A Scalability Model for Distributed Resource Management in Real-Time Online Applications

Dominik Meiländer[1], Sebastian Köttinger, and Sergei Gorlatch
University of Muenster
Muenster, Germany
[1]d.meil@uni-muenster.de

***Abstract*—We consider a challenging class of highly interactive virtual environments, also known as Real-Time Online Interactive Applications (ROIA). Popular examples of ROIA include multi-player online computer games, e-learning and training based on real-time simulations, and other challenging applications. ROIA combine high demands on scalability and real-time user interactivity with the problem of an efficient and economic utilization of resources, which is difficult to achieve due to the changing number of users. This paper proposes a generic scalability model for ROIA that analyzes the application performance during runtime and predicts the demand for load balancing, i.e., when to add/remove resources or redistribute workload. We prove the practical relevance of the model by incorporating it into our RTF-RMS resource management system where it is used to predict the influence of different load-balancing actions on the application scalability. Our model is utilized by RTF-RMS for finding efficient load-balancing actions and thresholds for how often these actions should be applied. We report experimental results on the load balancing of a multi-player online game using predictions from the scalability model.**

I. Introduction

This paper is motivated by the challenges of the emerging class of *Real-Time Online Interactive Applications (ROIA)*. Popular and market-relevant representatives of this application class are massively multi-player online games, as well as real-time training and e-learning based on high-performance simulation. ROIA are characterized by high performance requirements, such as: short response times to user inputs (about 0.1-1.5 s); frequent state updates (up to 50 Hz); large and frequently changing number of users in a single application instance (up to 10^4 simultaneously).

Since the high demands on ROIA performance usually cannot be satisfied by a single server, *scalability* (i.e., accommodating an increasing number of users) is achieved by employing distributed, multi-server application processing. A major problem in this context is the efficient utilization of multiple server resources, which is a non-trivial task due to a variable number of users who are continuously connecting to the application and disconnecting from it.

Since Cloud Computing offers cost-efficient leasing resources on demand, ROIA providers start to use Cloud resources for ROIA, e.g., via Amazon EC2 [1]. The availability of potentially unlimited Cloud resources leads to large and complex distributed systems and, therefore, increases the demand for predicting application scalability and estimating the effect of different load-balancing actions, i.e., adding/removing resources or redistributing workload on multiple servers.

In order to choose among multiple possible load-balancing actions, there are several challenges to be met: analyzing the effect of load-balancing actions on ROIA scalability with respect to different application workloads; choosing between redistributing load on existing resources vs. adding new resources; analyzing the computational overhead caused by each load-balancing action in order to find a threshold for how often each load-balancing action should be applied.

In this paper, we target these challenges that arise when choosing among multiple load-balancing actions by enhancing our previously developed scalability model for distributed ROIA processing [15]. Our contribution is a novel scalability model that analyzes the performance of a ROIA application during runtime, predicts the effect of different load-balancing actions, and determines a threshold for the maximum number of resources that can be used efficiently depending on the particular ROIA.

In order to prove the practical relevance of our approach, we incorporate the scalability model into our resource management system *RTF-RMS* [14]. RTF-RMS is implemented on top of the *Real-Time Framework (RTF)* [2] which is a middleware platform for high-level development and efficient execution of scalable ROIA that provides mechanisms for distribution and monitoring of the application state. We use RTF to provide a case study application that illustrates how our scalability model can be utilized by RTF-RMS for finding efficient load-balancing actions and thresholds for how often these actions should be applied.

The paper is organized as follows. Section II describes our target class of Real-Time Online Interactive Applications (ROIA) and the Real-Time Framework (RTF) used for their development and execution. Section III presents our new model for predicting the effect of two particular load-balancing actions on application scalability. Section IV presents our dynamic resource management system RTF-RMS and illustrates how the proposed scalability model improves its load-balancing strategy. Section V reports our experimental results on the load balancing of a multi-player online game using predictions of the scalability model in RTF-RMS. Section VI compares our approach to related work and concludes the paper.

0190-3918/13 $26.00 © 2013 IEEE
DOI 10.1109/ICPP.2013.90

II. Scalable ROIA Development with RTF

Typically, there are three different actors involved in the development and execution of ROIA: (i) *Application developers* implement the ROIA application, i.e., server and client programs realizing the application logic, and suitable mechanisms for application state distribution and monitoring, (ii) *Application providers* make ROIA accessible to users by executing application servers on hardware resources and implement dynamic load balancing for ROIA sessions according to the current user workload, and (iii) *Users* connect their personal computers (*clients*) to application servers using the application client and control their avatars that interact with application *entities*, i.e., other users' avatars or computer-controlled characters, in the virtual environment. All users access a common application state and interact with other users concurrently within one virtual environment.

We use the *real-time loop* model [21] for describing ROIA execution on hardware resources. Each user is connected to one application server that processes users' inputs (e.g., commands and interactions with other users) and sends application state updates to its users (left-hand side of Fig. 1). One iteration of the real-time loop is called a *tick* and consists of three steps:

1) Each server receives inputs from its connected users.
2) Each server computes a new application state according to the application logic.
3) Each server sends the newly computed state to its connected users and to other servers.

Steps 1 and 3 involve communication to transmit the users' inputs and state updates between multiple application servers. The computation of a new application state (step 2) involves quite compute-intensive calculations which apply the application logic to the current state, taking into account the newly received users' inputs. Typically, application servers filter state update information by calculating an *area of interest* for each user, i.e., only changes that are visible for the user are sent to him, in order to reduce the used network bandwidth. The time required for one iteration of the real-time loop (*tick duration*) is directly related to the application's response time, and, hence, is used as a quality-of-experience criterion for ROIA.

The *Real-Time Framework (RTF)* [8] is our high-level development platform for ROIA which supports the application developer in three essential tasks as explained in the following: (i) application state distribution, (ii) communication handling, and (iii) monitoring and distribution handling.

RTF supports three common methods of *application state distribution* among servers (on the right-hand side of Fig. 1): zoning, instancing and replication, and combinations of them [8]. Zoning assigns the processing of the entities in disjoint areas (*zones*) to distinct servers. Instancing creates separate independent copies of a particular zone; each copy is processed by a different server. In the replication approach, each server keeps a complete copy of the application state, but each server is responsible for a disjoint subset of entities (*active entities*, black in Fig. 1) and receives updates for so-called *shadow entities* (grey in Fig. 1) from other servers.

