Scalability in Peer-to-Peer-based MMVEs: The Continuous Events Approach

Florian Heger, Gregor Schiele, Richard Süselbeck, Laura Itzel and Christian Becker
University of Mannheim
Mannheim, Germany
{ florian.heger | gregor.schiele | richard.sueselbeck | laura.itzel | christian.becker }@uni-mannheim.de

***Abstract*—A key challenge for Peer-to-Peer-based Massively Multi-user Virtual Environments is efficient delivery of updates. In contrast to centralized approaches, which can efficiently calculate changes and deliver these, peer-based approaches have to generate updates from arbitrary participating peers.**

In this paper we present an approach to reduce update messages in Peer-to-Peer-based Massively Multi-user Virtual Environments introducing a distinct class of update events, which we call *continuous events*. A continuous event results in a series of updates that can be pre-computed. By distributing the event series as a change function, affected peers can calculate and perform updates after receiving a continuous event without further update messages. We present a system design enabling the distribution and computation of continuous events and discuss basic algorithms for continuous event management. We also show simulation results which confirm a substantial reduction of update messages.

I. Introduction

A key challenge for Peer-to-Peer-based Massively Multi-user Virtual Environments (P2P-based MMVEs) is the efficient propagation of updates from peers that alter the state of the virtual environment to all peers that are interested in this state. Centralized approaches can reduce message load by computing a global state and distributing an aggregated message containing several updates. However, this is not possible for P2P-based MMVEs, where each update often leads to a number of messages which are sent to remote peers, considerably reducing the system's scalability.

To lower the message overhead and thus increase scalability, we propose a new concept for update propagation in P2P-based MMVEs: *continuous events*.

MMVEs include user actions which can result in more than a single state change. For example, a rocket which has been fired by a user, changes its position over time as it moves toward its target. A way to represent this in a P2P-MMVE system is to create a rocket object whose management is assigned to a peer. This peer moves the object and sends a new event to all peers which are affected by the rocket and therefore have to maintain a local copy of the rocket object. This leads to a large number of identical position updates, which have to be sent from the peer responsible for the rocket object to the peers affected by the rocket.

Another way to deal with this issue, which is typically used by *dead reckoning* algorithms [**?**][**?**], is to interpret the firing of the rocket and the resulting position changes as a single *continuous action*. Assuming that there is no more interaction with the rocket, the future position changes can be expressed via a mathematical function. The information about the continuous action and the changes can be sent in one event at the time the action is performed. Afterwards, all peers who received the event extrapolate the up-to-date position of the rocket without further sending of events. The rocket object may still be assigned to a certain peer, which checks for possible deviations between the original position and the extrapolated position and resends the information about the state of the rocket object in case a deviation occurs.

We propose to build on this concept, but to use a higher level of abstraction: We decouple the management of future state changes from the objects and introduce an explicit type of event to model continuous actions in a P2P-MMVE system. We call this concept *continuous events*. A continuous event includes *execution code* (i.e. what should be performed locally on a peer which received the continuous event?), *temporal information* (i.e. how long and how often should the code be executed?) and *spatial information* (i.e. objects within which spatial area should be influenced by the executed code?).

Continuous events can be used to send the information about continuous actions to other peers via an aggregated network message. Future state changes can then be calculated locally on these peers. Because of the decoupling of objects and management of state changes, as well as the addition of spatiality, a single continuous event is able to carry the information about future state changes to numerous objects within an area.

In this paper, we describe the basic concept of continuous events and algorithms for continuous event management in a P2P-based MMVE. We think that continuous events may ultimately be used to model very complex state changes of large object numbers over time. For example, an MMVE system which wants to move a large number of objects of a certain type according to a given swarm-like movement pattern, may send the pattern to all peers whose avatars are located in the vicinity by using a single continuous event. Based on the pattern, as well as the temporal and spatial influence information, all peers are able to calculate the movement for these objects within the given area for the given time. Obviously, for such a scenario a lot of algorithms and management functions by the MMVE system are needed.

This paper includes the foundations for our future work towards such complex continuous events.

The paper is structured as follows: First we give an overview of our system model and assumptions in Section II. In Section III we model continuous events and identify system functionalities, which are needed for continuous event support.

Section IV presents our approach and algorithms. In Section V we describe the simulation and present evaluation results. Section VI includes related work. Finally in Section VII we draw conclusions and give a preview on future work.

II. System Model

This work is part of the Peers@Play project [**?**], which is developing a framework for P2P-based MMVEs. The paper builds on existing concepts from this project. In the following, we describe our system model, existing components that we build upon and our main assumptions.

