# Dealing with Heterogeneity for Mapping MMOFPS in Distributed Systems*

Ignasi Barri, Josep Rius, Concepció Roig, and Francesc Giné

Computer Science Department, University of Lleida, Spain
{ignasibarri,jrius,roig,sisco}@diei.udl.cat

**Abstract.** In this paper, we present a distributed heterogeneous system called *OnDeGaS* (On Demand Game Service), that fits the scalability and latency requirements of MMOFPS networked games. To exploit platform capabilities efficiently, the *OnDeGaS* system performs a mapping mechanism that assigns the game sessions of a MMOFPS, taking advantage of the specific available computational resources of individual nodes. We show through simulation that this mapping mechanism is able to deal with different heterogeneity conditions in the distributed area. It allows the system to grow at any moment according to the existing demand, while latency values are maintained under the acceptable threshold permitted in MMOFPS games.

## 1 Introduction

*Massively Multiplayer Online Games (MMOG)* are the most popular genre in the computer game world. They can be divided into three categories: *MMORPG* Role Games, *MMORTS* based on Real Time Strategy and *MMOFPS* known as First Person Shooter. The execution requirements vary with the way of playing in each of them [11]. On the one hand, MMORPG and MMORTS can have thousands of players in a single party, so bandwidth is an important feature for supporting them [5]. On the other hand, in MMOFPS, players are divided into many isolated game sessions, each with a handful of players, who are continuously interacting. Thus, response latency is the key factor in this case. In this paper, we focus on the optimization of MMOFPS games.

Traditionally, client-server systems have been the platforms to provide service to massively networked games. However, when the number of players increases this approach reaches its limits due to problems of scalability. The research community has proposed some alternatives to overcome client-server limits with decentralized structures where each machine contributes to, and benefits from, a large service oriented network. The way to distribute the entire game into the machines varies according to the category it belongs to. For MMORPG, some authors present solutions [6,7,8] where the exploitation of the distributed area is based on mapping the pieces of the splitted game world, or groups of players, into

* This work was supported by the MEyC-Spain under contract TIN2008-05913 and the CUR of DIUE of GENCAT and the European Social Fund.

M.R. Guarracino et al. (Eds.): Euro-Par 2010 Workshops, LNCS 6586, pp. 51–61, 2011.

the distributed nodes. Other proposals of game distribution are focused on the execution requirements of MMRPOG and MMORTS, such as solving cheating problems [8]. In the case of MMOFPS, cheating is not a key issue to face because it will potentially affect a single game service, having a negative impact for a small set of players, and a duration of the order of minutes. Then, for MMOFPS the research community focuses other challenges. Nharambe et al [3] proposes a solution to assign game sessions to a pure P2P system. The increase in latency time, inherent to this kind of architecture, is solved by proposing new rules in many features of current MMOFPS games, such as the size of the AOI (Area of Interest) of players in order to decrease the number of messages transferred among players. These improvements need to be included in the internal code of the game, which implies important implementation efforts.

In this paper, we propose a new system named *OnDeGaS* (On Demand Game Service), devoted to execute the game sessions of a MMOFPS without affecting the game's internal code. *OnDeGaS* is a hybrid system that combines the functionalities of a centralized server infrastructure with a distributed area composed of players' machines. A preliminary version of *OnDeGaS* was reported in [1]. This proposal was designed taking into account a study of the execution of MMOFPS's servers, where real traces of player's activity were monitored in order to tune and analyze the proper values for the configuration parameters, that determine the way to add and to remove game services in the distributed area. In the present paper, we propose the mapping mechanism for *OnDeGaS*, to assign game sessions to nodes, taking heterogeneity features into account such as latency and available cores. To the best of our knowledge, only the work presented by Iosup et al. [9], is also based in the dynamic assignment of resources in MMOFPS's. However they propose a prediction based mechanism that does not take into account the existing demand. The effectiveness of our approach, based on the resource assignment taking into account current demand, has been evaluated by means of simulation. Our results show that the *OnDeGaS* mapping mechanism is able to deal with different heterogeneity conditions, by properly exploiting the computational resources making up the distributed area of the system. We also show that latency values of the entire game are maintained under an acceptable threshold for MMOFPS in all cases.

The remainder of this paper is organized as follows. Section 2 describes the *OnDeGaS* system and the proposed mapping mechanism. Section 3 evaluates the *OnDeGaS* mapping performance taking system heterogeneity into account. Finally, Section 4 outlines the main conclusions and future work.