RTF provides high-level mechanisms for *communication handling*: automatic (de-)serialization for objects to be transferred over network (user inputs, application state updates, etc.), (un-)marshalling of data types, and optimization of the bandwidth usage. At runtime, (de-)serialization and transmission of objects are performed implicitly and asynchronously. RTF's *monitoring and distribution handling* allows the provider to change the distribution of the application state during runtime, as well as to receive monitoring data from RTF inside an application server. The distribution of the application state can be changed by load-balancing actions according to the following approaches: (i) adding/removing resources to/from the application processing using zoning, instancing and replication, and (ii) migrating users between application servers, i.e., transferring the responsibility for user input processing and state update computation from one server to another.

In the next section, we describe how our scalability model analyzes the performance of ROIA during runtime and predicts the effect of different load-balancing actions following the approaches (i) and (ii).

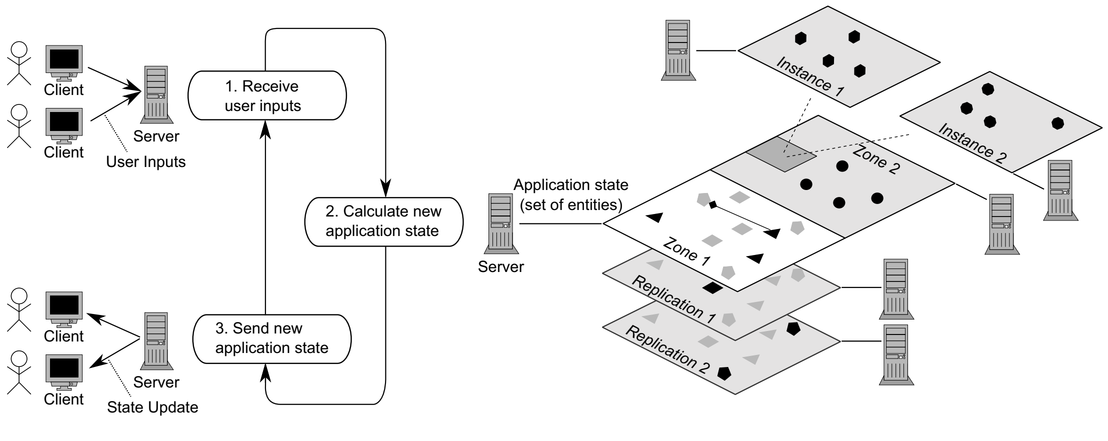

Fig. 1. One iteration of the real-time loop (left); RTF methods for application state distribution (right).

III. The Scalability Model

Our scalability model aims at executing load-balancing actions whenever the tick duration monitored at runtime indicates a poor quality of the ROIA. In order to analyze the distribution of the application state using the approaches (i) and (ii) mentioned at the end of Section II, we investigate the impact of two load-balancing actions on the scalability of ROIA: (a) *replication enactment* adds a new application server for replicating a highly frequented zone, and (b) *user migration* transfers the responsibility for input processing and state update computation between application servers that are replicating the same zone, in order to migrate users from overloaded application servers to underutilized servers.

The right-hand side of Fig. 1 illustrates how three application servers are processing Zone 1 cooperatively. Each user is connected to one application server to which he sends inputs and from which he receives application state updates. Application servers communicate with each other by (1) sending updates of their own users to other servers that are replicating the same zone, and (2) forwarding the interactions between users that are connected to different servers to the responsible server (in the figure, users connected to the same server have the same geometric shape). Hence, replication enactment involves additional overhead for inter-server communication and computing a consistent application state. User migration involves additional overhead too, namely for switching user connections between servers.

A. Achieving Scalability using Replication Enactment

Using replication for ROIA distribution is a promising approach since it allows to use multiple servers for the same zone, if that zone becomes overcrowded at runtime. However, assessing the scalability of replication is very challenging, because many aspects must be taken into account, including the replication overhead caused by the processing of shadow entities.

In the following, we describe four major computational tasks for each application server during the execution of a single real-time loop iteration when using replication. For each task, we enhance the generic description from [15] by elaborating on how the computational time can be measured in a ROIA, given n users and m computer-controlled non-player characters (NPCs) distributed equally on l replicas:

1) *Processing inputs* from $\frac{n}{l}$ connected users, which requires: (i) time $t_{\text{ua\_dser}}(n,m)$ for asynchronous reception and deserialization of user inputs, and (ii) time $t_{\text{ua}}(n,m)$ for validating and applying user inputs.
2) *Processing forwarded inputs* from $(n-\frac{n}{l})$ shadow entities that interact with active entities: e.g., an active entity A can be hit by an attack of a shadow entity S which results in the server responsible for S sending a message about the attack to A's server which updates the state of A (i.e., lowering A's health). Processing forwarded inputs requires: (i) time $t_{\text{fa\_dser}}(n,m)$ for asynchronous reception and deserialization of forwarded inputs from other servers, and (ii) time $t_{\text{fa}}(n,m)$ for applying forwarded inputs.
3) *Updating NPCs*, which requires time $t_{\text{npc}}(n,m)$ for calculating interactions between NPCs and users. We assume that NPCs within one zone are equally distributed on all replicas, hence, each server needs to process $\frac{m}{l}$ NPCs. Since the actual value of $t_{\text{npc}}(n,m)$ depends on the application logic, this parameter is included in our model, but will be neglected in the remainder of this paper for brevity.
4) *Sending application state updates* to $\frac{n}{l}$ connected users, which requires: (i) time $t_{\text{aoi}}(n,m)$ for computing the area of interest, and (ii) time $t_{\text{su}}(n,m)$ for computing and serializing state updates for each user. We assume that state update filtering is applied before sending state updates to the connected users, in order to determine the visible changes and reduce the used network bandwidth.