Our system consists of a number of networked end-user computers (the *peers*), which operate a P2P-based MMVE cooperatively. Peers can enter and leave the system at any time. Each peer is able and willing to take over tasks for the MMVE operation like storing a part of the MMVE state and computing state updates for it. To do so, each peer executes a special MMVE software.

The MMVE consists of numerous stateful objects that are placed at specific locations in the virtual space. Users can move and interact in the virtual space using virtual representations of themselves, their so-called *avatars*. If an avatar performs an action (initiated by its user), a corresponding event is generated, which modifies the state of all objects that are affected by the event. As an example, an avatar can pick up an object, affecting the object itself but also all nearby other avatars that can see this action. Events are propagated to other peers based on relevancy. The peers then calculate the state changes of objects based on the events.

We assume the availability of a reliable system service to determine the necessary receiving peers for each event and to perform the actual propagation of the event to them. In the context of the Peers@Play project, this service determines the relevancy of events based on spatial areas in the virtual environment: The *Area of Interest* (AoI) and the *Area of Effect* (AoE) [**?**][**?**]. Every user avatar has an AoI and every event has one or multiple AoEs. A user's peer has to be provided with an event, if at least one of the event's AoEs intersects the AoI of the user's avatar.

The propagation of events in our system can be characterized as *Spatial Publish Subscribe* [**?**]: The users' peers perform spatial AoI subscriptions. Updates are provided by making one or more spatial AoE publications. From a conceptual point of view, in a P2P-based Spatial Publish Subscribe system with *n* peers, the intersection of spatial subscriptions and publications may be done by one up to *n* so-called interest matchers. Our system partitions the virtual environment into disjoint *zones*. For each zone a *coordinator peer* is assigned. The coordinator peer acts as interest matcher for its zone. It receives AoI subscriptions and AoE publications from within the zone, intersects the areas and propagates the events based on the result. Due to space restrictions, we do not go into detail about the partitioning algorithm and the algorithm to select and assign the coordinator peers to zones. More information about our propagation service can be found in [**?**].

In addition to the aforementioned, we make the following assumptions:

1) *Loosely synchronized time* - We assume that time is synchronized loosely between the peers.
2) *No consideration of disconnects* - At the moment, our approach does not consider disconnections of peers. This will be part of our future research.
3) *No consideration of security or cheating* - Our approach includes distributed calculation of crucial MMVE logic on user peers. This may be a potential point of attack for cheaters. While we are certainly aware of potential problems, handling of security is beyond the scope of this paper.
4) *Restriction to a single zone* - For reasons of simplicity and due to space restrictions we focus on a single zone within the virtual environment. From a conceptual point of view, our approach can be extended to multiple zones: Algorithms for the coordination of continuous events whose spatial area is located at zone borders are needed. This will be part of a future publication.

III. Continuous Events

We model a continuous event (CE) as a 6-tuple $CE = \{t_0, \delta t, n, f_{AoE}(t), f_E(t), f_{apply}(AoE, E)\}$:

- t_0 defines the point in time of the first execution of code on the peers which received CE.
- δt describes the interval between every execution of code.
- n defines the number of intervals which have to be completed in case CE is not aborted.
- The function $f_{AoE}(t)$ calculates the AoE of CE at a given point in time t. The result of $f_{AoE}(t)$ can change over time, e.g. the function may return an AoE of a different size or shape after an interval is completed.
- The function $f_E(t)$ calculates the effect E of CE at a given point in time *t*, e.g. a certain damage value or a movement pattern. The result can change over time, e.g. to increase the damage value or to adjust the movement pattern after an interval is completed.
- The function $f_{apply}(AoE, E)$ executes the effect locally on a peer based on the results of $f_{AoE}(t)$ and $f_E(t)$.

Based on this model we identify the following functionalities, which our system has to provide in order to support CEs:

1) *Explicit Use of Continuous Events* - The system has to provide the possibility to create and terminate CEs explicitly.
2) *Execution of Continuous Events on Target Peers* - The system has to execute received CEs locally on the peers.
3) *Management of Existing Continuous Events* - Peers may leave or join the MMVE system and user avatars may move. Both changes the AoI subscriptions of the peers while CEs may still run. Our system has to calculate

up-to-date states of all existing CEs, to check for intersections with new AoI subscriptions and to provide the peers which made these subscriptions with up-to-date states of existing CEs.