## 2 OnDeGaS System Description

In this section, the *OnDeGaS* system is described globally, discussing the components, their operation and implementation details.

### 2.1 System Model

Figure 1 shows the *OnDeGaS* system model that is made up of two main areas: one central area performing central services and a distributed area with several zones composed of a set of heterogeneous nodes.

The central area is devoted to performing the global control of the system and also to supplying players with services. Its components are the following:

- *Master Server (MS)* is the system's main server and acts as the bootstrap point.
- *Waiting Queue (WQ)* is a logical space in MS used to insert those players who cannot be served due to overload situations. It is a transitory state for players, who will be distributed in a short term.
- *Zones Queue (ZQ)* is a logical space in MS used to keep the information about the created zones updated. This information is used for distributing players to the already created zones.

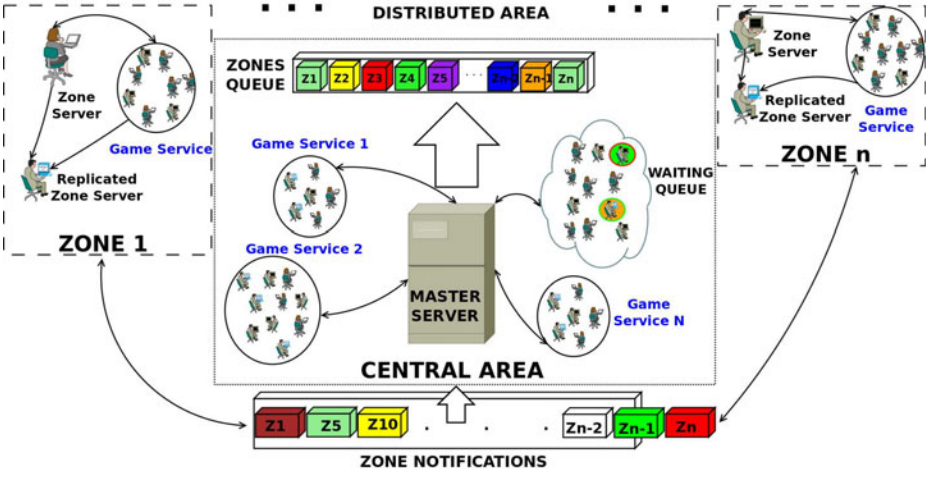



**Fig. 1.** OnDeGaS system model

The distributed area is composed of players' machines that are logically grouped in zones. A Zone number $i$, $Z_i$, has the following components:

- *Zone Server (ZS)* is the current server of the Zone.
- *Replicated Zone Server (RZS)* is the current replicated server of the Zone. It has the role of implementing fault tolerance policies. The role of RZS is to replace the ZS in case of failure. For this reason, players in the distributed area play against the ZS and its RZS, and the ZS sends the game state to both, players and the RZS. Players also update to RZS, to avoid losses of the state of the game, when the RZS replaces ZS.

Due to the inherent heterogeneity of nodes conforming the distributed area, each player in the system is characterized by two determinant computational attributes: latency in relation to the MS and the number of cores conforming its CPU. Both attributes are taken into account to find the best ZS and RZS for each Zone among the players waiting in the WQ. It is worth remarking that memory is not taking into account due to the low requirements of the MMOFPS games (order of tens MB per game service).

Regarding games, the following elements are distinguished:

- *Player ($P_i$)* is a client who connects to the system in order to play a MMOFPS.
- *Game Service (GS)* is an instance of a game, where a set of players is connected to play. Each GS will be hosted in the MS or in a single core of a ZS. At any moment, each GS can be in two different states: *active* when players are interacting in the GS, or *over*, when the GS has ended due to player disconnections or caused by the rules of the GSs. Normally, in MMOFPS, the *number of players* per GS is in the order of tens, while the *duration of the GS* is in the order of a few minutes.
- *Zones Notifications ($Z^N$)* are the set of N Zones that have sent a message to the MS to notify that their respective GSs are over. In this case, the MS will decide if the zone's players can be reaccepted.

### 2.2 System Operation

The operation of the *OnDeGaS* platform is a hybrid between the classical centralized client-server model, performed in the central area, and the distributed model, performed in Zones. The main idea of system operation consists of executing a set of GSs in the central area until it reaches the limit of its capabilities. When no more players can be accepted by the MS, new players are dynamically distributed to avoid large waiting times and to provide scalability to the system. Each Zone will execute $N$ GSs as maximum, $N$ being the number of cores of the chosen ZS.