Altogether, the tick duration $T(l,n,m)$ of common ROIA is modeled as:

$$\begin{aligned} T(l,n,m) \;=\; & \frac{n}{l}\cdot(t_{\text{ua\_dser}}(n,m)+t_{\text{ua}}(n,m) \\ & \quad + t_{\text{aoi}}(n,m)+t_{\text{su}}(n,m)) \\ & +(n-\frac{n}{l})\cdot(t_{\text{fa\_dser}}(n,m)+t_{\text{fa}}(n,m)) \\ & +\frac{m}{l}\cdot t_{\text{npc}}(n,m) \end{aligned} \tag{1}$$

We can now determine the maximum number of users $n_{\text{max}}(l,m,U)$ for a given number of l replicas ($l \geq 1$) and an upper threshold U for the tick duration:

$$n_{\text{max}}(l,m,U) = max\{n \in \mathbb{N} \mid T(l,n,m) < U\} \tag{2}$$

Due to the increasing replication overhead, we specify a minimum improvement that we expect from adding another resource, in order to avoid adding more and more resources without any (or only marginal) benefit. For this purpose, we introduce a constant factor $0 < c \leq 1$ that reflects how much a further replication should increase the maximum user number as compared to the basic value of a single application server $n_{\text{max}}(1,m,U)$. Then, the minimum improvement (in terms of user number) expected from adding replica l is expressed as $n'_{\text{max}} = n_{\text{max}}(l-1,m,U) + c\cdot n_{\text{max}}(1,m,U)$. The maximum number of replicas $l_{\text{max}}(m,U,c)$ can be then computed as the maximum number of replicas for which the tick duration does not exceed threshold U given n'_{max} users and m NPCs:

$$\begin{aligned} l_{\text{max}}(m,U,c) \;&=\; max\{l \geq 1 \mid T(l,n'_{\text{max}},m) < U\}, \text{ with} \\ n'_{\text{max}} \;&=\; n_{\text{max}}(l-1,m,U) + c\cdot n_{\text{max}}(1,m,U) \end{aligned} \tag{3}$$

B. Achieving Scalability using User Migration

User migration causes additional overhead required for switching user connections between application servers. Since this overhead is application-specific, we introduce two parameters, $t_{\text{mig\_ini}}(n)$ and $t_{\text{mig\_rcv}}(n)$, that denote the time for initiating and receiving user migration data on an application server with n users. For a migration from server A (source) to server B (target), $t_{\text{mig\_ini}}(n)$ denotes the time that server A requires for initiating user migration and $t_{\text{mig\_ini}}(n)$ denotes the time that server B requires for receiving migration data.

Based on the computational tasks during the real-time loop identified in Section III-A, our model computes a threshold for the maximum number of migrations per second. Since user migration is typically used if users are not distributed equally on all replicas, we need to consider the number of active entities that are managed by the involved servers in order to compute the current workload of servers.

Considering non-equal user distribution on multiple replicas, the tick duration for each application server responsible for n users, including a active entities, and m NPCs distributed equally on l replicas, is modeled as:

$$\begin{aligned} T(l,n,m,a) \;=\; & a \cdot (t_{\text{ua\_dser}}(n,m) + t_{\text{ua}}(n,m) \\ & \quad + t_{\text{aoi}}(n,m) + t_{\text{su}}(n,m)) \\ & + (n-a) \cdot (t_{\text{fa\_dser}}(n,m) + t_{\text{fa}}(n,m)) \\ & + \frac{m}{l} \cdot t_{\text{npc}}(n,m) \end{aligned} \tag{4}$$

Typically, the overhead for initiating migrations differs from the overhead for receiving migrations; hence, we use two functions, $x^{\text{ini}}_{\text{max}}(l,n,m,a,U)$ and $x^{\text{rcv}}_{\text{max}}(l,n,m,a,U)$, for the maximum number of migrations that should be initiated and received per second for a given number of l replicas ($l \geq 1$), n users, m NPCs, a active entities and an upper threshold U for the tick duration:

$$\begin{aligned} x^{\text{ini}}_{\text{max}}(l,n,m,a,U) \;=\; & max\{x \in \mathbb{N} \mid T(l,n,m,a) \\ & \quad + x \cdot t_{\text{mig\_ini}}(n) < U\} \\ x^{\text{rcv}}_{\text{max}}(l,n,m,a,U) \;=\; & max\{x \in \mathbb{N} \mid T(l,n,m,a) \\ & \quad + x \cdot t_{\text{mig\_rcv}}(n) < U\} \end{aligned} \tag{5}$$

The left-hand side of Fig. 2 presents an example scenario in which the scalability model is used for achieving an equal distribution of overall 45 users across three replicas. For each replica, the maximum numbers of migrations that can be initiated ($x^{\text{ini}}_{\text{max}}$) and received ($x^{\text{rcv}}_{\text{max}}$) are calculated. Then, users are migrated from the replica with the highest number a of active entities to other replicas in order to achieve a user distribution of 15 users on each replica. During migration, each replica only initiates/receives a maximum of $x^{\text{ini}}_{\text{max}}/x^{\text{rcv}}_{\text{max}}$ user migrations per second as calculated by the scalability model: e.g., on the left-hand side of Fig. 2, the upper replica migrates 1 user to the middle replica and 4 users to the lower replica, since it can only initiate 5 user migrations per second. The remaining users are migrated in the second step, as illustrated on the right-hand side of the figure.

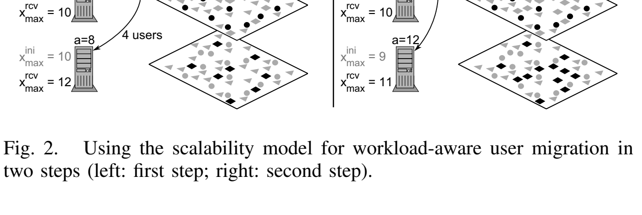

Fig. 2. Using the scalability model for workload-aware user migration in two steps (left: first step; right: second step).

C. Applying the Scalability Model to a particular application

Since our scalability model is based on four major tasks that are common to ROIA processing, it provides generic formulas for calculating thresholds for (i) the maximum number of replicas for a given application (Eq. (3)), (ii) the maximum number of users for a given number of replicas (Eq. (2)), and (iii) the maximum number of migrations per second (Eq. (5)). In order to apply the scalability model for a particular ROIA, the application-specific values of parameters $t_{\text{ua\_dser}}$, t_{ua}, $t_{\text{fa\_dser}}$, t_{fa}, t_{npc}, t_{aoi}, t_{su}, U and c have to be determined. Then, the scalability model determines the particular thresholds (i)-(iii) for this application. Since different applications have different values for these parameters, the scalability model calculates individual thresholds for each application.

While the choice of parameter c depends on economic aspects, which are not within the scope of this paper, the choice of threshold U for the tick duration depends on the characteristics of the particular application. For example, fast-paced action games require lower thresholds U than online role-playing or real-time strategy games. Suitable values for parameter U can be determined by conducting benchmark experiments [3, 9] or by performing user surveys [6, 17] to investigate how particular values of U affect the users' quality of experience. The remaining parameters of our model need to be measured during a test execution of the application.

