A Synchronization Protocol For Supporting Peer-to-Peer Multiplayer Online Games in Overlay Networks

Stefano Ferretti
Department of Computer Science, University of Bologna
Mura Anteo Zamboni 7, 40127
Bologna, Italy
sferrett@cs.unibo.it

ABSTRACT

We propose a new synchronization protocol devised to support multiplayer online games (MOGs) over peer-to-peer architectures. The dissemination of game events is performed through an overlay network. Peers are kept synchronized thanks to an optimistic mechanism which is able to drop obsolete events (i.e., events that lose their importance as the game goes on) and to allow different processing orders for non-correlated events (i.e., events which do not represent competing actions in the virtual world). To allow a fast identification of obsolete events, a gossip protocol is added to the synchronization mechanism, which is in charge of spreading in background information on generated game events. Results coming from extensive simulations confirm the viability of our approach.

Categories and Subject Descriptors

C.2.4 [**COMPUTER-COMMUNICATION NETWORKS**]: Distributed Systems—*Distributed applications*; K.8.0 [**Computing Milieux**]: PERSONAL COMPUTING—*Games*

General Terms

Algorithms, Synchronization, Performance

Keywords

Event Distribution, Synchronization Algorithms, Peer-to-Peer

1. INTRODUCTION

Multiplayer Online Games (MOGs) recently gained more and more attention from the the distributed systems research community. These highly interactive applications represent a tricky case study. Indeed, several concurrent issues must be considered to guarantee compelling game sessions among geographically distributed players. Scalability, responsiveness, consistency, fairness, cheating-avoidance are just some of the main items to cope with [1, 9, 12]. Moreover, the interesting aspect is that practical solutions, devised for the support of MOGs, can be factually employed over a plethora of different contexts, ranging from distributed simulation to e-learning, remote monitoring and telemedicine.

Based on a typical MOG scenario, a massive number of players participates to the same game session. Each player interacts with others by repeatedly generating some events, corresponding to game actions. Due to the real-time nature of MOGs, these events must be responsively delivered to all other players [26, 27]. Moreover, events are typically produced at a very high rate, following the well-accepted rule that the faster the game the more exciting the experience perceived by the players. Then, correct game state advancements must be provided to all participants, thanks to a careful processing of these events.

Needless to say, as the number of players increases, and the demand for interactive and reliable gaming applications augments, also a need grows for new distributed strategies to support these applications.

In this context, it turns out that guaranteeing a massive and responsive dissemination of events is a hard task. This problem is further exacerbated by the fact that, for the sake of scalability and fault tolerance, researchers often suggest to deploy MOGs over distributed architectures where multiple nodes concurrently maintain a redundant game state [9, 12, 14, 24, 38]. Basically, the idea is that a pure peer-to-peer (P2P) approach can be employed where each node in the system locally stores and manages its own copy of the state. Alternatively, hybrid approaches are possible where a subset of nodes is devoted to this task (while other nodes simply act as clients and periodically receive a novel version of the produced game state). Hereinafter, with *nodes* (or *peers*) we will refer only to those nodes in the game system that maintain a local view of the game state. These nodes must be synchronized so as to guarantee a consistent evolution of the game state provided to all the game players.

The general approach of most of these proposed schemes is that nodes are fully connected, thus assuming that a low number of peers is involved in the synchronization. In fact, a fully connected mesh presents a message complexity of the order of $O(n^2)$, if n is the number of peers, i.e., each message is directly sent from one peer to all others. As a consequence, these solutions are extremely impractical for

DEBS '08, July 1-4, 2008, Rome, Italy

large networks.

Conversely, application-level overlay networks represent viable solutions for developing scalable event delivery strategies. Indeed, these schemes reduce the message complexity, by "mimicking" at the application layer the general idea of multicast approaches of minimizing the amount of messages sent through a given network link. Thus, the adoption of an application-based overlay network, as the infrastructure for disseminating game events produced by players, could represent a scalable choice when designing synchronization protocols for P2P MOGs. Of course, a careful management of the overlay is needed, so as to ensure that game events are delivered to *all* nodes in a timely fashion and that the game advances in real-time. This entails that, for example, in certain MOGs it is not possible to adopt a simple gossip delivery scheme that, despite its scalable and in general rapid (for a majority of peers) dissemination of events, simply guarantees that eventually all peers will receive a given message [10, 19, 32]. Thus, structured approaches would probably provide higher delivery guarantees for the support of MOGs.

Several works have been devoted at building resilient and effective application level multicast schemes over P2P networks [5, 7, 29, 34, 40]. Taking inspiration to these seminal approaches, some proposals have adapted these techniques to develop scalable P2P MOGs, e.g., [1, 16]. However, none of these approaches was able to exploit the specific characteristics of MOGs to hasten the event distribution and make the peer synchronization highly responsive.

This paper discusses a new optimistic synchronization scheme among peers connected through an overlay network. Trying to control and limit the work to synchronize all peers, hence augmenting the responsiveness degree of the overall system, the adopted protocol exploits the semantics of game events, thanks to two specific properties, i.e., correlation and obsolescence [12].

Stated simply, two events are said to be correlated if the final game state depends on their execution order. Correlated game events represent significant interactions among game events which require a careful processing in order to obtain a correct evolution of the game state. As a matter of fact, to maintain the consistency of the game state, only correlated events are required to be processed in the same order at all the nodes (correlation-based order). Instead, different delivery orders are possible at different nodes when a given sequence of non-correlated events is considered.

Obsolescence, instead, enables peers to drop those game events that lose their importance during the game evolution, thus relaxing the reliability constraints of the delivery protocol [14, 26, 30]. The idea is that an event is considered as obsolete when a fresher event produced by the same player annuls the importance of the previous one. It has been demonstrated that discarding superseded events for processing fresher ones may be of great help for delay-affected nodes, without introducing inconsistencies on the distributed game state computation [12]. Obviously, not all events may become obsolete in MOGs. For instance, concurrent events generated by two different players, which represent competing interactions among players (e.g., two concurrent attempts to gain a given virtual game element), are not supposed to become obsolete (processing just one of two events, at a given peer, may lead to a different game state reached by other ones) [12, 14].

Based on these two properties of game events, the synchronization protocol works as follows. Events are disseminated through the overlay. The delivery procedure is deterministic, i.e., each event received by a peer is forwarded to all its neighbors, excepting the node from which the message was received. When an event is identified as obsolete it is discarded, thus reducing the amount of events travelling through the overlay. Concurrently, received events are processed by resorting to an optimistic synchronization protocol. Basically, non-obsolete events are optimistically processed as soon as they are received at a given peer. Then, in the case that correlated events have been processed out of the correlation-based order, a rollback procedure is invoked to obtain a correct game state computation. Obviously, the rollback procedure involves only out-of-order, non-obsolete correlated events.