The system operation is controlled by the continuous execution of Alg. 1, which has two input flows: new player connections ($P_i$) and zones ($Z_i$) that have ended their GS and want to enter the MS. At each iteration the MS checks its state (*MS.state*). According to this, the two following cases are considered:

1. **MS.State() == OVERLOAD**, If the MS is overloaded, each new player ($P_i$), will be added to the WQ queue. Next, the MS checks if the number of players in the WQ is greater than or equal to a predefined value $\alpha$, or whether the uptime of the WQ is greater than or equal to a predefined value $\beta$, too. If either of these two conditions is true, the algorithm tries to distribute all players located in the WQ, with function *MS.Distribute_Players* (see Alg. 2), to one already created zone in ZQ. If the previous function fails, then the MS will create a new Zone with function *MS.Create_Zone* (see Alg. 3).

```
Input: ∀P_i connecting to MS
Input: Z^N = {Z_i, Z_{i+1}, ..., Z_{n-i}, Z_n} notifying to MS
while True do
    switch MS.State() do
        case MS.State() == OVERLOAD
            if ∃ P_i then MS.Enqueue(P_i,WQ);
            if (WQ.size() ≥ α or WQ.uptime() ≥ β) then
                if (MS.Distribute_Players(WQ,ZQ) == FALSE) then
                    Z_i=MS.Create_Zone(WQ);
                    ZQ=ZQ+{Z_i};
        endsw
        case MS.State() == NOT OVERLOAD
            if WQ.uptime() ≥ β then
                forall the P_i in WQ do
                    MS.Accept(P_i);
                end
            if Z^N ≠ ∅ then MS.Reaccept(P_i);
            if ∃ P_i then MS.Accept(P_i);
        endsw
    endsw
end
```

**Algorithm 1.** OnDeGaS main Algorithm

The *MS.Distribute_Players* function (see Alg. 2) looks for those zones of ZQ whose ZS has at least one free core (*ZS.freeCores()*). If there are available zones, the function will select the one which has the lowest latency between the respective ZS and the MS (*MS.lowestLatency()*). Then the ZS and RZS of the selected Zone will accept all players located in the WQ and finally, a boolean is returned.

The *Create_Zone* functionality (see Alg. 3) executes the *lowestLatency* function to find the best ZS and RZS, in latency and computational resource terms. Moreover, the function ensures that RZS is able to serve at least the same number of GSs as the ZS to avoid problems when the fault tolerance mechanisms acts (*if* statement with function *swap*). Then, all players in the WQ are linked to the new ZS/RZS (*ZS/RZS.accept()*) with the aim of RZS keeping the same information as the updated ZS. Thus, a fault tolerance mechanism is maintained by the system. Then, a Zone $Z_i$ comprises the ZS, the RZS and the set of players previously located in the WQ.

2. **MS.state() == NOT OVERLOAD**. When the MS is not overloaded, Alg. 1 evaluates the three following conditional statements:
   - The first condition evaluates if the uptime of the WQ is greater than or equal to $\beta$; if it is, players located in the WQ will be accepted to play in the MS (*MS.accept()*). This acceptance flow acts like a *FIFO*, the first

```
Input: ZQ,WQ
Output: Boolean
MS.Distribute_Players(ZQ,WQ):
begin
    Available Zones = AZ = ∅;
    forall the ((ZS and RZS) ∈ Z_i) ∈ ZQ do
        if (ZS.freeCores() and RZS.freeCores()) then AZ =AZ +{Z_i};
    end
    if (AZ!= ∅) then
        Z_i=MS.lowestLatency(AZ);
        Z_i.Accept(WQ);
        return TRUE;
    else
        return FALSE;
    end
end
```

**Algorithm 2.** OnDeGaS *Distribute_Players* function

```
Input: WQ
Output: Z_i
MS.Create_Zone(WQ):
begin
    ZS=MS.lowestLatency(WQ);
    RZS=MS.lowestLatency(WQ-{ZS});
    if (ZS.freeCores() > RZS.freeCores()) then swap(ZS,RZS);
    ZS/RZS.accept(WQ);
    Z_i = {ZS ∪ RZS};
    return Z_i;
end
```

**Algorithm 3.** OnDeGaS Create_Zone function

player in the WQ queue is the first to be connected to the MS if it has enough space.

