# Preventing Look-Ahead Cheating with Active Objects

**Conference Paper** *in* Lecture Notes in Computer Science · January 2006

DOI: 10.1007/11674399_21 · Source: OAI

Jouni Smed
University of Turku
**141** PUBLICATIONS **1,588** CITATIONS

# Preventing Look-Ahead Cheating with Active Objects

Jouni Smed and Harri Hakonen

Turku Centre for Computer Science (TUCS) and Department of Information Technology, University of Turku, Lemminkäisenkatu 14 A, FI-20520 Turku, Finland
jouni.smed@cs.utu.fi, harri.hakonen@cs.utu.fi

**Abstract.** In turn-based networked multiplayer computer games it is possible to cheat by delaying the announcement of one's action for a turn until one has received messages from all the other players. This look-ahead cheating can be prevented with the lockstep protocol, which requires the players first to announce a commitment to an action, which can later on be checked against the announced action, and then the action. However, because lockstep protocol requires separate transmissions for the commitment and the action, and a synchronization step before the actions can be announced, it slows down the turns of the game. In this paper, we propose that active objects can be used to prevent look-ahead cheating. Moreover, we can parameterize the probability of catching cheaters: the smaller this probability is, the less bandwidth and transmissions are required. In most cases, the mere threat of getting caught is enough to discourage cheating and, consequently, this probability could be quite small.

## 1 Introduction

Online security has recently become a major concern for the entertainment industry. The gaming sites periodically report on attacks and warn the users against misbehaviour and cheating. The cheaters attacking the games are mainly motivated by an appetite for vandalism or dominance. However, only a minority of the cheaters try to create open and immediate havoc, whereas most of them want to achieve a dominating, superhuman position and hold sway over the other players.

As the online gaming is becoming a more lucrative business, potential financial losses, caused directly or indirectly by cheaters, are now a major concern among the online gaming sites and the main motivation to implement countermeasures against cheating. In this respect, cheating prevention has three distinct goals [1], [2]:

- protect the sensitive information,
- provide a fair playing field, and
- uphold justice inside the game world.

Each of these goals can be viewed from a technical or social perspective: Sensitive information (e.g., players' accounts) can be gained, for instance, by cracking the passwords or by pretending to be an administrator and asking the players to give their passwords. A fair playing field can be compromised, for instance, by tampering with the network traffic or by colluding with other players. The sense of justice can be violated, for instance, by abusing inexperienced and ill-equipped players or by ganging up and controlling parts of the game world.

Although this paper concentrates on a specific problem concerning the fair playing field, namely preventing look-ahead cheating in turn-based games, we begin with a review of common cheating methods in Sect. 2. In Sect. 3 we examine the lockstep protocol and its variations, which aim at preventing look-ahead cheating. To improve the responsiveness, we introduce a scheme which uses active objects to prevent and detect look-ahead cheating in Sect. 4. The concluding remarks appear in Sect. 5.

## 2 Methods Used in Cheating

In a networked multiplayer game, a cheater can attack the clients, the servers, or the network connecting them. Figure 1 illustrates the typical attacks [3]: On the client side, the attacks focus on compromising the software or game data, and tampering with the network traffic. Game servers are vulnerable to network attacks as well as physical attacks such as theft or vandalism. Third party attacks on clients or servers include IP spoofing and denial-of-service attacks. In the following subsections we briefly review the common cheating methods.