The effectiveness of the proposed protocol is strongly related to the ability of the scheme to identify obsolete events. As a matter of fact, this could be not an easy task in a highly distributed overlay network. To ease the identification of obsolete events, according to our scheme additional information on last generated game events (but not the events themselves) is piggybacked within messages exchanged among nodes. Not only, an additional lightweight gossip protocol is exploited to spread information about generated game events. The idea is that, since events travel through the overlay, when network traffic and congestion slow down the forwarding activity of game events, the gossip protocol (which connects peers that do not communicate through the overlay) can be of help to promptly identify those events that become obsolete. This means that they can be dropped (or not waited), thus reducing the delivery work at different peers, hence gaining responsiveness.

Our simulations demonstrate that this approach improves the performances of the synchronization protocol and represents a good candidate to deliver game events in overlay P2P networks.

The remainder of this paper is organized as follows. Section 2 introduces the basic definitions and the system model at the basis of our protocol. The section also discusses on how to model time advancements in P2P MOGs. Indeed, without the cental control of a server, due to the best effort nature of the network and the drift rate of peers' clocks, the timestamping activity of game events must be carefully managed. Section 3 provides a discussion on the correlation and obsolescence properties, with a description on how it is possible to factually implement these two properties within the proposed synchronization protocol. Section 4 presents in detail the synchronization protocol. Section 5 reports on a simulation we performed, which confirms the viability of the proposed scheme. Finally, Section 6 concludes the work.

2. SYSTEM MODEL AND BASIC DEFINITIONS

2.1 System Architecture

We suppose that games are deployed based on a P2P architecture. Each peer maintains a local view of the game state and updates it, based on game events generated and notified by all peers in the game. We denote with Π the set of peers; p_i identifies a single peer, i.e., $p_i \in \Pi$. (Notations and symbols employed in the remainder of the work are summarized in Table 1.)

Peers exchange game events through an overlay network. It is not the aim of this paper to propose a specific scheme to build such overlay. Many solutions exist to structure a P2P overlay network, such as those presented in [20, 32, 33, 36, 37, 39]. For the sake of a simpler discussion of our scheme, we assume that the overlay is structured as a tree. (The adaptation of our synchronization protocol over more complex mesh-based overlays requires some simple modification on the message forwarding strategy.) Thus, we assume that some scheme is employed to build a spanning tree such as those presented in [4, 23], or algorithms utilized for the tree construction in end system multicast approaches [7, 29]. N_p identifies those peers which are neighbors of p in the tree structure, i.e., N_p is a set composed of the parent and the child nodes of p.

Game events are notified within messages. Typically, MOGs exploit UDP-based delivery solutions to transmit game events [26]. Indeed, it is well known that TCP-based synchronization protocols should be avoided while developing distributed games [25]. The motivation is that much of TCP's behavior, such as congestion control and retransmissions, is detrimental to meeting the MOGs real-time constraint. According to our protocol, reliability is guaranteed (when necessary) by exploiting a receiver-initiated communication protocol that utilizes NACKs (Negative ACKnowledgments).

As to game events, sometimes we use subscripts to characterize different game events, e.g., e_j, e_k, $j \neq k$. Conversely, prime notations (e.g., e^l) denote that the event has been generated by a specific peer (in this case p_l).

2.2 Modeling Game Time

In the following, we present a framework to characterize gaming advances, which is exploited by our synchronization protocol. The game time evolution is modeled according to a real-time based scheme, so that each advance in the game is paced to occur in synchrony with an equivalent advance in the time of execution [13, 15].

The framework is based on the concepts of simulation and wallclock times [13, 15]. *Simulation time* is the abstraction that is used to model when events have been produced within the virtual game timeline. As soon as an event is produced at a given peer, a simulation time is associated, which represents when the event would occur if the virtual world would be the real world. The simulation time measured at p_i is denoted with ST_i. With some abuse of notation, $ST_i(e)$ represents the simulation time associated to the game event e, generated by p_i.

Simulation times are produced by managing wallclock times. *Wallclock time* corresponds to the physical clock measured at a given node. In this paper, WT_i represents the wallclock time measured at p_i. $WT_i(e)$ identifies the wallclock time of generation of the game event e at p_i.

Table 1: Notations and Symbols

Symbol	Definition
e_k^i	k-th event generated by p_i
N_p	Neighbors of p
p_i	Single peer
Π	Set of peers
ST_i	Simulation time at p_i
T_i^S	Mapping function from WT_i to ST_i
WT_i	Wallclock time at p_i

In our model, simulation time advances in synchrony with the wallclock time. A scaled real-time factor k may be exploited to determine the pace of game advancements in the simulated world [8, 13, 15]. In substance, wallclock times WT_i are mapped into simulation times ST_i based on a mapping function T_i^S defined as:

$$T_i^S(t_{i,actual}) = T_i^S(t_{i,start}) + k(t_{i,actual} - t_{i,start}), \quad (1)$$

where $t_{i,actual}$ is the actual wallclock time at p_i, $t_{i,start}$ denotes the wallclock time when p_i begins the game session. The value of the simulation time $T_i^S(t_{i,start})$ is a value agreed and shared by all nodes, representing the simulation time of the beginning of the game, i.e., $T_i^S(t_{i,start}) = s_{start} \in ST$, $\forall p_i \in \Pi$.

A game session initialization phase is devoted to distribute among all nodes, all the wallclock times $t_{i,start}$, which are associated to the starting simulation time, i.e., $T_i^S(t_{i,start}) = s_{start}$. Based on (1), the simulation time of a given game event e is thus measured as:

$$ST_i(e) = s_{start} + k(WT_i(e) - t_{i,start}). \quad (2)$$

Most distributed approaches often assume a perfect synchronization among physical clocks' nodes. This assumption is quite strong and met with difficulty. Moreover, another utopian assumption is that in a P2P MOG scenario, nodes simultaneously start the game. In practice, despite methods may be set to avoid gaps among start times at different peers, it is more likely that $t_{i,start} \neq t_{j,start}$, $\forall p_i, p_j \in \Pi$, $i \neq j$. The adopted approach solves these two issues. Indeed, despite drifts among clocks' nodes and different starting times, the use of simulation time, as far as it is defined, allows a fair method to managing timing properties of game events and totally ordering events during the synchronization protocol.