For example, a first-person shooter game typically requires 25 or more updates per second in order to not disturb the users' immersion, which leads to a threshold for the tick duration of 40 ms [19]. Complex physics calculations on the servers and a large area of interest for each user may lead to high values for t_{ua}, t_{aoi} and t_{su}. The effort for (de-)serialization primarily depends on the size of data that needs to be (de-)serialized. Since the large area of interest leads to large update sizes, the values for parameters t_{ua_dser}, t_{fa_dser} and t_{su} would be high as well. Hence, our model would calculate low thresholds (i)-(iii) for this application.

In contrast, a role-playing game typically requires less updates per second, hence, a higher tick duration may be tolerated (up to 1.5 s) [20]. Since role-playing games typically require the user to select a target explicitly and to choose among a set of predefined interactions, the processing of user inputs may be not as complex as in the example above, leading to lower values for t_{ua} and higher thresholds (i)-(iii).

Basically, our model can be applied to any ROIA that is implemented according to the generic real-time loop model and deals with the four major tasks described in Section III-A. In order to simplify model parameter determination for a particular application, we implemented measurement and logging mechanisms for parameters t_{ua_dser}, t_{fa_dser}, t_{su}, t^{rcv}_{mig} and t^{ini}_{mig} in RTF. Since RTF provides generic mechanisms for (de-)serialization and user migration, these parameter values can be measured inside RTF regardless of the application logic. Since parameters t_{ua}, t_{aoi} and t_{fa} depend heavily on the application logic, they need to be measured manually in the application source code.

IV. ROIA Resource Management with RTF-RMS

We employ the scalability model described in Section III for dynamic load balancing in our resource management system RTF-RMS as an alternative to triggering load-balancing actions in static intervals, without taking into account the exact workload of the application servers. While the static approach causes an unnecessarily high amount of additional workload which may lead to a lower application performance, our scalability model takes into account the dynamic workload of the application servers at runtime for triggering load-balancing actions.

In RTF-RMS, the application provider specifies performance requirements by defining upper and lower thresholds for particular monitoring parameters of interest, e.g., tick duration in ms, as described in Section III-C. During application runtime, if the parameter values are in the interval between the upper and lower threshold, then the application is regarded as not requiring load balancing; otherwise, RTF-RMS chooses between different load-balancing actions.

Fig. 3 illustrates four load-balancing actions in RTF-RMS as explained in the following: (i) user migration, (ii) replication enactment, (iii) resource substitution, and (iv) resource removal.

User migration: Users are migrated from an overloaded server to an underutilized server which is replicating the same zone. User migration is the preferred action if the load of an overloaded server can be compensated by running resources. In the initial implementation of RTF-RMS, user migration was used in each tick to distribute users equally on all application servers for a particular zone. However, continuous migration of users involves an overhead on all servers involved in the migration. The scalability model determines in Eq. (5) a maximum number of migrations that can be initiated ($x^{\text{ini}}_{\text{max}}$) and received ($x^{\text{rcv}}_{\text{max}}$) per second. Using the model, user migration is adjusted to the current workload on the involved servers.

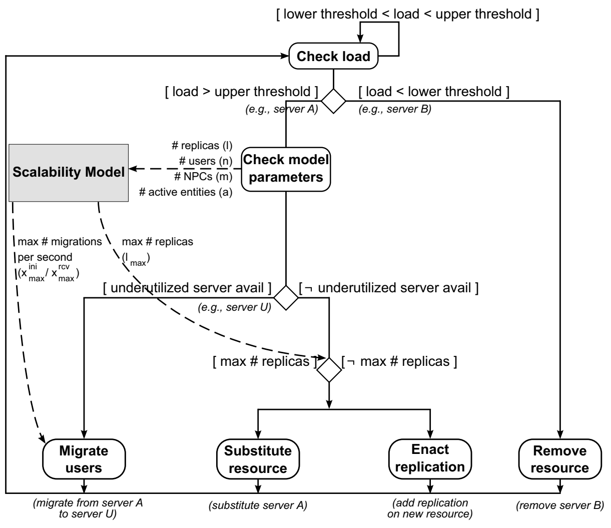
[lower threshold < load < upper threshold]
Check load
[load > upper threshold]
(e.g., server A)
[load < lower threshold]
(e.g., server B)
# replicas (l)
# users (n)
# NPCs (m)
# active entities (a)
Scalability Model
Check model parameters
max # migrations per second ($x^{\text{ini}}_{\text{max}}$ / $x^{\text{rcv}}_{\text{max}}$)
max # replicas (l_{max})
[underutilized server avail]
(e.g., server U)
[¬ underutilized server avail]
[max # replicas]
[¬ max # replicas]
Migrate users
Substitute resource
Enact replication
Remove resource
(migrate from server A to server U)
(substitute server A)
(add replication on new resource)
(remove server B)

Fig. 3. RTF-RMS chooses between four different adaptation strategies.

Replication enactment: New servers are added in order to provide more computation power to highly frequented zones. Given n users distributed equally on l replicas, RTF-RMS adds a new server and then migrates $\frac{n}{l(l+1)}$ users from each replica to the new replica in order to distribute users equally on all $(l+1)$ servers and balance the load. Since the replication overhead increases for each new replica, RTF-RMS uses a threshold for the maximum number of replicas per zone for a particular application. If the number of active replicas for a zone is below the threshold, then replication is used; otherwise, the resource substitution strategy (described next) is preferred. In the initial implementation of RTF-RMS, the developer had to specify the threshold for the maximum number of replicas manually, which was a complex task since the replication overhead depends on the inter-server communication in a particular application and, hence, may be different for each application. The scalability model determines in Eq. (3) a suitable value l_{max} for the maximum number of replicas.

Resource substitution: A running server is substituted by a more powerful resource in order to increase the computation power for highly frequented zones. For this purpose, RTF-RMS replicates the targeted zone on the new resource and migrates all users to the new server. The substituted server is shut down. If no more powerful resource is available for substitution, then the application has reached a critical user density in the same geographical area, which cannot be resolved by the generic adaptation strategies of RTF-RMS. In this case, the application requires redesign, see [13].

Resource removal: An underutilized server is removed from the application processing if its zone is replicated on other servers. In this case, users are migrated from the underutilized server to the other servers of this zone, after which the underutilized server is shut down; otherwise, nothing happens since each zone must be assigned to at least one server.

Our scalability model helps finding suitable thresholds for the maximum number of migrations per second and replicas per zone. While the user migration and replication enactment directly benefit from these thresholds, the remaining load-balancing actions also benefit from finding a threshold for the maximum number of migrations per second, since resource substitution and resource removal also involve user migration.