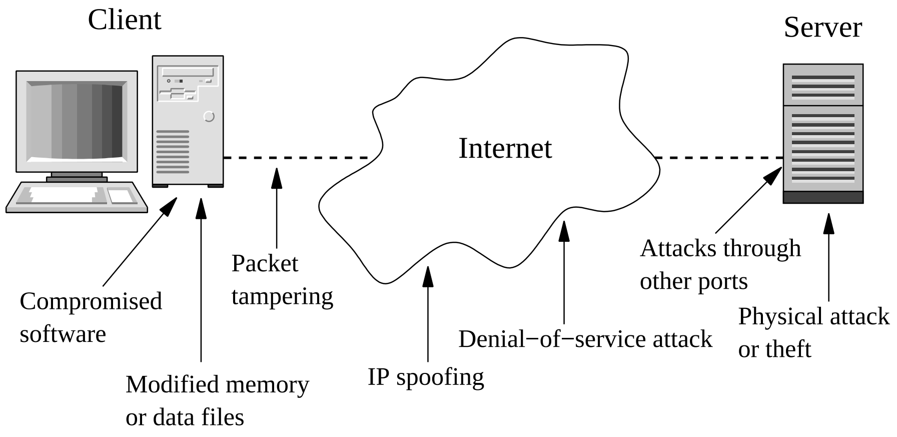

**Fig. 1.** Typical attacks in a networked multiplayer game.

## 2.1 Tampering with the Network Traffic

In first-person shooter games, a usual way to cheat is to enhance the player's reactions with *reflex augmentation* [4]. For example, an aiming proxy can monitor the network traffic and keep a record of the opponents' positions. When the cheater fires, the proxy uses this information and sends additional rotation and movement control packets before the fire command thus improving the aim. Reversely, in *packet interception* the proxy prevents certain packets from reaching the cheating player. For example, if the packets containing damage information are suppressed, the cheater becomes invulnerable. In *look-ahead cheating*, time-stamped packets are forged so that they seem to be issued before they actually were (see Fig. 2). In a *packet replay* attack, the same packet is sent repeatedly. For example, if a weapon can be fired only once in a second, the cheater can send the fire command packet hundred times a second to boost its firing rate.

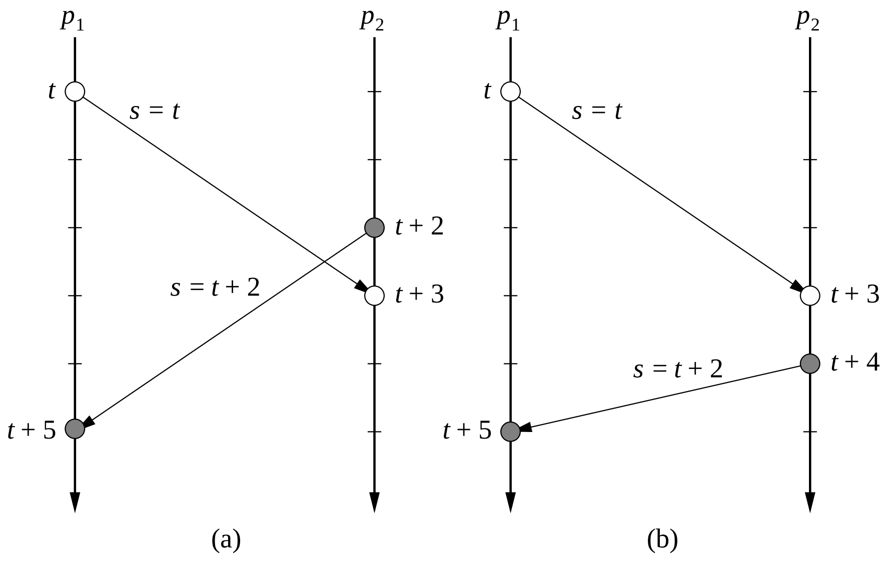

**Fig. 2.** Assume the players must time-stamp their outgoing messages, and the latency between the players is 3 time units. (a) If both players are fair, $p_1$ can be sure that the message from $p_2$, which has the time-stamp $t+2$, was sent before the message issued at $t$ has arrived. (b) If $p_2$ has a latency of 1 time unit but pretends that it is 3, look-ahead cheating using forged time-stamps allows $p_2$ to base decisions on information that it should not have.

A common method for breaking the control protocol is to change bytes in a packet and observe the effects. A straightforward way to prevent this is to use checksums. For this purpose, the MD5 algorithm [5] is widely used (e.g., in the PunkBuster system [6]), because it is well tested, publicly available and fast enough for real-time computer games. However, there are two weaknesses that cannot be prevented with checksums alone: The cheaters can reverse engineer the checksum algorithm or they can attack with packet replay.

By encrypting the command packets, the cheaters have a lesser chance to record and forge information. However, to prevent a packet replay attack requires that the packets carry some state information so that even the packets with a similar payload appear to be different. Instead of serial numbering, pseudo-random numbers provide a better alternative. Random numbers can also be used to modify the packets so that even identical packets do not appear the same. Dissimilarity can be further induced by adding a variable amount of junk data to the packets, which eliminates the possibility of analysing their contents by the size.

### 2.2 Illicit Information

A cracked client software may allow the cheater to gain access to the replicated, hidden game data (e.g., the status of other players) [7]. On the surface, this kind of passive cheating does not tamper with the network traffic, but the cheaters can base their decisions on more accurate knowledge than they are supposed to have. For example, typical exposed data in real-time strategy games are the variables controlling the visible area on the screen (i.e., the fog of war). This problem is common also in first-person shooters, where compromised graphics rendering drivers may allow the player to see through walls.

Strictly speaking, information exposure problems stem from the software and cannot be prevented with networking alone. Clearly, the sensitive data should be encoded and its location in the memory should be hard to detect. Nevertheless, it is always susceptible to ingenious hackers and, therefore, requires some additional counter-measures. In a centralized architecture, an obvious solution is to utilize the server, which can check whether a client issuing a command is actually aware of the object with which it is operating. For example, if a player has not seen the opponent's base, he cannot give an order to attack it—unless he is cheating. When the server detects cheating, it can drop out the cheating client. A democratized version of the same method can be applied in a replicated architecture: Every node checks the validity of each other's commands, and if some discrepancy is detected, the nodes vote whether its source should be debarred from participating in the game. Moreover, the game world can be partially replicated so that the server sends data to each participating client on a need-to-know basis, which prevents the clients from having—and hacking—complete information about the game world [8].