IV. System Approach

Concerning functionality 1, we identified two design alternatives: When a CE is created, the code for $f_{AoE}(t)$, $f_E(t)$ and $f_{apply}(AoE, E)$ can either be stored in the CE before sending (alternative A) or a pre-defined selection of code can be provided on each peer and only references are sent (alternative B). A allows to define very generic function code, but results in potentially larger network messages because more information has to be sent. B has a restriction to pre-defined code, but results in potentially smaller network messages because only references have to be sent. Because our focus is on enhancing scalability and therefore also reducing bandwidth usage, we decided to include B into our design. To provide extensibility, our architecture includes a repository for function code on the peers. New code can be inserted via an interface and is distributed to other peers by our system.

CEs are created via a *Continuous Event Interface* (CEI). Values for δt and n have to be assigned. t_0 is determined by the system. References to the code for $f_{AoE}(t)$, $f_E(t)$ and $f_{apply}(AoE, E)$ from the repository have to be given and are stored in the CE.

In addition to the CEI and the code repository, our system includes a *Continuous Event Executor* (CEE), which is responsible for executing received CEs on the peers (functionality 2) and a *Continuous Event Manager* (CEM), which is responsible for managing existing CEs (functionality 3), a *Continuous Event Storage* (CES) for storing CE states between intervals and a *Timer Service* (TS) to register timers.

After a CE is created on a peer, it is first sent to the CEM which is responsible for continuous event management for this peer. From a conceptual point of view, there may be *1* to *n* CEMs in the system and responsibility may be assigned based on different criterias (e.g. by assigning responsibility zones, multicast groups etc.). In order to manage existing CEs, the CEM needs information about which peer already has received a certain CE and about new AoI subscriptions. This information can be retrieved from the existing *Spatial Publish Subscribe Networking Service* (SPS) of our framework, which is also used to send CEs. This results in a frequent information exchange between CEM and SPS. The SPS stores the information about AoI subscriptions for a zone at the coordinator peer. Therefore, we decided to run a CEM at each coordinator peer and to assign continuous event management responsibility according to the existing zones. This avoids additional network messages between CEM and SPS, because the required information can be exchanged between CEM and SPS at the coordinator peer.

Algorithm 1 describes the CEM behaviour after a new CE arrives: The CE is sent by using the SPS. The SPS returns a receiver list, which is used to manage existing events later on. All CEs can be identified by a system-wide unique continuous event ID. The CEM stores the receiver list by using this ID. Afterwards, the CEM first checks if there are missed state changes of the CE, e.g. due to delay, and calculates the up-to-date state. If there is still time left until the end of the CE, the CE state is stored in the CES and a timer is registered at the TS which triggers the next calculation of an up-to-date state by the CEM.

```
Input: continuous event (CE)
Data: continuous event storage (CES), timer service (TS), spatial
      publish subscribe service (SPS)
begin
    receiverList = SPS.Send(CE);
    this.ReceiverLists.Put(CE.EventID, receiverList);
    nextCalc = CE.t_0;
    finalCalc = CE.t_0 + (CE.n * CE.δt);
    while nextCalc < GetMMVETime() and nextCalc < finalCalc do
        calculate AoE and E of CE at nextCalc;
        nextCalc += CE.δt;
    end
    if nextCalc < finalCalc then
        store state of CE in CES;
        ts.Register(new Timer(nextCalc, CE.EventID));
    end
end
```

Algorithm 1: CEM receives a new CE

```
Input: continuous event (CE), point in time (t)
Data: code repository (coderep)
Result: AoE and E of CE
begin
    f_AoE = coderep.Retrieve(CE.f_AoE Reference);
    f_E = coderep.Retrieve(CE.f_E Reference);
    AoE = f_AoE(t);
    E = f_E(t);
    return AoE, E;
end
```

Algorithm 2: Calculation of AoE and E for a CE

Algorithm 2 shows the calculation of AoE and E: The code for f_{AoE} and f_E is retrieved from the repository and the functions are called using a given point in time as arguments in order to calculate up-to-date values for AoE and E.

After a timer triggers, the state of the CE which is registered together with the timer is retrieved from the CES. In the following, the CE is either removed (in case the lifetime has expired), or new values for AoE and E are calculated, the point in time of the next calculation is determined, the state stored and a new timer registered (see algorithm 3).

The SPS continually provides the CEM with information about new AoI subscriptions. In case an information about a new AoI arrives at the CEM, it checks for all stored CEs if the peer which made the new subscription already has received CE and if the AoI intersects the AoE of CE. If the peer did not receive CE and there is an intersection, the peer is added to the receiver list and the CE is send via the SPS (see algorithm 4).

