K. Lin, and Y.H. Cheng, "Hotstreaming: Enabling scalable and quality iptv services," .
[9] Reza Rejaie and Antonio Ortega, "Pals peer-to-peer adaptive layered streaming," in *International workshop on Network and operating systems support for digital audio and video*, 2003, pp. 153–161.
[10] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "Splitstream: High-bandwidth content distribution in a cooperative environment," in *Peer-to-Peer Systems II*, February 2003.
[11] Vinay Pai, Kapil Kumar, Karthik Tamilmani, Vinay Sambamurthy, and Alexander E. Mohr, "Chainsaw: eliminating trees from overlay multicast," in *Proceedings of the 4th international conference on Peer-to-Peer Systems*, Berlin, Heidelberg, 2005, IPTPS'05, pp. 127–140, Springer-Verlag.
[12] V. Venkataraman, K. Yoshida, and P. Francis, "Chunkyspread: Heterogeneous unstructured tree-based peer-to-peer multicast," in *Network Protocols, 2006. ICNP '06. Proceedings of the 2006 14th IEEE International Conference on*, Nov., pp. 2–11.
[13] P. Baccichet, Jeonghun Noh, E. Setton, and B. Girod, "Content-aware p2p video streaming with low latency," in *Multimedia and Expo, 2007 IEEE International Conference on*, July, pp. 400–403.
[14] Guanzhong Xu, Yusuo Hu, Yao Shen, and Minyi Guo, "Directedpush - a high performance peer-to-peer live streaming system using network coding," in *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, Dec., pp. 292–298.
[15] A. Magnetto, R. Gaeta, M. Grangetto, and M. Sereno, "Turinstream: A totally push, robust, and efficient p2p video streaming architecture," *Multimedia, IEEE Transactions on*, vol. 12, no. 8, pp. 901–914, 2012.
[16] IETF, "Peer to peer streaming protocol (ppsp)," http://datatracker.ietf.org/wg/ppsp/charter/.
[17] The Xiph.org Foundation, "Icecast.org," http://www.icecast.org.
[18] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and David E. Culler, "SPINS: security protocols for sensor networks," *Wirel. Netw.*, vol. 8, pp. 521–534, 2002.
[19] Lihao Xu and Cheng Huang, "Computation-efficient multicast key distribution," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 5, pp. 577–587, May 2008.
[20] J. Lin, K. Huang, F. Lai, and H. Lee, "Secure and efficient group key management with shared key derivation," *Comput. Stand. Inter.*, vol. 31, no. 1, pp. 192 – 208, 2009.
[21] Z. Zhou and D. Huang, "An optimal key distribution scheme for secure multicast group communication," in *INFOCOM'10*, 2010, pp. 331–335.
[22] Eun-Jun Yoon and Kee-Young Yoo, "A secure broadcasting cryptosystem and its application to grid computing," *Future Generation Computer Systems*, vol. 27, no. 5, pp. 620 – 626, 2011.
[23] J.A.M. Naranjo and L.G. Casado, "An updated view on centralized secure group communications," *Logic Journal of IGPL*, 2012.
[24] J. A. M. Naranjo, L. G. Casado, and J. A. López-Ramos, "Group oriented renewal of secrets and its application to secure multicast," *Journal of Information Science and Engineering*, vol. 27, no. 4, pp. 1303–1313, july 2011.