### 2.3 Exploiting Design Defects

Network traffic and software are not the only vulnerable places in a computer game, but design defects can create loop-holes, which the cheaters are apt to exploit. For example, if the clients are designed to trust each other, the game is unshielded from *client authority abuse*. In that case, a compromised client can exaggerate the damage caused by a cheater, and the rest accept this information as such. Although this problem can be tackled by using checksums to ensure that each client has the same binaries, it is more advisable to alter the design so that the clients can issue command requests, which the server puts into operation.

In addition to poor design, distribution—especially the heterogeneity of network environments—can be the source of unexpected behaviour. For instance, there may be features that become eminent only when the latency is extremely high or when the server is under a denial-of-service attack (i.e., an attacker sends it a large number of spurious requests).

## 2.4 Collusion

The basic assumption of *imperfect information games* is that each player has access only to a limited amount of information. A typical example of such a game is poker, where the judgements are based on the player's ability to infer information from the bets thus outwitting the opponents. A usual method of cheating in individual imperfect information games is *collusion*, where two or more players play together without informing the rest of the participants. Normally this would not pose a problem, since the players are physically present and can (at least in theory) detect any attempts of collusion (e.g., coughs, hand signals, or coded language). However, when the game is played online, the players cannot be sure whether there are colluding players present. This means a serious threat to the e-casinos and other online gaming sites, because they cannot guarantee a fair playing field [9].

Only the organizer of an online game, who has the full information on the game, can take counter-measures against collusion. These counter-measures fall into two categories: *tracking* (determining who the players are) and *styling* (analysing how the players play the game). Unfortunately, there are no preemptive nor real-time counter-measures against collusion. Although tracking can be done in real-time, it alone is not sufficient. Physical identity does not reflect who is actually playing the game, and a cheater can always avoid network location tracking with rerouting techniques. Styling allows to find out if there are players who participate often in the same games and, over a long period, profit more than they should. However, this analysis requires a sufficient amount of game data, and collusion can be detected only afterwards.

The situation becomes even worse, when we look at the types of collusion in which the cheating players can engage. In active collusion, cheating players play more aggressively than they normally would. In poker, for example, the colluding players can outbet the non-colluding ones. In passive collusion, cheating players play more cautiously that they normally would. In Poker, for example, the colluding players can let only the one with strongest hand to continue the play whilst the rest of them fold. Although styling can manage to detect active collusion, it is hard—if not impossible—to discern passive collusion from a normal play.

### 2.5 Offending Other Players

Although players may act in accordance with the rules of a game, they can cheat by acting against the spirit of the game. For example, in online role-playing games, killing and stealing from other players are surprisingly common problems [10]. The players committing these "crimes" are not necessarily cheating, because they can operate well within the rules of the game. For example, in the online version of *Terminus* (Vicarious Visions, 2001) different gangs have ended up owning different parts of the game world, where they assault all trespassers. Nonetheless, we may consider an ambush by a more experienced and better-equipped player on a beginner cheating, because it is not fair nor justified.

There are different approaches to handle this problem. *Ultima Online* (Origin, 1997) originally left the policing to the players, but eventually this led to gangs of player killers which terrorized the whole game. This was counteracted with a rating system, where everybody is initially innocent, but any misconduct against other players (including the synthetic ones) brands the player as a criminal. Each crime increases the bounty on their head ultimately preventing them from entering shops. The only way to clear one's name is not to commit crimes for a given time. *EverQuest* (Verant Interactive, 1999) uses a different approach, where the players can mark themselves able to attack and be attacked by other players, or completely unable to engage in such activities. However, killing and stealing are not the only ways to harm another player but there are other, non-violent ways to offend such as blocking exits, interfering with fights, and verbal abuse.

## 3 Lockstep Protocol

In multiplayer games based on a peer-to-peer architecture, all nodes uphold the game state, and the players' time-stamped actions must be conveyed to all nodes. This opens a possibility to use look-ahead cheating (see Fig. 2), where the cheater gains an unfair advantage by delaying his actions—as if he had a high latency—to see what the other players do before choosing his action.

The *lockstep protocol* tackles this problem by requiring that each player announces first a commitment to an action; when everyone has received the commitments, the players reveal their actions, which can be then checked against the original commitments [11]. The commitment must meet two requirements: It cannot be used to infer the action, but it should be easy to compare whether an action corresponds to a commitment. An obvious choice for constructing the commitments is to calculate a hash value of the action. Figure 3 describes an implementation for the lockstep protocol. Although the details of the auxiliary functions, including the function Hash($\cdot$), are omitted here, their intention should be apparent.