While a CEM is only run by a coordinator peer, every peer runs a CEE. After a CE was received from the SPS, the CEE checks if there are missed calculations of the AoE and E of CE and missed local executions of E and catches up. If there is still time left until the ending of CE, the state of CE is

```
Input: timer (tm) registered with continuous event (CE)
Data: continuous event storage (CES), timer service (TS)
begin
    retrieve state of CE from CES;
    now = GetMMVETime();
    if now >= CE.lastCalc then
        CES.Remove(CE.EventID);
    end
    else
        calculate AoE and E of CE at now;
        nextCalc = now + CE.δt;
        store state of CE in CES;
        TS.Register(new Timer(nextCalc, CE.EventID));
    end
end
```

Algorithm 3: Timer registered by CEM triggers

```
Input: aoi subscription (AoI)
Data: continuous event state storage (CES), spatial publish subscribe
      service (SPS)
begin
    foreach receiverlist in this.ReceiverLists do
        key = this.ReceiverLists.KeyOf(receiverlist);
        retrieve state of CE from CES;
        if AoI.Intersects(AoE) then
            if not receiverlist.Contains(AoI.PeerID) then
                SPS.Send(AoI.PeerID, CE);
                receiverlist.Add(AoI.PeerID);
                this.ReceiverLists.Put(key,receiverlist);
            end
        end
    end
end
```

Algorithm 4: CEM receives a new AoI subscription from SPS

```
Input: continuous event (CE)
Data: continuous event state storage (CES), code repository (coderep),
      timer service (TS)
begin
    nextExec = CE.t_0;
    finalExec = CE.t_0 + (CE.n * CE.δt);
    while nextExec < GetMMVETime() and nextExec < finalExec do
        calculate AoE and E of CE at nextExec;
        f_apply = coderep.Retrieve(CE.f_apply Reference);
        f_apply(AoE, E);
        nextExec += CE.δt;
    end
    if nextExec < finalExec then
        store state of CE in CES;
        TS.Register(new Timer(nextExec, CE.EventID));
    end
end
```

Algorithm 5: CEE receives a new CE

stored and a timer is registered (see algorithm 5).

The algorithm, which is performed after a timer of the CEE triggers, works mostly analogous to algorithm 3. In addition, f_{apply} is retrieved and called.

A CE is terminated explicitly via the CEI based on the event ID: A termination message is sent to the CEM, which sends the message via the SPS to all peers included in the receiver list and drops the receiver list afterwards.

V. Simulations

We have implemented our approach in C# and performed simulations for a single zone. We used a peer number of 100, which corresponds to the maximum number of avatars, which can be handled by a Second Life region server. We chose a zone size of 256 x 256 m and used circular AoIs with a radius of 35 m, which also correspond to the values used by Second Life [**?**].

We performed three series of simulations on an IBM Blade Server including 2 Intel Xeon QuadCore CPUs with 2.33 GHz and 6 GB RAM. Each series consisted of 10 runs. In order to elimininate deviations, each run was repeated three times and the avarage values were used.

During a simulation run, on each peer we simulated 30 continuous actions each consisting of 10 single actions in an interval of 1 sec. The actions were propagated by using circular AoEs. In addition to the actions, we simulated user movement by using a random way-point model. In order to get comparable results, we pre-recorded action and movement data and reused the same data for all runs.

In each series, we varied key parameters to compare the performance of CEs and single events: The max length of CEs which were used to propagate the actions, the AoE size and the user speed.

To assess system performance, we recorded the number and size of messages which were sent between regular peers and the coordinator peer. We did not factor in the AoI subscription messages, because they are criteria for the performance of the SPS service and not CEs.

The presented results focus on the messages which were sent from the coordinator running the CEM to regular peers running CEEs. Obviously, when sending a CE instead of 10 single events the number of messages sent from peers to the coordinator is reduced constantly. Due to space reasons, we decided to leave these figures out. More specifically, we focus on message numbers. Concerning size: During all simulations runs, a CE message was about 5,2 percent larger then a single event message.

Variation of Continuous Event Length - In this series, we adjusted the number of aggregated actions per CE ranging from 1 to 10 and used fixed numbers of 10 m and 10 m/sec for AoE radius and user speed. CEs are able to reduce the message number significantly (minimum 44,76 % for a length of 2, maximum 80,36 % for a length of 10). Message reduction increases according to the CE length.