3. OBSOLESCENCE AND CORRELATION

Our synchronization protocol is based on two notions of obsolescence and correlation of game events. Here, we outline the main characteristics of these notions. Details on the formal specification of these notions, as well as viable solutions to factually implement them can be found in [12, 14].

Correlation regards the ordering of execution of game events and the final result that is produced, based on the adopted order. Two events are said to be correlated if the final game

state depends on their execution order. Basically, correlation allows to identify all those events that should be carefully processed according to an agreed order at all nodes, to guarantee that each peer (eventually) reaches a game state which is consistent to other states at all other nodes. Such a correlation-based order is a partial order, which requires that all correlated game events are processed based on the order imposed by the simulation time associated to game events. Instead, ordering constraints can be relaxed for non-correlated game events.

Obsolescence is exploited to relax the reliability requirements of the delivery process. Indeed, it has been observed that depending on the semantics of the game, some events usually lose their importance as time passes, due to fresher events which make the previous ones superseded [14, 30]. For instance, consider a rapid succession of movements performed by a character in the virtual world; in this case, the event representing the last destination may supersede the previous ones. The interesting thing here is that obsolete events can be dropped, in the case that these are not responsively delivered at some nodes, without affecting the consistency of the game state. Discarding strategies can hence be employed to speed up the delivery and synchronization procedures at slower peers and make them more responsive.

However, not every event in a game may be considered as obsolete [12, 14]. There are in fact important information that should be never discarded, for the sake of consistency, e.g., in a war game, an event whose effect is the damage or the destruction of a virtual game element. Similarly, correlated events, generated at different nodes within a short (simulation) time interval, and which represent important interactions among players (e.g., two competing actions representing the attempt the gain some virtual game element), should never be considered as obsolete events.

As a matter of fact, it results that viable implementation strategies exist to characterize obsolete and correlated events [12]. Indeed, each event acts on a subset of virtual game elements, possibly updating their values. This information, encoded within the event being distributed, can be utilized to understand whether events are made obsolete by newer ones, or whether certain events are correlated. For example, virtual game elements can be identified through some kind of tag, or minimal data structure (e.g., an id, a flag on a bit-field), encoded and added within the message being distributed. Thus, two events generated by different players that share the same tag are identified as correlated events. Instead, a given event e_j^i is considered as obsolete when a subsequent one exists $e_k^i, k > j$, with the same associated tags, and no other event e^l correlated to e_j^i has been generated by some other player, with $ST_i(e_j^i) < ST_l(e^l) < ST_i(e_k^i)$.

Another delicate issue is that, since we assume the use of an unreliable, best-effort network, message losses and misorderings may occur during the event distribution. This could be an obstacle to identify obsolete events; indeed, it is not possible to assert that an event is obsolete, until it sure that no events correlated to it have been generated. To face this possible problem and speed up the obsolescence identification activity, each peer includes within each transmitted message a small report, which basically piggybacks information stating which virtual game elements have been involved in "recent" actions generated by peers.

It is worth mentioning that similar strategies are employed in different contexts such as forward error correction schemes for streaming applications [2, 31, 35], and multicast communication protocols [17, 21, 28, 30]. Of course, due to message size limitations, reports cannot include a complete history of each peer's actions. However, lessons learned from real case studies demonstrated that there is room to include sufficient information within each message [12]. Indeed, events generated in MOGs, and generally in all distributed simulations, are typically smaller than 200 Bytes, while the maximum size of an IPv4 datagram is 65535 Bytes, and IPv4 datagrams of 1500 Bytes are transmitted in Ethernet networks. Thus, there is room in game messages to carry these additional reports, without increasing the transmission delays among peers. Moreover, optimized compression strategies can be exploited to encode and further reduce the information on correlation (and obsolescence) among generated events.

Summing up, each message distributed by each peer is composed of the following data:

$$\langle p_i, e, sn_e, ST_i(e), WT_i(e), rep \rangle, \tag{3}$$

where p_i is the process that generated the event, e is the event, sn_e is a sequence number exploited to identify possible holes in the received event trace, $ST_i(e), WT_i(e)$ are, respectively, the simulation and wallclock times associated to e, *rep* is the piggybacked information on last generated events.

For the sake of completeness, other messages are distributed based on a gossip subprotocol we employ in our approach. These messages simply contain reports, as discussed in the next section.

We conclude this section by mentioning that the adopted approach represents a general, easy to implement, and effective way to dynamically characterize the semantics of events generated during a game session. Moreover, the scheme can be employed for any given game; the game developer is only required to add within delivered messages some additional tags derived from the transmitted events [12].

4. SYNCHRONIZATION PROTOCOL

This section presents our synchronization protocol for the support of MOGs on overlay networks. It guarantees that peers pass through consistent game states during the event processing activities. This is accomplished by guaranteeing a correlation-based order processing for those events that are not discarded due to obsolescence.

The protocol is composed of different subprotocols that act together. In particular, a delivery protocol specifies how the communication among peers is accomplished. A game state update protocol specifies how events are processed at all peers to compute consistent game state advancements. A gossip protocol is added to the basic scheme, which is in charge of distributing reports on events produced by dis-

tributed peers, so as to facilitating the identification of obsolete events. Finally, a classic protocol for identifying faulty nodes is exploited.

The idea is that a deterministic delivery scheme is utilized to distribute game events, through an overlay network. This allows to ensure that generated game events will reach all other peers in a bounded number of hops (provided that reliable communications can be established and that the event does not become obsolete). This is an important factor in MOGs, since information about all peers is needed to ensure that the game may evolve correctly. Concurrently, a randomized gossiping protocol allows to spread additional information, which is useful to identify obsolete events, thus hastening the synchronization and the delivery of game events through the overlay network.

4.1 Delivery Protocol

The pseudo-code of the delivery protocol is reported in Figure 1. Basically, the protocol reacts to different situations, i.e., when:

1. a generated event must be sent to other peers. In this case, the SENDEVENT() procedure is invoked;
2. an event is received. In this case, the RECEIVE() procedure is invoked;
3. a received event must be forwarded to other peers. In this case, the FORWARD() procedure is invoked;
4. a NACK message is received. In this case, the RECEIVENACK() procedure is invoked;
5. a NACK message must be sent to a given peer. In this case, the CHECKNACK() procedure is invoked.

The delivery activity for an event starts when the event is produced at a given node, i.e., the player generates an event. In this case, the SENDEVENT() procedure is invoked, which simply consists in sending the event to all the neighbor peers (lines 1-4). The transmitted message contains also a report (GENERATEREPORT() procedure, line 1).