Listing 1 demonstrates the implementation of user migration in RTF-RMS using our scalability model. Since one server can initiate user migrations to different replicas, RTF-RMS must consider the overall number of concurrent user migrations. For each zone, RTF-RMS determines one server $\texttt{s}_{\texttt{max}}$ with currently the highest number of users and initiates user migrations from $\texttt{s}_{\texttt{max}}$ to the other servers for this zone, taking into account: (i) the deviation of their user counts from the average value, (ii) the maximum number of migrations that can be initiated by server $\texttt{s}_{\texttt{max}}$, and (iii) the maximum number of migrations that can be received by each remaining server. Then, $\texttt{s}_{\texttt{max}}$ initiates $\texttt{x}^{\texttt{ini}}_{\texttt{max}}$ migrations to the remaining servers, in order to distribute users equally on all servers, but each server `s[i]` receives not more than $\texttt{x}^{\texttt{rcv}}_{\texttt{max}}\texttt{[i]}$ migrations, i.e., not more than it can handle without violating performance requirements as calculated by the model.

Listing 1. User migration in RTF-RMS using the scalability model.

```
// n: number of users on all replicas
// s[]: array of all replicas for a zone
// $s_{max}$: server with highest user count
avg = n / s.length;

// (i): Calc deviation of user count for
   each server
for (i = 0; i < s.length; i++)
 d[i] = avg - s[i].userCnt;

// (ii): Calc $x^{ini}_{max}$ for server $s_{max}$
$x^{ini}_{max}$ = $max_{x \in \mathbb{N}}${T(l,n,m,$s_{max}$.userCnt)
             + x*$t_{mig\_ini}(n)$ < U};

// (iii): Calc $x^{rcv}_{max}$ for remaining servers
for (i = 0; i < s.length; i++)
 $x^{rcv}_{max}$[i] = $max_{x \in \mathbb{N}}${T(l,n,m,s[i].userCnt)
                 + x*$t_{mig\_rcv}(n)$ < U};

for (i = 0; i < s.length; i++) {
 // if server i has < avg users, then
    migrate users to it
 if (d[i] > 0)
  // migrate min{d[i], $x^{rcv}_{max}$[i], $x^{ini}_{max}$} users
     from $s_{max}$ to i
 else
  continue; }
```

V. Experiments

In order to evaluate our scalability model and its implementation, we conducted several scalability experiments with a first-person shooter game, called RTFDemo, that was developed using RTF and employs replication for the distribution of the game state. Our previous work [18] demonstrated that in RTFDemo, the users' gaming experience is negatively affected if users receive less than 25 updates per second, i.e., the tick duration exceeds 40 ms, over a longer time period.

In our evaluation, we focus on how replication enactment and user migration can distribute additional workload caused by increasing numbers of users. First, we discuss in detail how the scalability model is instantiated for RTFDemo, i.e., how model parameters are determined, and we illustrate how RTF-RMS' load balancing is affected by the output of the scalability model. Then, we demonstrate how the improved RTF-RMS load-balancing strategy can efficiently manage resources for a running RTFDemo session with a continuously changing number of users. In our experiments, we use non-virtualized hardware resources for the application servers: Intel Core Duo PCs with 2.66 GHz and 4 GB of RAM running Ubuntu 11.04.

A. Determining Model Parameters

In order to determine the maximum number of replicas for RTFDemo, we calculate how many users can be accommodated by a given number of replicas, using Eq. (1)-(3). However, the interactivity between users influences this number, since highly interactive users may produce a significantly higher workload than the same number of users that only interact infrequently. In order to simulate an average workload, we use randomly interacting, computer-controlled bots for our experiments. In order to compute particular values for the parameters of our model, we connect up to 300 bots to two application servers replicating the same zone. We distribute bots equally on both servers, in order to simulate a high amount of inter-server communication for sending interactions between users connected to different servers (forwarded inputs).

In our experiments, we measure the CPU time for parameters $t_{\text{ua\_dser}}$, t_{ua}, $t_{\text{fa\_dser}}$, t_{fa}, t_{aoi} and t_{su} in Eq. (1). However, measured values typically have a high variation due to frequently changing interactivity between users. In order to compute average values for each of the above parameters, we calculate an approximation function for each parameter using the nonlinear least-squares Levenberg-Marquardt algorithm [12] implemented in the visualization tool `gnuplot` [22]. The Levenberg-Marquardt algorithm calculates the coefficients c_i of a given function $f(x) = \sum_{i=0}^{m} c_i * x^i$, such that the resulting function optimally fits the data points (x_i, y_i) from our measurements, i.e., $\sum_{i=0}^{m}(f(x_i) - y_i)^2$ becomes minimal. We have to choose an appropriate function (linear, polynomial, etc.) for approximation in order to get a good result. In the following, we describe for each parameter which function we choose for approximation and illustrate them in Fig. 4.

During each tick in RTFDemo, each user can issue a move command, an attack command or both commands. Users typically send move commands regardless of the overall user number; the frequency of attack commands increases for higher user numbers due to a higher number of potential targets. While it is difficult to find a generic dependency between the user number and the number of attack commands, we observed that the number of attack commands in RTFDemo increases almost linearly for higher user numbers. For each attack command, the application server needs to iterate through all users in order to check which users are hit by the attack. For this reason, t_{ua} grows faster than any linear function for higher user numbers. Therefore, we choose a polynomial function $f(x) = c_1x^2 + c_2x + c_3$ for approximating t_{ua} where x denotes the number of users. We use the Levenberg-Marquardt algorithm to calculate coefficients c_1, c_2, c_3. The resulting function t_{ua} is illustrated in Fig. 4. In the following, the Levenberg-Marquardt algorithm is also used for calculating the coefficients of the approximation functions for parameters $t_{\text{ua\_dser}}$, $t_{\text{fa\_dser}}$, t_{fa}, t_{aoi} and t_{su}.

Each user input has to be deserialized once by the application server. Deserialization of the same input type (move or attack command) requires the same amount of CPU time. Since the number of attack commands increases linearly for higher user numbers, the CPU time $t_{\text{ua\_dser}}$ required per user for deserialization also increases linearly. Thus, we choose a linear function for approximation of parameter $t_{\text{ua\_dser}}$.

In the RTFDemo application, there is only one type of forwarded inputs, i.e., attack commands between active and shadow entities. In our experiments for parameter determination, each server has the same number of active and shadow entities. Since users cannot differentiate between active and shadow entities, both are attacked with equal frequency. Hence, the number of forwarded inputs increases linearly for higher user numbers. Therefore, we choose linear functions for approximation of parameters t_{fa} and $t_{\text{fa\_dser}}$. However, the number of forwarded inputs in RTFDemo is quite small (in the order of tens of forwarded inputs per tick for 300 bots), which results in a very short CPU time for t_{fa} and $t_{\text{fa\_dser}}$ as compared to the other parameters of our model. For this reason, we omit graphs for t_{fa} and $t_{\text{fa\_dser}}$ in Fig. 4.