We can readily see that the game progresses in the terms of the slowest player because of the synchronization. Nobody announces their action, until everybody has received all commitments—otherwise, if a player announces his action too

```
LOCKSTEP($a_p$, $P$)
  in:    action $a_p$ of the local player $p$; set of remote players $P$
  out:   set of players' actions $A$
  local: commitment $c$; action $a$; set of commitments $C$
 1: $c \leftarrow$ HASH($a_p$)
 2: SEND-ALL($P$, $c$)                    ▷ Announce commitment to other players.
 3: $C \leftarrow \{c\}$
 4: $C \leftarrow C \cup$ RECEIVE-ALL($P$)        ▷ Get other players' commitments.
 5: SYNCHRONIZE($P$)                      ▷ Wait until everyone is ready.
 6: SEND-ALL($P$, $a_p$)                  ▷ Announce the action.
 7: $A \leftarrow \{a_p\}$
 8: $A \leftarrow A \cup$ RECEIVE-ALL($P$)        ▷ Get other players' actions.
 9: for all $a \in A$ do
10:     find $c \in C$ for which $sender(c) = sender(a)$
11:     if $c \neq$ HASH($a$) then          ▷ Are commitment and action different?
12:         error action $a$ does not comply with commitment $c$
13:     end if
14: end for
15: return $A$
```

**Fig. 3.** Pseudocode for the lockstep protocol.

early, those that have not yet announced their commitments could still change their mind. This may suit a turn-based game, which is not time critical, but if we want to use the lockstep protocol in a real-time game, the turns have to be short or there has to be a time limit inside which a player must announce the action or lose that turn altogether.

To overcome this drawback, we can use an *asynchronous lockstep protocol*, where each player advances in time asynchronously from the other players but enters into a lockstep mode whenever interaction is required. The mode is defined by a sphere of influence surrounding each player, which outlines the game world that can possibly be affected by a player in the next turn (or subsequent turns). If two players' spheres of influence do not intersect, they cannot affect each other in the next turn, and hence their decisions will not affect each other when the next the game state is computed and they can proceed asynchronously.

In the *pipelined lockstep protocol*, synchronization is loosened by having a pipe of size $\ell$ where the incoming commitments are stored (i.e., in basic lockstep $\ell = 1$) [12]. Instead of synchronizing at each turn, the players can send several commitments, which are pipelined, before the corresponding opponents' commitments are received. In other words, when player $p$ has received all other players $r$ commitments $c_r^t$ for the turn $t$, it announces its action $a_p^t$ (see Fig. 4). The pipeline may include commitments for the turns $(t+1), \ldots, (t+\ell)$, and player $p$ can announce commitments $c_p^{t+1}, \ldots, c_p^{t+\ell}$ before it has to announce action $a_p^t$. However, this opens a possibility to reintroduce look-ahead cheating: If a player announces its action earlier than required by the protocol, the other players can change both their commitments and actions based on that knowledge. This can be counteracted with an *adaptive pipeline protocol*, where the idea is to measure the actual latencies between the players and to grow or shrink the pipeline size accordingly [13].