The reception of a game event is managed by the RECEIVE() procedure (see Figure 1). Basically, the event (say e_i) is extracted from the received message, as well as the report (lines 5-7). The information contained within the report, together with the event itself, is exploited to understand if these new data make obsolete some events (which are late or waiting to be processed, line 8). As to e_i, a check is executed to detect whether e_i is obsolete (line 9). In the positive case, it is simply discarded (line 10). Otherwise, the event is forwarded to other peers (line 12), the game state is updated (line 13, see the subprotocol presented in Section 4.2) and a check for the need to send NACK messages is performed (line 14).

The FORWARD() procedure simply consists in forwarding the received message to all the neighbor peers, excepting the one that sent it (lines 16-18).

CHECKNACK() is employed to guarantee a reliable delivery of non-obsolete events. As soon as a receiving peer detects that a non-obsolete event is missing, a NACK is sent back to the sending peer. In substance, a list is computed which is comprised of all non-obsolete, not yet received events generated by the same peer that sent e^i, i.e., p_i (line 19). Then, the neighbor peer p^* in the overlay, which forwards the messages from p_i is identified (line 20). A NACK is sent to p^* to request all events in the list (line 21). Needless to say, such request should be repeatedly accomplished, after a specific timeout, until the missing events are received or become obsolete (such control is omitted in the pseudo-code, for the sake of conciseness).

When a NACK is received at a given peer, the RECEIVENACK() procedure is executed. It is worth noticing that nothing has to do if the peer is not the one which has generated the event (i.e., it is a forwarder of the considered event), the requested event is not obsolete and the local peer is waiting for that event too. Indeed, this means that the event has already been requested through some NACK message. The first line of the RECEIVENACK() procedure does this check (line 22). Conversely, upon request for an event e^i, the newest event e_f is selected among e^i and all those events (if any) that make e^i as obsolete (lines 23-24). Then, e_f is sent to the requesting peer (line 25).

UPDATEOBSOLESCENCE() is in charge of establishing whether some event can be identified as obsolete. Basically, the peer exploits the report to obtain new information on game events which have been (recently) generated at other peers, even if it has not yet received all these events. Using this data, it is possible to build a cut (again, even if not all events have been received; line 27); for all events within the cut, it is now possible to identify whether some among these ones can be considered as obsolete (lines 28-30) [12]. This enables the peer to discard obsolete events, or to avoid their request through some NACK transmission. Moreover, the peer does not have to wait for not-yet-received obsolete events. Clearly, this approach may augment the responsiveness of the synchronization protocol.

Finally, the GENERATEREPORT() procedure is in charge of generating a report containing information on those events with an associated simulation time contained within a simulation time interval of size Δ_{ST} (lines 31-33).

It is worth mentioning that the amount of information which can be contained in a single report, together with the best way to encode this information, is game-dependent. For this reason, the algorithm does not provide any specific suggestion concerning this issue. The interested reader may find a detailed discussion in [12].

4.2 Game State Update Protocol

The pseudo-code of the protocol for updating the game state is reported in Figure 2. In particular, the algorithm is based on an optimistic strategy inspired by the Time Warp algorithm [18].

Simply stated, when an event e must be processed, a check is executed to verify whether there exist other already processed events e_j (correlated to e) with a simulation time larger than $ST(e)$ (line 1). For each event e_j meeting this condition, the system invokes a rollback procedure which is

DELIVERY PROTOCOL:

```
procedure sendEvent(e)
1  rep = generateReport();
2  msg = createMessage(e, rep);
3  for each p_i ∈ N_p
4    send(msg, p_i);

procedure receive()
5  msg := receiveMsg();
6  e_i := extractEventFromMsg(msg);
7  report := extractReportFromMsg(msg);
8  updateObsolescence(report);
9  if (e_i is obsolete)
10   drop(e_i);
11 else {
12   forward(msg);
13   updateState(e_i);
14   checkNack(e_i);
15 }

procedure forward(msg)
16 p_s = sender of msg;
17 for each p_i ∈ N_p\p_s
18   send(msg, p_i);

procedure checkNack(e^i)
19 eList = { e^i_k | e^i_k not yet received ∧ ST_i(e^i_k) < ST_i(e^i) ∧
                     e^i_k not obsolete};
20 p* = neighbor that forwards events coming from p_i;
21 sendNack(p*, eList);

procedure receiveNack(e^i, p)
22 if ¬(p_i ≠ localhost ∧ e^i not obsolete ∧ waiting for e^i) {
23   eList = e^i ∪ {e^i_k | e^i_k makes e^i as obsolete};
24   e_f = select e_* ∈ eList | ST(e_*) > ST(e_k), ∀e_k ∈ eList;
25   send(e_f, p);
26 }

procedure updateObsolescence(report)
27 cut = build a cut using the current global state, e and report;
28 for each e_i in cut
29   if (e_i is obsolete)
30     mark e_i as obsolete;

procedure generateReport()
31 eList = {e_i | ST{e_i} ∈ {ST_current − Δ_ST, ST_current}};
32 rep = createReportFromEventList(eList);
33 return rep;
```

Figure 1: Delivery Protocol

GAME STATE UPDATE PROTOCOL:

```
procedure updateState(e)
1  eList := {e_j | e_j already processed ∧
                   e_j correlated to e ∧ ST(e_j) > ST(e)};
2  if (eList ≠ NULL) {
3    rollback(eList);
4    process(e);
5    for (each e_j ∈ eList)
6      if (e_j is obsolete)
7        drop(e_j);
8      else process(e_j);
9  }
10 else process(e);
```

Figure 2: Game State Update Protocol

based on the standard incremental state saving technique (lines 2-3) [15]. At this point, e_i is processed (line 4), followed by the execution of all those rolled back events which have not become obsolete (lines 5-8). Indeed, obsolete events are discarded during the rollback. If the check fails, thus implying that no rollback procedure is needed, then e is directly processed (line 10).

This scheme guarantees that game state consistency is preserved. In particular, the scheme does not guarantee that peers have the same vision of the game state at the same instant. Rather, it guarantees that peers eventually compute game states which are reached by other peers, possibly at different wallclock times, without incurring in erroneous game state computations.

Clearly, such processing approach reduces the amount of rollbacks executed during the game processing activity with respect to a classic Time Warp mechanism [18]. Indeed, a rollback is performed only when a late event is received that cannot be considered as obsolete. Instead, if obsolete, the event is dropped without invoking the rollback procedure. Moreover, a rollback is needed only if correlated events have been processed out of order and, even in this case, only non-obsolete events are reprocessed. In essence, our scheme has the positive effect of reducing the computational burden typically required to maintain the state consistency.