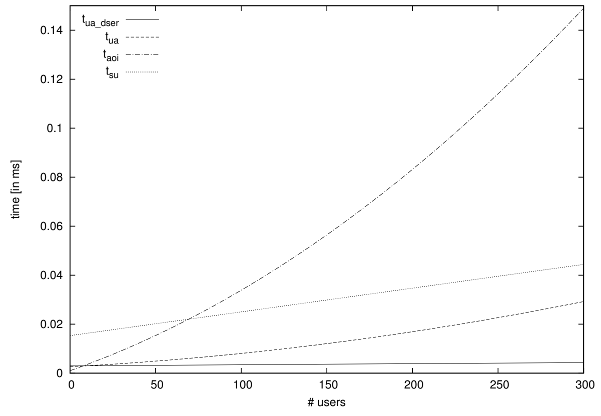

Fig. 4. Model parameters for replication in the RTFDemo application.

In order to compute the area of interest for a user, RTFDemo employs the Euclidean Distance Algorithm for calculating which changes in the virtual environment are visible for this user [5]. For user U, it has to be checked for all users whether they are in the visibility area of user U, i.e., the application iterates through all users (except for U). Each user in the visibility area of user U is subscribed to the update list of user U (denoting all users for which user U receives updates from its server); for each subscription, RTFDemo iterates through the update list in order to avoid duplicate entries that would cause user U receiving multiple, identical updates. Thus, we choose a polynomial function $f(x) = c_1x^2+c_2x+c_3$ for approximating parameter t_{aoi} which denotes the CPU time for computing the area of interest of one user.

At the end of each tick, each server computes and serializes state updates for its users. Update computation requires more time for higher user numbers, since more users typically lead to more interactivity, which results in more changes in the virtual environment that need to be sent to each user. Moreover, each state update has to be serialized once by the application server. Since more changes in the virtual environment lead to larger update sizes, the state update serialization requires more CPU time for higher user numbers. We assume that the CPU time for serialization increases linearly for larger updates, since updates in RTFDemo simply aggregate equivalent data types, rather than creating new data types in each update, which could require a completely different CPU time for serialization. Thus, we choose a linear function for approximating parameter t_{su}.

After determining approximation functions for each parameter in Eq. (1), we can calculate n_{max} using Eq. (2); for RTFDemo, we take an upper threshold of 40 ms for tick duration. In order to calculate a maximum number of replicas in Eq. (3), we have to choose a value $c \in]0, 1]$ as introduced in Section III-A. Values close to 0 would lead to a large maximum value for the number of replicas (e.g., $l_{\text{max}} = 48$ for $c = 0.05$), while values close or equal to 1 would lead to $l_{\text{max}} = 1$ due to the replication overhead. The choice of c is application-specific and depends also on the resource costs and business aims of the application provider. For further measurments, we choose a “compromise” value of $c = 0.15$ which results in $l_{\text{max}} = 8$.

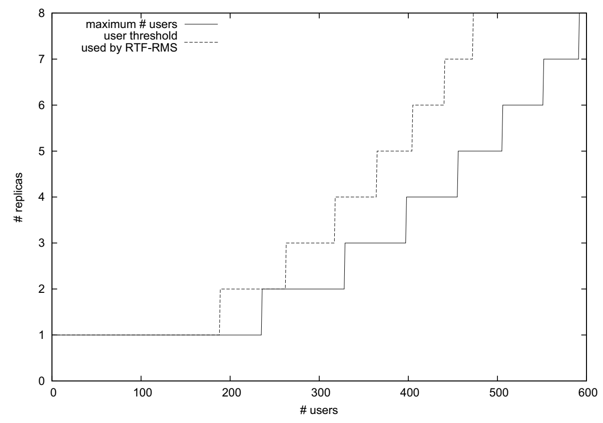

Fig. 5. The effect of replication on scalability of the RTFDemo application.

Fig. 5 shows how many replicas are required for a given number of users before violating performance requirements (denoted as “maximum # users” in the figure). Since migrating users to the new replica puts additional load on servers, RTF-RMS triggers replication enactment as soon as a particular percentage of the maximum number of users has been reached. Our empiric observations show that replication enactment should be triggered at 80% of the maximum number of users (dashed line in Fig. 5) in order to compensate for the load overhead from migrating users and potentially additional users connecting to the application. RTF-RMS uses this threshold for triggering replication enactment in RTFDemo; e.g., a single server could compute up to 235 users, but adding a new replica and initiating user migrations on a fully-loaded server with 235 users would lead to violation of performance requirements due to the computational overhead caused by user migration.

Hence, RTF-RMS activates a second replica if the user number exceeds 188 users, i.e., 80% of 235 users.

In order to determine the maximum number of user migrations per second, we measure the CPU time for parameters $t_{\text{mig\_ini}}$ and $t_{\text{mig\_rcv}}$ of Eq. (4) while issuing user migrations between two servers for different numbers of users. We observe that the CPU time for both parameters increases almost linearly for higher user numbers due to a higher workload on the servers, so we choose linear approximation functions for these parameters. The resulting functions illustrated in Fig. 6 show that the CPU time for initiating migrations is higher than for receiving migration requests in RTFDemo.

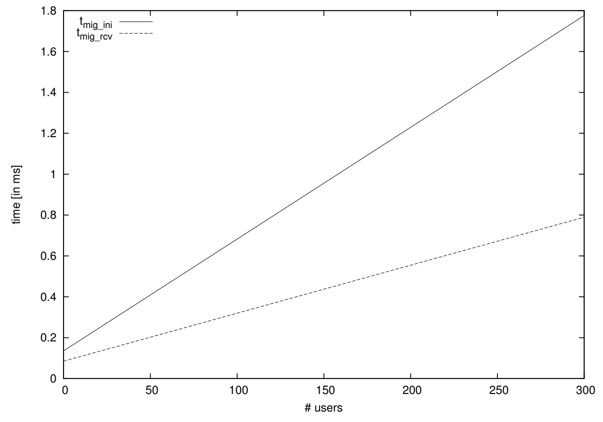

Fig. 6. Model parameters for user migration in the RTFDemo application.