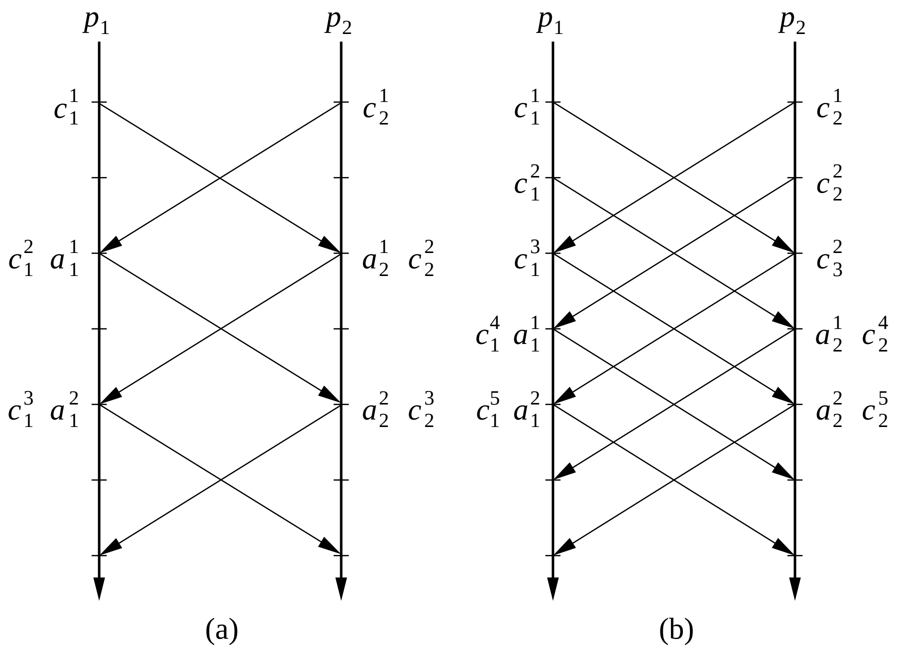

**Fig. 4.** (a) The lockstep protocol synchronizes after each turn and waits until everybody has received all commitments. Hence, player $p_1$ can announce only commitment $c_1^1$ for the turn 1 before it must announce action $a_1^1$, but of course that happens only after it has received commitment $c_2^1$ from player $p_2$. (b) The pipelined lockstep protocol has a fixed size pipe (here $\ell = 3$), which holds a sequence of commitments. Now, player $p_1$ can announce commitments $c_1^1$, $c_1^2$ and $c_1^3$ before it must announce action $a_1^1$.

## 4 Active Objects

The lockstep protocol requires the players to send two transmissions, commitment and action, in each turn. Let us now address the question, whether we can use only one transmission and still detect look-ahead cheating. Single transmission means that the action must be included in the outgoing message, and the receiver is allowed to view it only after it has replied with its own action. This would require that the communication system itself ensures the exchange of messages. But how can a player know that the exchange of messages in another player's computer has not been compromised? A cheater could intercept and alter the outgoing messages or simply hack the communication system itself.

To secure the exchange of messages, which happens in a possibly "hostile" territory, we can use active objects. A player provides an active object, a *delegate*, which includes a program code to be run in the other player's computer. The delegate acts then as a trusted party for the originating player by guaranteeing the message exchange.

Let us illustrate the idea with the following example using the game rock-paper-scissors. Player $p$ goes through the following stages:

1. Player $p$ decides the action "paper", puts this message inside a box and locks it. The key to the box can be generated by the delegate of player $p$, which has been sent beforehand to player $r$.
2. Player $p$ gives the box to the delegate of player $r$ which closes it inside another box before sending it to player $r$. Thus, when the message comes out from the delegate, player $p$ cannot tamper with its contents.
3. Once the double-boxed message is sent, the delegate of player $r$ generates a key and gives it to player $p$. This key will open the box enclosing the incoming message from player $r$.
4. When player $p$ receives a double-boxed message originating from player $r$, it can open the outer box, closed by its own delegate, and the inner box using the key it received from the delegate of player $r$.
5. Player $p$ can now view the action of player $r$.

At the same time, player $r$ goes through the following stages:

1. Player $r$ receives a box from player $p$. It can open the outer box, closed by its own delegate, but not the inner box.
2. To get the key to the inner box, player $r$ must inform its action to the delegate of player $p$. Player $r$ chooses "rock", puts it in a box and passes it to the delegate.
3. When the message is sent, player $r$ receives the key to the inner box from the delegate of player $p$.
4. Player $r$ can now view the action of player $p$.

Although we can trust, at least to some extent, to our delegates, there remains two problems that should be addressed. First, the delegate must ensure that it really has a connection to the originating player, which seems to incur extra talk-back communication. Second, although one-to-one exchange of messages is secure, there is no guarantee that the player does not alter its action when it sends a message to another player.

Let us tackle first the problem of ensuring the communication channel. Ideally, the delegate, once started, should contact the originating player and convey a unique identification of itself. This identification should be a combination of dynamic information (e.g., the memory address where the delegate is located or the system time when the delegate was created) and static information (e.g., built-in identification number or the Internet address of the computer where the delegate is run). Dynamic information is needed to prevent a cheater from creating a copy of the delegate and using that as surrogate to test out how it works.

Static information ensures that after the communication check, the delegate has not been moved to somewhere else or replaced.

If we could trust the run environment where the delegate resides, there would no need to do any check-ups at all. On the other hand, in a hostile environment we would have to ensure the communication channel every time, and there would be no improvement over the lockstep protocol. To reduce the number of check-up messages, the delegate can initiate them randomly with some parameterized probability. In practice, this probability can be relatively low—especially if the number of turns in the game is high. Rather than detecting the cheats this imposes a threat of being detected. Although a player could get away with a cheat, in the long run attempts to cheat are likely to be noticed. Moreover, as the number of participating players increases, it also increases the possibility of getting caught.

A similar approach helps us to solve the problem of preventing a player from sending differing actions to other players. Rather than detecting inconsistent action in the current turn, the players can "gossip" among themselves about the actions made in the previous turns. These gossips can then be checked against the recorded actions from the previous turns, and any discrepancy would indicate that somebody has cheated. Although the gossip could include all previous actions, it is quite sufficient to include only a small, randomly chosen subset of them—especially if the number of participants is high. This gossiping does not require extra communication because it can be piggybacked in the transmitted messages.