- The second conditional statement gives priority of entry into the MS to players of those zones that sent an *over* message to notify that they had finished the GS, and wanted to start another GS in the MS. This happens whenever a round of the game has finished and players are waiting for the next round. In this case, if the set of notifying zones, $Z^N$, is not empty, the MS executes the *Reaccept* function. Note that distributed players are playing continuously in the MS or the zones, and the time transitions from WQ to the zones, and from the zones to the MS are of the order of seconds, which is an acceptable delay for the players.
- The last conditional statement allows to connect new players to the MS.

### 2.3 Implementation Issues

The following needs to be considered for the proper performance of the system:

**Lowest Latency Functionality.** *lowestLatency* function is based on a loop that checks the latency of all players located in the WQ (case Alg. 3) or all the ZS located in the ZQ with respect to the MS (case Alg. 2). Then, it selects the closest ZS or Zone to the MS to assign the players located in the WQ respectively.

**System Overload State.** To determine the system overload state, real traces of player's activity of a MMOFPS have been monitored and analyzed during two months, which allowed us to verify that the state of system overload is determined by the number of concurrent players playing in the MS. Many authors also corroborate this [2,5,11], as it has been proved experimentally that the number of concurrent players is directly related to the CPU and network usage.

**Free Cores Functionality.** In Alg. 2 and 3, the *freeCores* function is used. This function returns the number of free cores of the ZS or RZS (depending on which node executes the function). The studies carried out by Ye and Cheng in [11] show that with an idle processor, is possible to provide a MMOFPS with QoS easily. Thus, if the number of GSs per core was increased the QoS would decrease. Likewise, our system assumes that the player's computational resources are totally dedicated to the MMOFPS and therefore, it is feasible to take advantage of all these computational resources of a player. Thus, the maximum number of GSs that a Zone is able to execute is equal to the number of cores of the ZS, given that, a ZS reserves a core to run its own GS when the ZS is involved in a player role, apart from the ZS role.

## 3 Experimental Results

In this section, experimentation is conducted to demonstrate the feasibility and good performance of the proposed mapping mechanism for *OnDeGaS* system. The experimentation was performed through simulation using SimPy [10]. SimPy is a discrete-event simulation language based on standard Python. SimPy tools have been used to implement nodes of the platform, which can fulfill four distinct roles: player, ZS, RZS and MS. The SimPy procedures allow random behavior of the simulation to be created to represent the real behavior of a player.

Each simulation consists of $100,000$ player connections to the MS. The connections are sequential with constant inter-arrival time ($\approx 1$ second) to submit the MS to a constant stress situation or constant peak load, in order to verify that the distributed area is dynamically adapted to the on-demand queries of players. When the MS reaches its limit, $2,000$ concurrent players, (since the computational resources of a typical single machine server can support $2,000$ to $6,000$ concurrent clients [7]), no more players will be accepted, and new ones will be distributed to zones. Another important issue is the calculus of the players' latency against MS. This is determined by a triangulated heuristic, delimiting the 2-Dimensional Euclidean Space to ($x = [-110, +110], y = [-110, 110]$). This

methodology is based on the relative coordinates explained in [4]. Furthermore, each player has a lifetime determined by a Weibull distribution scaled from 0 seconds up to 24 hours. For the parameters $\alpha$ and $\beta$ used in Alg. 1, we considered the values of 32 players and 120 seconds respectively, it having been demonstrated in [1], that they are appropriate values to ensure a good performance of the whole system taking the characteristics of real MMOFPS into account. The length of a GS is 900 seconds [3] on average, following an exponential distribution.

To configure a heterogeneous system, we considered that a player can have 2, 4 or 8 cores, where $\omega_2$, $\omega_4$ and $\omega_8$ are the percentages of players with this number of cores in the system. Let $\omega_{max} = max(\omega_2, \omega_4, \omega_8)$ and $\omega_i$ and $\omega_j$ the remaining two percentages excluding this $\omega_{max}$. We define the heterogeneity degree of the system (*het_degree*) with equation (1).

$$het\_degree = 1 - \frac{\frac{(\omega_{max} - \omega i)}{\omega_{max}} + \frac{(\omega_{max} - \omega j)}{\omega_{max}}}{2} \tag{1}$$

The values of *het_degree* range from 0 to 1, where 0 means that is a homogeneous system, while 1 corresponds to a system with the same percentages of each type of players ($\omega_2 = \omega_4 = \omega_8$), it means totally heterogeneous.