Fig. 7 shows the output of the scalability model considering user migration, i.e., how many user migrations can be initiated ($x^{\text{ini}}_{\text{max}}$) and received ($x^{\text{rcv}}_{\text{max}}$) per second for a given tick duration without violating performance requirements. For a given tick duration, RTF-RMS performs $\min\{x^{\text{ini}}_{\text{max}}, x^{\text{rcv}}_{\text{max}}\}$ user migrations to avoid the violation of performance requirements on any involved server.

For example, having a user distribution of 180 users on server A with tick duration 35 ms (i.e., heavy load) and 80 users on server B with tick duration 15 ms (i.e., light load), one might intuitively expect that load balancing should prefer a significant user migration, e.g., migrating 50 users per second from server A to server B. However, our scalability model predicts that 50 migrations per second would lead to the violation of performance requirements due to the migration overhead. Hence, RTF-RMS will initiate 3 user migrations per second from server A to server B to equalize the workload since $\min\{x^{\text{ini}}_{\text{max}}, x^{\text{rcv}}_{\text{max}}\} = \min\{3, 34\} = 3$ (according to Fig. 7). As the tick duration changes on both servers due to user migration, the number of migrations per second is increased by RTF-RMS; e.g., if the tick duration on server A decreases to 30 ms (160 users), RTF-RMS will perform $\min\{5, 29\} = 5$ user migrations per second.

B. Evaluating Predictions from the Scalability Model

This experiment demonstrates how RTF-RMS uses the scalability model for load balancing of an RTFDemo session

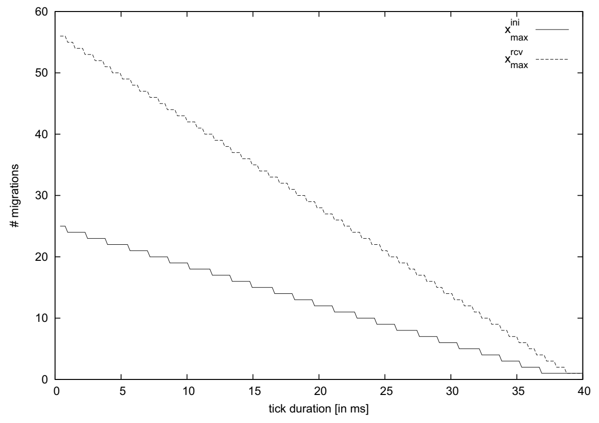

Fig. 7. Scalability Model output: Number of user migrations for the RTFDemo application.

with a continuously changing number of users (up to 300). RTF-RMS distributes users equally on all servers using the migration thresholds from Fig. 7 and adds/removes replicas according to the replica thresholds from Fig. 5 (dashed line).

The experimental results are presented in Fig. 8 which shows the overall number of users connected to the application and the average CPU load of all servers that are currently used by RTF-RMS. The figure illustrates that the CPU load grows initially with the number of users and that each replication enactment significantly reduces the average CPU load.

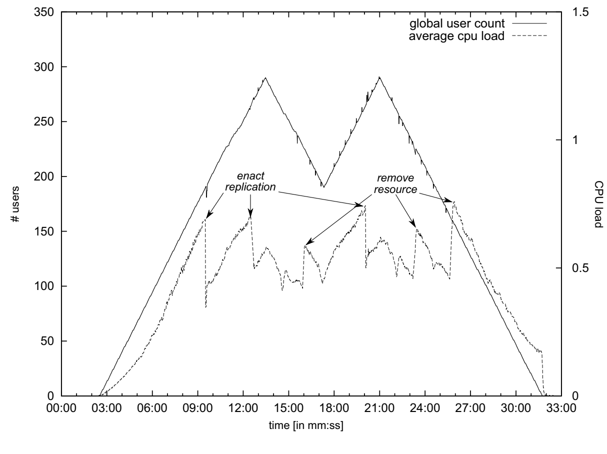

Fig. 8. Dynamic load balancing of the RTFDemo application for a changing number of users.

Fig. 8 shows that the servers are not fully loaded, i.e., the average CPU load stays below 100%. While one might intuitively expect that this behavior is not desirable due to the non-optimal resource utilization, RTF-RMS intentionally causes this behavior by using the replica thresholds from Fig. 5 in order to avoid violation of performance requirements caused by migration overhead and by additional users potentially connecting to the application during load balancing.

During this experiment, the tick duration on all application servers did not exceed 40 ms, i.e., performance requirements were not violated. This proves that RTF-RMS did not remove resources too early and users were distributed appropriately on all application servers using user migration.

VI. Related Work and Conclusion

This paper develops a novel analytical model that improves the load balancing for Real-Time Online Interactive Applications (ROIA) on distributed platforms like Grids and Clouds. Our model improves the load balancing for ROIA, in particular for multi-player online games. The automatic load balancing at runtime based on our prediction model is a promising alternative to the current practice of overprovisioning computing resources in the attempt to avoid negatively affecting users' quality of experience at all costs. However, permanent and static overprovisioning of computing resources is not efficient and makes it difficult for small companies to enter the market.

The challenging research area of load balancing for online games has been studied by many researchers. Several existing approaches, like [11], balance the load by partitioning the virtual environment into small areas which are each computed by one application server and changing the distribution of areas to application servers dynamically during runtime according to the current user distribution. This approach is particularly suitable for clustered user distributions, i.e., few areas of the virtual environment have very high user density while large parts of the virtual environment remain unused by the users. The replication approach targeted by our scalability model divides the virtual environment into zones which are typically larger areas, and it allows providers to use multiple application servers for the computation of a single zone via its replication. Our approach is typically more suitable for uniform user distributions. While both approaches involve overhead for inter-server communication, our proposed scalability model enhances the usability of the replication approach. Our experiments with RTF-RMS demonstrate the suitability of the replication approach for load balancing of an online game with a continuously changing number of users.

In [16], the authors model the resource usage of online games that are using the replication approach. Their model combines CPU, network and memory usage and utilizes neural networks for predicting the user count. In contrast to our model, they do not address the additional workload caused by user migration and they do not analyze the overall limitations of ROIA scalability, i.e., what is the maximum number of resources that can be used efficiently.

Several existing approaches, like [4, 7], identified the demand to limit the number of user migrations and proposed efficient user-migration strategies. Our work differs from these approaches by considering the current workload of the application servers, rather than assigning static thresholds to each server. In [7], the authors define a static threshold denoting the maximum number of users that can be handled by each server. In [4], the authors allocate the load on heterogeneous server resources proportionally to each server's networking bandwidth. Our scalability model considers the tick duration representing the current workload on each server for user migration: this is more suitable for avoiding performance requirement violations since the same number of users can interact with different frequencies causing different workloads.