In the following subsections we give a more detailed description of the delegate and the communication turn.

## 4.1 Delegate

Pseudocodes for the routines in a delegate are described in Fig. 5. The details of the encoding and decoding routines Encode-Message$(\cdot)$, Encode-ID$(\cdot)$, and Decode-ID$(\cdot)$ are omitted. The delegate receives from the caller a sender object $\mathcal{S}$ and a receiver object $\mathcal{R}$, which provide communication routines for networking.

The message exchange may begin with a connection check, where the delegate encodes its identification and sends it to the originating player. If the decoded reply complies with the identification, the delegate presumes that connection is established and proceeds encoding and sending the message. After that, it constructs a decoder object, which will open the message from the originating player in the current turn.

It is worth noticing that the delegate poses a threat to the player who is running it. Therefore, the operations it is allowed to carry out should be limited to the bare necessities. For example, in the Java run environment this can be realized by defining a tight security policy on the delegate objects, which prevents them, for example, from accessing files or sending data over the network to a third party.

```
Exchange($e, \mathcal{S}, \mathcal{R}$)
  in:     encoded message $e$ to be sent; sender $\mathcal{S}$; receiver $\mathcal{R}$
  out:    originating player's decoder $\mathcal{C}_p^t$ for the current turn
  local:  encoded outgoing message $f$
  static: originating player $p$; unique and dynamically created identification $i$
  1: Check-Connection($\mathcal{S}, \mathcal{R}, p, i$)
  2: $f \leftarrow$ Encode-Message($e$)
  3: $\mathcal{S}$.Send($p, f$)
  4: construct $\mathcal{C}_p^t$
  5: return $\mathcal{C}_p^t$

Check-Connection($\mathcal{S}, \mathcal{R}, p, i$)
  in:     sender $\mathcal{S}$; receiver $\mathcal{R}$; player $p$; unique identification $i$
  local:  encoded response $c$ from the originating player
  static: probability $b$ ($0 \leq b \leq 1$) of checking the connection
  1: if Random() $< b$ then
  2:     $\mathcal{S}$.Send($p$, Encode-ID($i$))
  3:     $c \leftarrow \mathcal{R}$.Receive($p$)              ▷ Waits until a message arrives.
  4:     if $i \neq$ Decode-ID($c$) then
  5:         error connection to the player $p$ is uncertain
  6:     end if
  7: end if
```

**Fig. 5.** Pseudocodes for the routines in a delegate.

### 4.2 Communication Turn

Figure 6 describes the communication turn. The message to be encoded includes three pieces of data: the action to be carried out in the current turn, the delegate to be used in the next turn, and gossip about the actions made in the previous turn. The implementation of the routine Encode-Message($\cdot$) is omitted, but it should comply with the decoder that was sent along delegate in the previous turn. The encoded message is exchanged with all the delegates which were received in the previous turn. The resulting set of decoders is then used to decode the incoming messages. The actions, delegates for the next turn, and gossip from the previous turn are extracted from the decoded messages. Finally, the received gossip is checked against the actions from the previous turn.

Figure 7 illustrates the situation from a single player's point of view. The game process comprises the application and delegates from the other players. The application communicates with the current delegates $D^t$ by passing encoded messages $e$ and receiving decoder objects $\mathcal{C}$. The sender object $\mathcal{S}$ and receiver object $\mathcal{R}$ provide a communication layer to the network. The game process of each player is connected to network as illustrated in Fig. 8.

```
COMMUNICATE($M, \mathcal{S}, \mathcal{R}, P, D^t, A^{t-1}$)
  in:    message $M = \langle a_p^t, \mathcal{D}_p^{t+1}, G_p^{t-1} \rangle$ of player's $p$ action $a_p^t$ for the current turn,
         delegate $\mathcal{D}_p^{t+1}$ for the next turn, and gossip $G_p^{t-1}$ from the previous turn;
         sender $\mathcal{S}$; receiver $\mathcal{R}$; set of remote players $P$; set of delegates $D^t$ for the
         current turn; set of all actions $A^{t-1}$ from the previous turn
  out:   set of players' actions $A^t$ for the current turn; set of delegates $D^{t+1}$ for the
         next turn
  local: set of decoders $C$; encoded message $e$; delegate $\mathcal{D}$; decoder $\mathcal{C}$; set $G^{t-1}$ of
         gossiped actions of the previous turn; set of encoded messages $E$; encoded
         message $f$; decoded message $N$
 1: $C \leftarrow \emptyset$
 2: $e \leftarrow$ ENCODE-MESSAGE($M$)
 3: for all $\mathcal{D} \in D^t$ do                    ▷ Get decoders from all delegates.
 4:     $\mathcal{C} \leftarrow \mathcal{D}$.EXCHAGE($e, \mathcal{S}, \mathcal{R}$)
 5:     $C \leftarrow C \cup \mathcal{C}$
 6: end for
 7: $A^t \leftarrow \{action(M)\}$                    ▷ Store the player's own action.
 8: $D^{t+1} \leftarrow \emptyset$
 9: $G^{t-1} \leftarrow \emptyset$
10: $E \leftarrow \mathcal{R}$.RECEIVE($P, t$)          ▷ Get all messages for the current turn.
11: for all $e \in E$ do
12:     $f \leftarrow$ DECODE-MESSAGE($e$)
13:     find decoder $\mathcal{C} \in C$ for which $origin(\mathcal{C}) = sender(f)$
14:     $N \leftarrow \mathcal{C}$.DECODE($f$)
15:     $A^t \leftarrow A^t \cup \{action(N)\}$
16:     $D^{t+1} \leftarrow D^{t+1} \cup \{delegate(N)\}$
17:     $G^{t-1} \leftarrow G^{t-1} \cup gossip(N)$
18: end for
19: if $G^{t-1} \nsubseteq A^{t-1}$ then
20:     error actions in $(G^{t-1} \setminus A^{t-1})$ are conflicting
21: end if
22: return $A^t, D^{t+1}$
```