According to the previous assumptions and functionalities, in the next Subsection, the performance provided by the mapping mechanism of *OnDeGaS* according to the *het_degree* is shown. The cases of the study are: ability to scale the distributed area and the QoS of the system, measured by the zone's average latency and the waiting time for players located in the WQ.

### 3.1 Performance Evaluation

The scalability of the *OnDeGaS* system indicates its ability to manage more zones on demand, while the QoS of the whole system is maintained.

Table 1 shows the average (AVG) and standard deviation (SD) for the number of created zones and QoS parameters under two conditions of heterogeneity: (a) *h_d=0* (*het_degree=0*), where all players have two cores ($\omega_2 = 100\%, \omega_4 = \omega_8 = 0\%$) and, (b) *h_d=1* (*het_degree=1*) with $\omega_2 = \omega_4 = \omega_8$. In each case we evaluated 100 different simulations. As can be observed in Zones column of Table 1, the distribution performed by *OnDeGaS* is able to exploit the additional cores of the heterogeneous system, as it creates a lower number of zones with more GSs. In practice, this will suppose a significant decrease in the overhead for the management of the set of ZS and RZS in the distributed area.

Regarding QoS, Table 1 shows similar average in the latency values in both cases, but the standard deviation vary between the homogeneous and heterogeneous case. This is due to the fact that in the heterogeneous, fewer zones are created and this behavior means that the set of potential ZS to distribute players located in the WQ was smaller than the homogeneous case. However, it is worth remarking that in all the cases, both systems, latency values are below the maximum acceptable threshold for MMOFPS (180 ms).

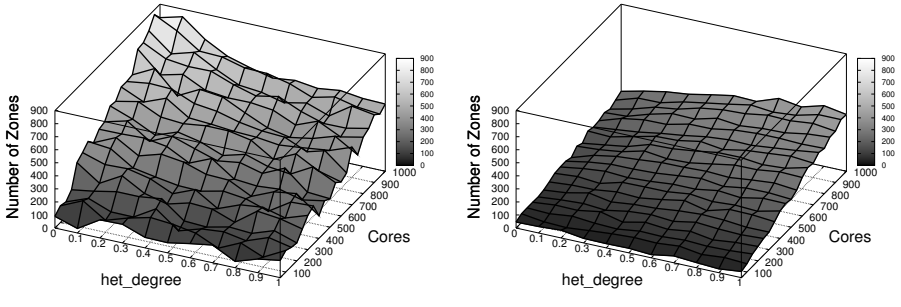
**Table 1.** System performance analysis

| | **Zones (No.)** | | **Latency (ms)** | | **WQ Time (sec.)** | | **FT Gain (sec.)** | |
|---|---|---|---|---|---|---|---|---|
| | $h\_d = 0$ | $h\_d = 1$ | $h\_d = 0$ | $h\_d = 1$ | $h\_d = 0$ | $h\_d = 1$ | $h\_d = 0$ | $h\_d = 1$ |
| **AVG** | 888.38 | 395.84 | 87.43 | 87.79 | 30.16 | 24.40 | 1007.865 | 1007.865 |
| **SD** | 193.37 | 76.92 | 0.293 | 0.955 | 4.70 | 6.30 | 166.65 | 166.65 |

For the waiting time of players located in the WQ, the experiment reveals a significant impact on the average, depending on the number of cores. Whenever a new Zone is created, a new ZS and RZS must be searched for. This process takes 30 seconds on average. This situation happens frequently in the homogeneous case (considering 2 cores) as indicated by the average of 30.16 sec. Nevertheless, this happens less often in the heterogeneous case as more players are mapped to zones already created, 24.40 sec. in this case.

It is also shown in Table 1 the benefits of the fault tolerance policy (FT Gain) that is implemented by the use of the RZS. On average, implementing zones with a single RZS represents an extra lifetime of $\approx 16$ minutes for zones, which means an average increase of 22% of the zones' lifetime. The standard deviation points out a small deviation of $\approx 3$ minutes, caused by the player's lifetime determined by a Weibull distribution. Thus, the heterogeneity of the system has not any influence in the effectiveness of the fault tolerance mechanisms.