In [10], a traffic analysis of online games was presented that revealed an asymmetry between the bandwidth used for incoming and outgoing server messages and showed the importance of distinguishing between both. While we still need to implement bandwidth analysis for our scalability model, our model distinguishes between processing of incoming events and outgoing state updates. Furthermore, the authors showed a strong relationship between the number of users and bandwidth usage, which implies that our approach of calculating a maximum number of users for a given number of replicas is also suitable for modelling network traffic in ROIA.

We have shown how our scalability model addresses four major computational tasks in ROIA processing. In a practical case study of a dynamic multi-player online game, we demonstrated the practical relevance of our scalability model by incorporating it into the resource management system RTF-RMS. This case study illustrates how our model improves RTF-RMS's load-balancing strategy by limiting the number of replicas and the number of user migrations per second in order to avoid violation of performance requirements caused by computational overhead from load balancing. Our experiments illustrate how model parameters can be determined for a particular application and demonstrate that RTF-RMS uses the scalability model successfully for efficient resource management of a running ROIA session without violating performance requirements.

In our future work, we will extend the evaluation of our scalability model using heavier user workloads, as well as modern server hardware and Cloud resources.

Acknowledgment

We are grateful to the anonymous reviewers for their helpful remarks on the preliminary version of the paper. Our research has received funding from the EC's 7th Framework Programme under grant agreements 318665 (OFERTIE) and 295222 (MONICA).

References

[1] "Amazon Web Services (AWS)," http://aws.amazon.com/game-hosting, 2013.

[2] "Real-Time Framework (RTF)," http://www.real-time-framework.com, 2013.

[3] A. Abdelkhalek, A. Bilas, and A. Moshovos, "Behavior and Performance of Interactive Multi-Player Game Servers," *Cluster Computing*, vol. 6, no. 4, pp. 355–366, 2003.

[4] C. E. Bezerra and C. F. Geyer, "A load balancing scheme for massively multiplayer online games," *Multimedia Tools and Applications*, vol. 45, no. 1-3, pp. 263–289, 2009.

[5] J.-S. Boulanger, J. Kienzle, and C. Verbrugge, "Comparing Interest Management Algorithms for Massively Multiplayer Games," in *Proc. of 5th ACM SIGCOMM workshop on Network and system support for games*, ser. NetGames '06. ACM, 2006.

[6] M. Dick, O. Wellnitz, and L. Wolf, "Analysis of Factors Affecting Players' Performance and Perception in Multiplayer Games," in *Proc. of 4th ACM SIGCOMM workshop on Network and system support for games*, ser. NetGames '05. ACM, 2005, pp. 1–7.

[7] T. N. B. Duong and S. Zhou, "A Dynamic Load Sharing Algorithm for Massively Multiplayer Online Games," in *Proc. of 11th IEEE International Conference on Networks (ICON2003)*, 2003, pp. 131–136.

[8] F. Glinka, A. Ploss, S. Gorlatch, and J. Müller-Iden, "High-Level Development of Multiserver Online Games," *International Journal of Computer Games Technology*, vol. 2008, no. 5, pp. 1–16, 2008.

[9] T. Henderson and S. Bhatti, "Networked games: a QoS-sensitive application for QoS-insensitive users?" in *Proc. of the ACM SIGCOMM Revisiting IP QoS workshop*. ACM, 2003, pp. 141–147.

[10] J. Kim, J. Choi, D. Chang, T. Kwon, Y. Choi, and E. Yuk, "Traffic Characteristics of a Massively Multi-player Online Role Playing Game," in *Proc. of 4th ACM SIGCOMM workshop on Network and system support for games (NETGAMES'05)*. ACM, 2005, pp. 1–8.

[11] J. Lim, J. Chung, J. Kim, and K. Shim, "A Dynamic Load Balancing for Massive Multiplayer Online Game Server," in *Entertainment Computing - ICEC 2006*, ser. Lecture Notes in Computer Science, vol. 4161. Springer, 2006, pp. 239–249.

[12] D. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963.

[13] D. Meiländer, A. Bucchiarone, C. Cappiello, E. D. Nitto, and S. Gorlatch, "Using a Lifecycle Model for Developing and Executing Real-Time Online Applications on Clouds," in *Service-Oriented Computing – ICSOC 2011 Workshops*, ser. Lecture Notes in Computer Science, vol. 7221, 2011, pp. 33–43.

[14] D. Meiländer, A. Ploss, F. Glinka, and S. Gorlatch, "A Dynamic Resource Management System for Real-Time Online Applications on Clouds," in *Euro-Par 2011: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science, vol. 7155, 2012, pp. 149–158.

[15] J. Müller and S. Gorlatch, "GSM: A Game Scalability Model for Multiplayer Real-time Games," in *IEEE Infocom 2005*. IEEE Communications Society, 2005, pp. 2044–2055.

[16] V. Nae, A. Iosup, and R. Prodan, "Dynamic Resource Provisioning in Massively Multiplayer Online Games," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 3, pp. 380–395, 2011.

[17] L. Pantel and L. C. Wolf, "On the Impact of Delay on Real-Time Multiplayer Games," in *Proc. of the 12th international workshop on Network and operating systems support for digital audio and video*, ser. NOSSDAV '02. ACM, 2002, pp. 23–29.

[18] A. Ploss, D. Meiländer, F. Glinka, and S. Gorlatch, "Towards the Scalability of Real-Time Online Interactive Applications on Multiple Servers and Clouds," *Advances in Parallel Computing*, vol. 20, pp. 267–287, 2011.

[19] S. Ratti, B. Hariri, and S. Shirmohammadi, "A Survey of First-Person Shooter Gaming Traffic on the Internet," *Internet Computing, IEEE*, vol. 14, no. 5, pp. 60–69, 2010.

[20] P. Svoboda, W. Karner, and M. Rupp, "Traffic Analysis and Modeling for World of Warcraft," in *IEEE International Conference on Communications (ICC'07)*, 2007, pp. 1612–1617.

[21] L. Valente, A. Conci, and B. Feijó, "Real Time Game Loop Models for Single-Player Computer Games," in *SBGames '05 – IV Brazilian Symposium on Computer Games and Digital Entertertainment*, 2005.

[22] T. Williams, C. Kelley *et al.*, "Gnuplot 4.4: an interactive plotting program," http://gnuplot.sourceforge.net/, 2010.