**Fig. 6.** Pseudocode for the communication turn.

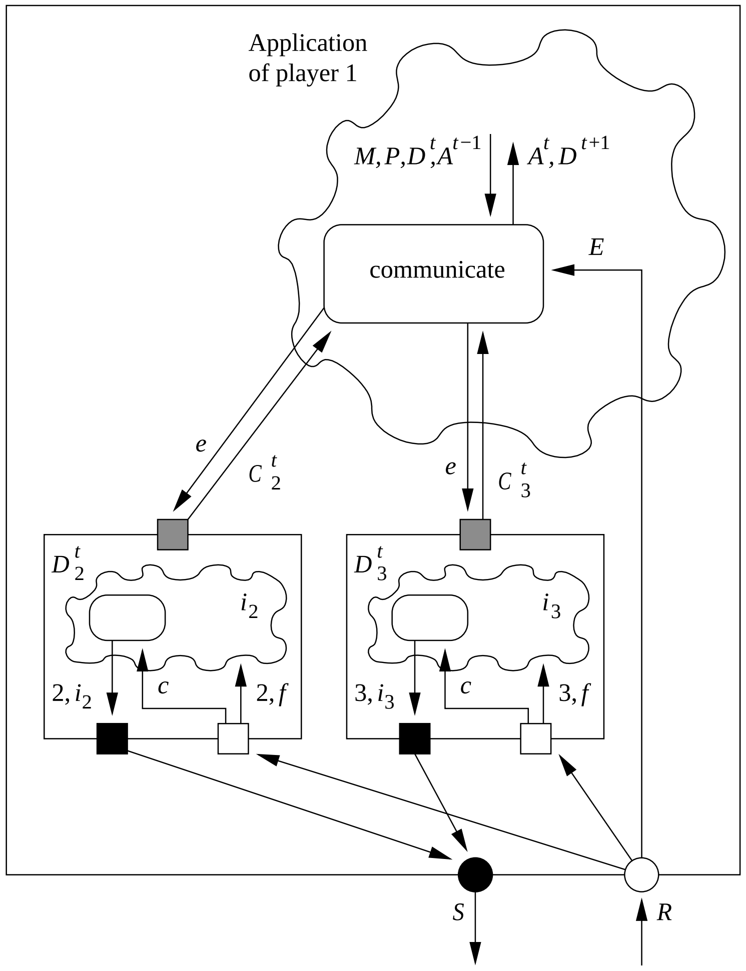

**Fig. 7.** The game process of player 1 comprises the application and the delegates of the other two players. Grey square represents message exchange, black square connection to the sender service (black circle), and white square connection to the receiver service (white circle). The oval inside the delegate cloud indicates the connection check.