4.3 Gossip Protocol

The synchronization protocol, as explained above, is able to work properly by simply resorting to the delivery and game state update subprotocols. As a matter of fact, it is evident that a main feature of the scheme is the ability to responsively identifying obsolete events. Indeed, this allows to drop superseded, late events and speed up the synchronization among peers. The problem here is that, due to the use of an overlay to distribute events, distant peers (in terms of number of hops) may have fresh events coming from the other peers only after different hops. This suggests that the use of an overlay-based forwarding strategy may prevent to promptly identifying obsolete events.

With the aim to solve this possible limitation, we have added to our synchronization protocol a gossiping scheme, devoted to randomly distribute reports stating which kinds of events have been recently generated. This choice is due to the fact that it has been demonstrated that gossip protocols are able to quickly spread information through large dynamic

systems [10, 19, 32]. The pseudo-code for this protocol is reported in Figure 3.

This is basically a push gossip protocol, i.e., each time interval a given peer autonomously selects another peer and sends to it a message, without any request for acknowledgments or responses. This lightweight protocol allows to keep a very low message complexity, with each peer that is involved in a single transmission for each time interval. Hence, the protocol does not affect the performances of the delivery subprotocol.

Since the aim of this subprotocol is to hasten the spreading of information that will be eventually distributed through the overlay (thanks to the delivery protocol for the event dissemination), each peer p_i running the gossip protocol tries to establish interactions with peers that are not neighbors in the overlay (pushReport() procedure, lines 1-5 in Figure 3), i.e., a random peer is chosen among the $|\Pi| - |N_{p_i}| - 1$ not neighbor peers (selectPeer() procedure, lines 6-8). Then, a report is generated, which contains information on last generated game events (line 3, generateReport() procedure already described in Figure 1). (As previously mentioned, the amount of information which can be contained in such report message strongly depends on the MOG we are considering [12].) The message is then sent to the selected peer (line 5).

Once a report is received (procedure receiveReport(), lines 9-11), the report is extracted from the message, and the updateObsolescence() procedure is invoked, which acts as previously detailed (see Figure 1).

The idea is that, upon reception of this message at a given peer, the contained information can be rapidly disseminated to its neighbors through the overlay, since these new data can be added to the reports included within messages transmitted through the overlay, based on the delivery subprotocol.

```
Gossip Protocol:

procedure pushReport()
1  sleep(Δ_WT time units);
2  p = selectPeer();
3  rep = generateReport();
4  msg = createMessageFromReport(rep);
5  send(msg, p);

procedure selectPeer()
6  p_local = localhost;
7  p = select random peer in Π \ {N_{p_local}, p_local };
8  return p;

procedure receiveReport()
9  msg := receiveMsg();
10 report := extractReportFromMsg(msg);
11 updateObsolescence(report);
```

Figure 3: Gossip Protocol

4.4 Fault Tolerance Protocol

For the sake of completeness of the protocol, we report in this section a brief discussion on fault tolerance issues.

The tricky case to consider here is when faulty nodes unintentionally leave the system, due to node/network failures. Indeed, from an application point of view, usually players intentionally leave a game only after the end of a game session. This situation can be handled by the system without difficulties, since the game is paused while the overlay is reconfigured.

From a communication point of view, instead, node faults represent an important issue to cope with. These faults basically require that: i) the overlay is restored so as to avoid network partitions, and ii) the membership list of game participants is updated at each peer, so that the player is removed from the game and other peers will not include the faulty process in the peer list, during the random peer selection of the gossip protocol.

As a matter of fact, a plethora of proposals exist which are able to eventually detecting node failures, dynamically reorganizing the overlay, updating the view of the system at each node [6, 22, 32]. We simply assume that one of these solutions is exploited, which allows to update the membership list of active nodes and to coordinate 2^{nd} neighbors to reorganize the overlay tree, as soon as a node has been detected to be failed. In this case, one among the child nodes in the tree should be selected to become the new parent, and connection with other peers must be updated.

5. EXPERIMENTAL EVALUATION

This section presents some results from a set of simulations we assessed, trying to understand the behavior of our protocol.

During the experimentation, we basically contrasted four different protocols:

1. the presented protocol with the gossip subprocotol activated (referred in the charts as *Gossip*);
2. the presented protocol without the gossip subprotocol (referred in the charts as *No Gossip*);
3. a classic optimistic synchronization protocol working without dropping obsolete events (referred in the charts as *Time Warp*).
4. a conservative synchronization protocol, which forces peers to "hear" from all other nodes, before advancing the computation of the game state (referred in the charts as *Conservative*).

We run several simulations varying different configuration parameters. For each configuration, about forty different simulations where run (here, we report the averaged outcomes). We varied the number of peers, nodes' degree, mean and standard deviation of network latencies. Since the outcomes of these tests were quite similar in terms of significance, the reported results relate to one single setting. Specifically, when not differently stated, a configuration is

considered with a number of peers equal to 150, an average network latency equal to 150 msec, a standard deviation set equal to 10 msec. We simulated a best-effort, non-reliable network. Following the literature [3, 11], the transmission delay for each event was obtained based on a lognormal distribution. We varied the packet loss probability; when not differently stated, this value was set equal to 0.15.

Regarding the frequency according to which events were generated at a given node, we shaped the event generation by means of a lognormal distribution whose average interdeparting time was set equal to 40 ms, with a standard deviation of 10 ms. This configuration represents a scenario of intense load traffic [3, 11, 26].

We assumed a topology where peers have an equal node degree (i.e., the number of neighbors of a given peer). Hence, focusing on tree-based overlays, in our simulation we exploited k-ary trees, varying the value of $k = |N_p| - 1$. In this case, the default value for the node degree is equal to 4.

During each simulation, each peer generated thousands of events, to be delivered to other peers. When not differently stated, we considered different event trace configurations where the probability that an event is non-correlated to other events was set equal to 0.9. Clearly, the higher the probability of non-correlation, the higher the number of events that will become obsolete during the game evolution. Indeed, a higher non-correlation probability entails more chances for an event e_j to make obsolete a preceding one e_i. In particular, it is more likely that no events e_k have been generated by other players which break the obsolescence relation among e_j and e_i. Setting the non-correlation probability equal to 0.9 represents a realistic scenario for a vast plethora of possible games. Indeed, we claim that this particular configuration is a good approximation of a real trace of game events as generated during a game session. For instance, in many first person shooter games, independent (i.e., non-correlated) movement actions performed by different characters largely surpass critical movement/shoot ones [26].