To expand the study of the influence carried out by heterogeneity of the distributed area, we evaluated its effects in the number of created zones, as a representative case. Figure 2 points out its trend according to different values of *het_degree*, where the total number of cores in the whole system and the percentage values for $\omega_2$, $\omega_4$ and $\omega_8$ vary. Figure 2a corresponds to a system where the majority of the players have 2 cores ($\omega_{max} = \omega_2$) and the weights have the relation: $\omega_2 \geq \omega_4 \geq \omega_8$. In the same way, Figure 2b, shows the results for a system with a relation of $\omega_2 \leq \omega_4 \leq \omega_8$, where $\omega_{max} = \omega_8$. As can be




(a) $(\omega_{max} = \omega_2) \geq \omega_4 \geq \omega_8$ (b) $\omega_2 \leq \omega_4 \leq (\omega_{max} = \omega_8)$

**Fig. 2.** Number of zones created in function of *het_degree*

observed, the number of cores and their distribution among players drastically determines the number of zones created in both plots. When $\omega_{max} = \omega_2$, the number of created zones is, in most of the series, larger than when $\omega_{max} = \omega_8$. Thus, the system creates new zones only when the cores of the existing zones are busy. It can also be seen that the number of created Zones increases in all cases when the total number of cores is also increasing. Regarding the *het_degree*, there is a different trend in the plots of the figure. When $\omega_{max} = \omega_2$, the number of zones increases when het_degree decreases as most players have only 2 cores. However, the trend is the opposite when $\omega_{max} = \omega_8$, as most players have 8 cores. Thus, we can conclude that the *OnDeGaS* mapping system is able to deal with different heterogeneity conditions by properly exploiting the system's computational resources.

## 4 Conclusion and Future Work

In this paper, we presented a distributed heterogeneous system called *OnDeGaS* (On Demand Game Service), that fits the scalability and latency requirements of MMOFPS networked games. The proposed new system is made up of a Master Server (MS) carrying out centralized functionalities, and several zones that make up a distributed area. Whenever the MS is overload, the system scales by creating zones to execute game sessions. Zones are created taking latency with respect to the MS and the available number of cores composing their CPUs into account.

By means of simulation, experimental results show that the system is able to scale according to the demand. Moreover, it has been demonstrated that the number of created zones depends directly on the heterogeneity degree (*het_degree*) value. For higher *het_degree* values, fewer zones are created, thus avoiding excessive fragmentation of the system. Likewise, it has also been shown that this scalability does not damage the average latency, which is always below the maximum threshold allowed in MMOFPSs. Furthermore, the waiting time for players located in the WQ is reduced as the *het_degree* of the system is increased.

In future work, we are interested in improving the fault tolerance taking the *het_degree* characteristics of the system into account and also implementing market policies to reward the ZS and RZS. Another important improvement would be to merge the current simulator with a network simulator, to make a deeper study of the network problems derived from MMOFPS gaming.

## References

1. Barri, I., Giné, F., Roig, C.: A Scalable Hybrid P2P System for MMOFPS. Parallel, Distributed, and Network-Based Processing. In: Euromicro Conference (2010)
2. Bauer, D., Rooney, S., Scotton, P.: Network Infrastructure for Massively Distributed Games. In: NetGames (2002)
3. Bharambe, A., Douceur, J., Lorch, J.R., Moscibroda, T., Pang, J., Seshan, S., Zhuang, X.: Donnybrook: Enabling Large-Scale, High-Speed, Peer-to-Peer Games. In: SIGCOMM (2008)

4. Eugene, T.S., Zhang, H.: Predicting Internet Network Distance with Coordinates-Based Approaches. In: INFOCOM (2001)
5. Huang, G., Ye, M., Cheng, L.: Modeling System Performance in MMORPG. In: Global Telecommunications Conference Workshops (2004)
6. Keller, J., Simon, G.: Solipsis: A Massively Multi-Participant Virtual World. In: PDPTA (2003)
7. Knutsson, B., Lu, H., Xu, W., Hopkins, B.: Peer-to-Peer Support for Massively Multiplayer Games. In: INFOCOM (2004)
8. Liu, H.I., Lo, Y.T.: Dacap-A Distributed Anti-Cheating P2P Architecture for Massive Multiplayer On-line Role Playing Game. In: CCGRID (2008)
9. Nae, V., Iosup, A., Prodan, R.: Dynamic Resource Provisioning in Massively Multiplayer Online Games. IEEE Transactions on Parallel and Distributed Systems (2010)
10. IBM Developers Works: Charming Python: SimPy Simplifies Complex Models (Simulate Discrete Simultaneous Events for Fun and Profit) (2002)
11. Ye, M., Cheng, L.: System-Performance Modeling for Massively Multiplayer Online Role-Playing Games. IBM Syst. J. (2006)