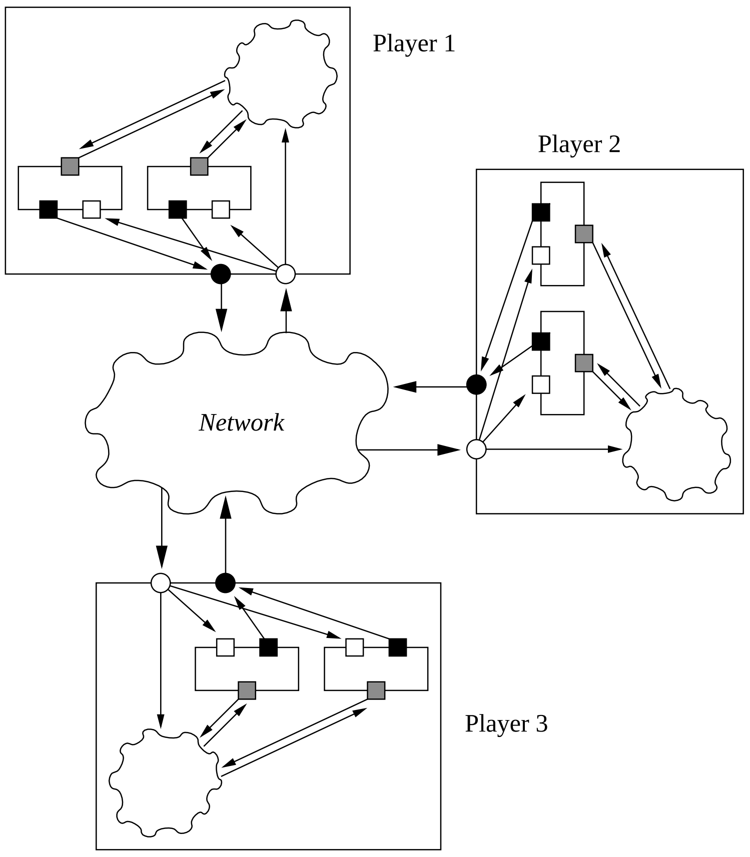

**Fig. 8.** Delegates in a three-player peer-to-peer network. Each player has received delegates to the other players' game processes.

## 5 Final Remarks

In this paper we described how to prevent look-ahead cheating by using active objects. Each player has a delegate on the other player's game process, which acts as a trusted party in the message exchange. The approach allows parameterization, because we can select how frequently the delegates check whether a connection to the originating player exists. Inconsistencies in the announced actions is controlled by including gossip about the previous actions, which can be also parameterized.

Further research on the effect of different parameter values is still required. Because we described in this paper the implementation of delegate and decoder objects in general terms, there is a need for a more rigorous technical study on how to implement them so that they are hard to reverse engineer and that they can endure also other kinds of attacks. The authors are currently testing presented approach on a Java run environment, since it provides a convenient way to implement delegates and decoders by using bytecodes and class loaders.

## References

1. Smed, J., Kaukoranta, T., Hakonen, H.: Aspects of networking in multiplayer computer games. The Electronic Library **20** (2002) 87–97
2. Yan, J.J., Choi, H.J.: Security issues in online games. The Electronic Library **20** (2002) 125–33
3. Kirmse, A., Kirmse, C.: Security in online games. Game Developer, July (1997) 20–8
4. Kirmse, A.: A network protocol for online games. In DeLoura, M., ed.: Game Programming Gems. Charles River Media (2000) 104–8
5. Rivest, R.: The MD5 message digest algorithm. Internet RFC 1321 (1992) Available at ⟨http://theory.lcs.mit.edu/∼rivest/Rivest-MD5.txt⟩.
6. Ray, T.: PunkBuster User Manual. (2001) Available at ⟨http://www.punkbuster.com/pbmanual/userman.htm⟩.
7. Pritchard, M.: How to hurt hackers: The scoop on Internet cheating and how you can combat it. Gamasutra, July 24 (2000) Available at ⟨http://www.gamasutra.com/features/20000724/pritchard_01.htm⟩.
8. Buro, M.: ORTS: A hack-free RTS game environment. In Schaeffer, J., Müller, M., Björnsson, Y., eds.: Proceedings of the Third International Conference on Computers and Games. Volume 2883 of Lecture Notes in Computer Science., Edmonton, Canada, Springer-Verlag (2002) 280–91
9. Johansson, U., Sönströd, C., König, R.: Cheating by sharing information—the doom of online poker? In Sing, L.W., Man, W.H., Wai, W., eds.: Proceedings of the 2nd International Conference on Application and Development of Computer Games, Hong Kong SAR, China (2003) 16–22
10. Sanderson, D.: Online justice systems. Game Developer, April (1999) 42–9
11. Baughman, N.E., Levine, B.N.: Cheat-proof playout for centralized and distributed online games. In: Proceedings of the Twentieth IEEE Computer and Communication Society INFOCOM Conference, Anchorage, AK (2001)

12. Lee, H., Kozlowski, E., Lenker, S., Jamin, S.: Multiplayer game cheating prevention with pipelined lockstep protocol. In Nakatsu, R., Hoshino, J., eds.: Entertainment Computing: Technologies and Applications, IFIP First International Workshop on Entertainment Computing, Makuhari, Japan (2002) 31–9
13. Cronin, E., Filstrup, B., Jamin, S.: Cheat-proofing dead reckoned multiplayer games. In Sing, L.W., Man, W.H., Wai, W., eds.: Proceedings of the 2nd International Conference on Application and Development of Computer Games, Hong Kong SAR, China (2003) 23–9