During our evaluation, we concentrated our attention on a given node, and measured:

1. the average processing time, i.e., the average time interval after which game events are processed at peers, when resorting to the contrasted protocols. This measure depends on the time to distribute game events and on the additional time required to synchronize peers;

2. the average amount of time needed by the peer to identify game events as obsolete, depending on the employed delivery protocol (i.e., with or without the gossip subprotocol);

3. the percentage of dropped events at the peer, depending on the employed delivery protocol (i.e., with or without the gossip subprotocol);

4. the amount of rollbacks the game state update protocol is subjected, when an optimistic approach is utilized. A lower rollback ratio implies a lower local computational overhead, with a consequent increment of the responsiveness degree at each node.

Figure 4 reports the average processing times experienced by the different protocols to distribute and process all the (non-dropped) game events over the network, during the peers' synchronization activity. As shown in the figure, protocols that exploit the correlation and obsolescence controls outperform others ones. Overall, on average the protocol reduces the processing times of more than the 60%, with respect to *Conservative*, and 12%, with respect to *Time Warp*. This means that our synchronization protocol is able to speed up the delivery and processing of game events, thus improving the responsiveness of the system. Moreover, the scheme with the gossip subprotocol activated performs better than the one without the gossip subprotocol, thus demonstrating that having a quicker view of those events that become obsolete helps in distributing fresh information. We do not report here results concerned with different average and standard deviation values for the network latencies among peers. In any case, the obtained results confirm our mentioned conclusions.

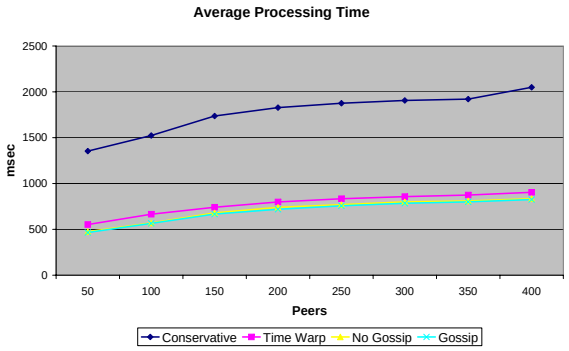

Figure 4: Average Processing Time as a Function of the Number of Peers

A main consideration is that on average, processing times result to be above the classic settings for responsiveness thresholds, i.e., latency intervals which represent the limit to guarantee fluid and interactive gaming experiences to distributed players. Indeed, most studies identify limits on network latencies which may vary from 150 to 300 msec, depending on the type of games [3, 11, 12, 26]. In point of this consideration, however, we recall that in our simulation, the average network latency to perform a single hop was set equal to 150 msec (a high value in current high speed networks, e.g., it corresponds to the average latency to ping Rice University, Texas U.S.A., from the University of Bologna, Italy), and that more than one hop is needed to cover the whole overlay. This observation is further confirmed if we look at the average delivery times due to distribute game events among peers. Figure 5 shows such metric, depending on the number of peers, when different delivery protocols are exploited. The main result here is that, by comparing the two charts, it is possible to argue that our

scheme introduces a very low overhead needed to synchronize peers. In the figure, the *No Dropping* refers to both *Conservative* and *Time Warp*, since these two schemes do not differ in the delivery algorithm (i.e., events are sent to all nodes through the overlay and obsolete events are not discarded during the transmission).

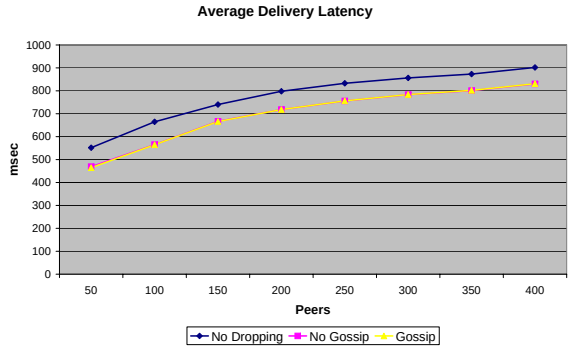

Figure 5: Average Delivery Latency as a Function of the Number of Peers; *No Dropping* Refers to a Delivery Scheme Where No Events Are Dropped Due to Obsolescence

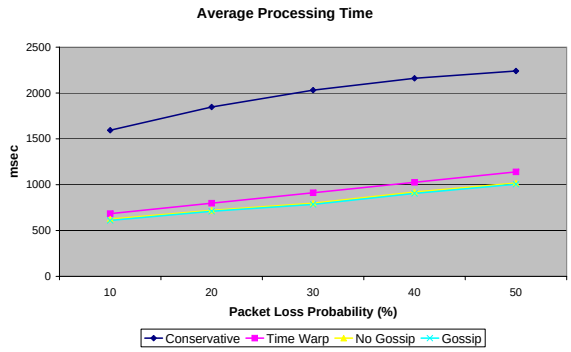

Figure 6: Average Processing Time as a Function of the Packet Loss Probability

Similar arguments on the good behavior of our protocol are valid also when the probability of packet loss is varied, as shown in Figure 6, or when we increment the node degree (i.e., number of neighbors, see Figures 7 and 8).

As concerns results reported on Figure 7, it is worth noticing that since the number of peers remains constant during these experiments (i.e., 150 peers), average processing times diminish when the node degree grows. This is due to the fact that a lower number of hops is needed to spread game events through the network. The small diameter of these settings is also responsible for the limited improvements of

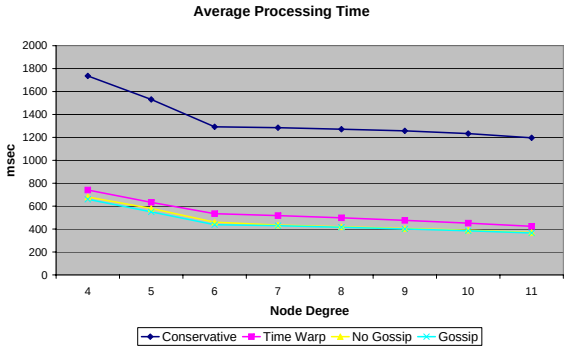

Figure 7: Average Processing Time as a Function of the Node Degree; Number of Peers = 150

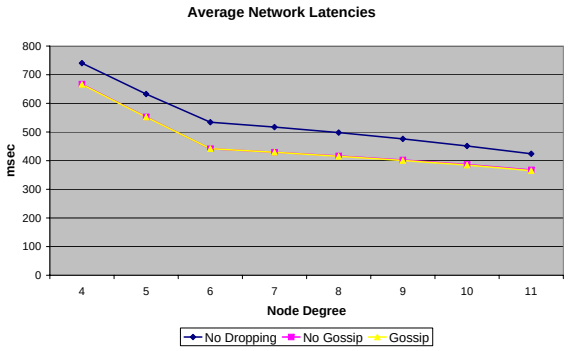

Figure 8: Average Delivery Latency as a Function of the Node Degree; Number of Peers = 150

the *Gossip* scheme, with respect to *No Gossip*.

Figure 9 shows the percentage of events dropped by our protocol, with and without the gossip subprotocol activated. In this case, the scheme that exploits the gossip subprotocol is able to drop more obsolete events, thus demonstrating that spreading information outside the overlay allows to have a fresher view of the game state more rapidly. As already mentioned, we made also several tests with different average network latencies among nodes, as well as different node degrees. We do not show results here, since also in this case, the trend of the obtained results leads to the same conclusions.

It worth observing that in our simulations, we have assumed that the event forwarding process consists in a negligible amount of time and that buffers employed to store events to be forwarded (and processed) are not bounded. Conversely, in a real context where buffers are bounded and the forwarding procedure is not instantaneous, the amount

of dropped events at forwarding peers gains even more importance. Indeed, dropping events waiting to be forwarded allows to speed up the delivery of fresher events in the forwarding buffer. At the same time, it reduces the probability of losing events due to the fact that buffers are full.

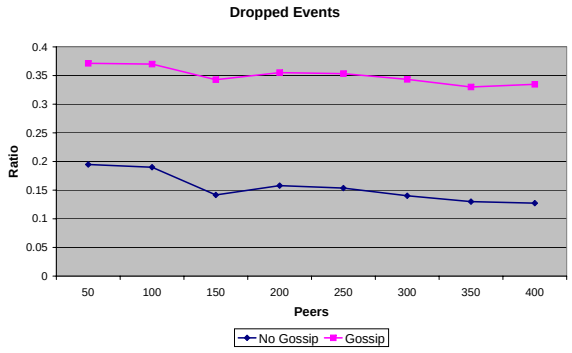

Figure 9: Percentage of Dropped Events

Figure 10 shows the average time needed to identify an event as obsolete, depending on the percentage of the packet loss. In the chart, times are considered with respect to the event reception times at the considered peer. Hence, positive values correspond to the fact that the event is identified as obsolete after it is received. Conversely, negative values indicate that, on average, events are identified as obsolete before they arrive at destination. As to the scheme that does not activate the gossip protocol, the higher the packet loss the more difficult to disseminate information on obsolescence, since events are marked as obsolete after higher delays. Conversely, when the gossiping protocol is active, higher packet loss probabilities correspond to a decrement, on average, of the difference between the times when an event is identified as obsolete and when it is received at a given node. This demonstrates that the gossip subprotocol is of great help to spread the additional information needed by our synchronization protocol.

The discussion above was mainly concerned to the delivery and gossip subprotocols. Then, delivered events are processed based on our game state update protocol. An interesting metric to measure here is the amount of rollbacks of the optimistic processing scheme, based on the use of the notions of correlation and obsolescence. To this aim, we show in Figure 11 the average rollback ratio measured in our simulations. In simple words, we measure the total number of rollbacks in the system over the total number of generated events. In this case, trying to show the benefits introduced by the checks for correlated and obsolete events, we varied the non-correlation probability. (We mentioned that higher non-correlation probabilities imply lower probabilities of rolling back some events and higher probabilities of having obsolete events.)

We contrasted our optimistic game state update protocol against a classic optimistic protocol that does not perform

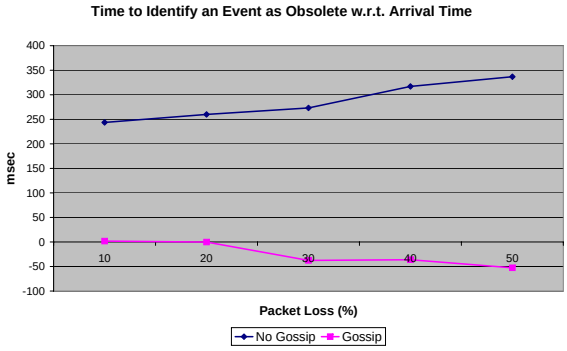

Figure 10: Time for Identifying Obsolete Events w.r.t. Arrival Time

any check for correlation and obsolescence (i.e., a pure Time Warp like synchronization scheme). It is worth noticing that the curve of the classic approach lies horizontally, as it does not gain benefits from obsolescence and correlation, while our protocol avoids to trigger the rollback procedure for obsolete or non-correlated events.

Having lower rollback ratios imply that first, lower computational overheads are locally experienced at different peers; this speeds up the event processing activity and the synchronization among peers. Second, having fewer rollbacks makes the the game more fluid. Indeed, processed events update the game state being perceived by players. These updates must be correctly and uniformly presented to players. As a matter of fact, when no notion of global virtual time is exploited, and computed updates are directly shown to the players, it is clear that an high number of rollbacks will result in a jerky game evolution [12, 15]. Conversely, in the case that a (global virtual time based) approach is exploited, where game state updates are shown only after having reached an agreement that rollbacks are no more possible for certain events, our synchronization protocol is anyhow faster in producing correct game state advancements, thanks to the combination of the delivery, gossip and game state update subprotocols.

These results confirm that higher responsiveness and playability degrees can be guarantee thanks to our synchronization protocol.

6. CONCLUSIONS

The rapid dissemination of events is a main issue in real-time distributed applications such as MOGs. Moreover, the need for scalable and fault-tolerant solutions suggest the use of peer-to-peer approaches. As the number of peer nodes grows, the communication topology should be carefully managed, by resorting to some kind of overlay network. According to this scenario, efficient event synchronization protocols are needed to guarantee a consistent and responsive evolution of the game state.

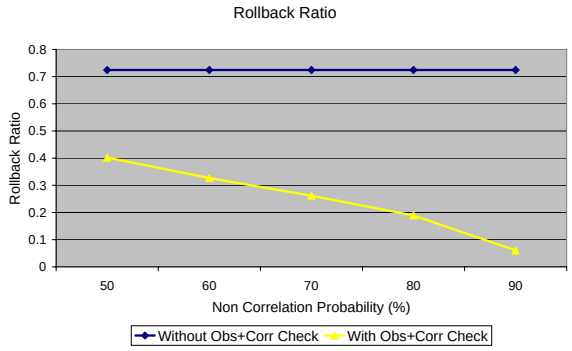

Figure 11: Game State Update Protocol: Rollback Ratio

In this context, the main outcomes of this work are that dropping strategies, which are devoted to discard superseded information, and techniques to relax event delivery orders, may be employed to hasten the synchronization among peers. Moreover, the use of a gossip protocol, running in background, thought to disseminate information about correlated/obsolete events, can improve the overall performances of the system.

7. ACKNOWLEDGMENTS

The author wishes to thank all the anonymous reviewers for their valuable comments and suggestions to improve the final version of this paper.

8. REFERENCES

[1] A. R. Bharambe, S. Rao, and S. Seshan. Mercury: a scalable publish-subscribe system for internet games. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 3–9, New York, NY, USA, 2002. ACM.

[2] J.-C. Bolot, S. Fosse-Parisis, and D. F. Towsley. Adaptive FEC-Based Error Control for Internet Telephony. In *Proceedings of IEEE INFOCOM'99*, pages 1453–1460. IEEE Computer Society, 1999.

[3] M. Borella. Source models for network game traffic. *Computer Communications*, 23(4):403–410, February 2000.

[4] M. Bui, F. Butelle, and C. Lavault. A distributed algorithm for constructing a minimum diameter spanning tree. *J. Parallel Distrib. Comput.*, 64(5):571–577, 2004.

[5] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 20(8):1489–1499, 2002.

[6] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[7] Y. H. Chu, S. G. Rao, and H. Zhang. A case for end system multicast (keynote address). In *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 1–12, New York, NY, USA, 2000. ACM.

[8] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):642–657, 1999.

[9] E. Cronin, B. Filstrup, S. Jamin, and A. Kurc. An efficient synchronization mechanism for mirrored game architectures (extented version). *Multimedia Tools and Applications*, 23(1):7–30, May 2004.

[10] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, 2003.

[11] J. Farber. Network game traffic modelling. In *Proceedings of the 1st workshop on Network and system support for games*, pages 53–57. ACM Press, 2002.

[12] S. Ferretti. Interactivity maintenance for event synchronization in massive multiplayer online games (ph.d. thesis). Technical Report UBLCS-2005-05, Department of Computer Science, University of Bologna, 2005.

[13] S. Ferretti. Cheating Detection Through Game Time Modeling: A Better Way to Avoid Time Cheats in P2P MOGs? *Multimedia Tools and Applications*, 37(3):339–363, March 2008.

[14] S. Ferretti, M. Roccetti, and C. E. Palazzi. An optimistic obsolescence-based approach to event synchronization for massive multiplayer online games. *International Journal of Computers and Applications*, 29(1):33–43, 2007.

[15] R. Fujimoto. *Parallel and Distribution Simulation Systems*. John Wiley & Sons, Inc., 1999.

[16] T. Hampel, T. Bopp, and R. Hinn. A peer-to-peer architecture for massive multiplayer online games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 48, New York, NY, USA, 2006. ACM.

[17] S. P. Hernandez, J. Fanchon, K. Drira, and M. Diaz. Causal broadcast protocol for very large group communication systems. In *Procedings of the 5th International Conference on Principles of Distributed Systems (OPODIS 2001)*, Studia Informatica Universalis, pages 175–188. Suger, Saint-Denis, rue Catulienne, France, December 2001.

[18] D. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[19] M. Jelasity, A. Montresor, and Ö. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, 2005.

[20] D. Kostić, A. C. Snoeren, A. Vahdat, R. Braud, C. Killian, J. W. Anderson, J. Albrecht, A. Rodriguez, and E. Vandekieft. High-bandwidth data dissemination for large-scale distributed systems. *ACM Trans. Comput. Syst.*, 26(1):1–61, 2008.

[21] A. Kshemkalyani and M. Singhal. An optimal algorithm for generalized causal message ordering. In *PODC '96: Proceedings of the fifteenth annual ACM*

symposium on Principles of distributed computing, page 87. ACM Press, 1996.

[22] X. Li, J. Misra, and C. G. Plaxton. Active and concurrent topology maintenance. In *DISC*, pages 320–334, 2004.

[23] L. Mathy, R. Canonico, and D. Hutchison. An overlay tree building control protocol. In *NGC '01: Proceedings of the Third International COST264 Workshop on Networked Group Communication*, pages 76–87, London, UK, 2001. Springer-Verlag.

[24] M. Mauve, S. Fischer, and J. Widmer. A generic proxy system for networked computer games. In *Proceedings of the 1st workshop on Network and system support for games*, pages 25–28. ACM Press, 2002.

[25] M. Oliveira and T. Henderson. What online gamers really think of the internet? In *NETGAMES*, pages 185–193, 2003.

[26] C. E. Palazzi, S. Ferretti, S. Cacciaguerra, and M. Roccetti. Interactivity-loss avoidance in event delivery synchronization for mirrored game architectures. *IEEE Transactions on Multimedia*, 8(4):874–879, 2006.

[27] J. Pang, F. Uyeda, and J. R. Lorch. Scaling peer-to-peer games in low-bandwidth environments. In *IPTPS '07: Proceedings of the 6th International Workshop on Peer-to-Peer Systems*, Feb. 2007.

[28] F. Pedone and A. Schiper. Generic broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 94–108, London, UK, 1999. Springer-Verlag.

[29] D. E. Pendarakis, S. Shi, D. C. Verma, and M. Waldvogel. Almi: An application level multicast infrastructure. In *USITS*, pages 49–60, 2001.

[30] J. O. Pereira, L. Rodrigues, and R. C. Oliveira. Semantically reliable multicast: Definition, implementation, and performance evaluation. *IEEE Trans. Computers*, 52(2):150–165, 2003.

[31] C. Perkins, O. Hodson, and V. Hardman. A survey of packet loss recovery techniques for streaming audio. In *Readings in multimedia computing and networking*, pages 607–615, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[32] S. T. Piergiovanni and R. Baldoni. Connectivity in eventually quiescent dynamic distributed systems. In *LADC*, pages 38–56, 2007.

[33] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[34] S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Networked Group Communication*, pages 14–29, 2001.

[35] J. Rosenberg, L. Qiu, and H. Schulzrinne. Integrating packet fec into adaptive voice playout buffer algorithms on the internet. In *INFOCOM*, pages 1705–1714, 2000.

[36] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[37] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[38] S. D. Webb, S. Soh, and W. Lau. Enhanced mirrored servers for network games. In *NetGames '07: Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 117–122, New York, NY, USA, 2007. ACM.

[39] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

[40] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *NOSSDAV '01: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 11–20, New York, NY, USA, 2001. ACM.