Variation of User Speed - Now we adjusted the user speed ranging from 1 to 10 m/sec and used fixed numbers of 10 m for AoE radius and a CE length of 10. Again, CEs are able to reduce the overall message number significantly (minimum 80,36 % for 10 m/sec, maximum 88,91 % for 1 m/sec). While the message reduction decreases for higher speed values, even for a speed of 10 m/sec the message number is still reduced strongly and all runs with CEs outperform the runs with single events consistently.

Variation of AoE Size - Finally we adjusted the AoE radius from 1 to 10 m and used fixed numbers of 10 m/sec for

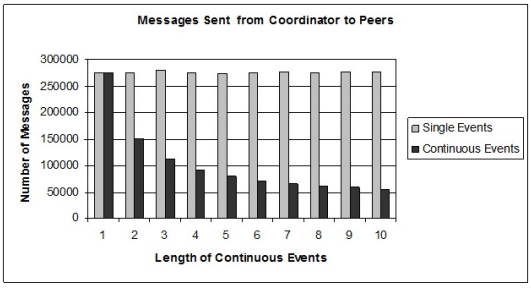

Fig. 1. Variation of Continuous Event Length

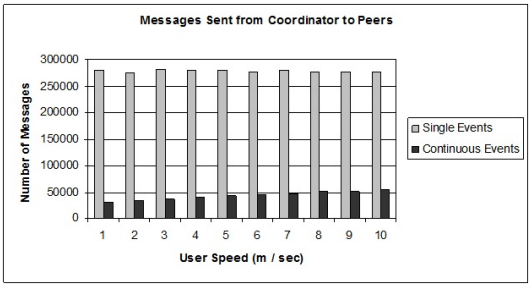

Fig. 2. Variation of User Speed

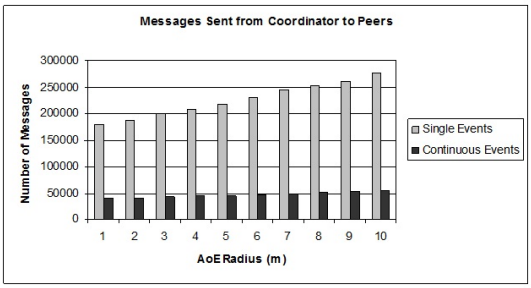

Fig. 3. Variation of AoE Size

user speed and a CE length of 10. The amount of messages increases for larger AoEs both when using single events and CEs. The increase for larger AoEs is much stronger when using single events, even CEs do need additional messages for management functionality. Again CEs outperform single events consistently in all runs.

In summary, CEs were able to reduce the message amount in all series significantly. The message amount increases when confronted with faster moving users and larger AoEs. But even in these series, single events were outperformed. Our simulations indicate a substantial reduction of update messages.

VI. Related Work

Dead reckoning (as described in section I) is often used by networked games to reduce the amount of network messages by extrapolating future user positions [**?**][**?**]. Dead reckoning is typically performed for positions of single objects. In contrast, our approach decouples the management of future states from the objects and is able to carry the information about future state changes of numerous objects by using a single CE.

[**?**] assigns the task of delaying and aggregating update messages within resonsibility zones in a P2P-based MMVE to superpeers. In contrast to our approach, it does not include an explicit event entity for continuous events and does not include computation functions.

[**?**] represents user avatars on remote nodes by objects which act bot-like according to given rule sets. The users only send guidance information from time to time. Our approach shows a stronger grade of abstraction by introducing artificial CE entities, which allow to apply state changes to numerous objects of different kinds over time.

Commercial MMVEs (e.g. [**?**][**?**]) often use temporal aggregation: The server delays the propagation of individual updates for a certain time period and then sends a single aggregated update messageTo the best of our knowledge, none of these systems include a technology similar to CEs.

VII. Conclusion

This paper presented an approach to reduce update messages in P2P-based MMVEs by introducing continuous events. We described our system design and algorithms to support continuous events. Our simulations show a significant potential to reduce update messages within a zone of our MMVE system.

Our next steps include the development of algorithms to extend our approach to multiple zones and coordinators and to handle disconnects. We further plan to add infinite continuous events and the possibility to change the functions of existing continuous events after creation to our approach.

References

[1] J. Smed, T. Kaukoranta, and H. Hakonen, "Aspects of networking in multiplayer computer games," *The Electronic Library*, vol. 20, pp. 87–97, 2002.

[2] J. Aronson, "Dead reckoning: Latency hiding for networked games," 1997. [Online]. Available: http://www.gamasutra.com/view/feature/3230/dead_reckoning_latency_hiding_for_.php

[3] The Peers@Play Project: http://pap.vs.uni-due.de.

[4] R. Süselbeck, G. Schiele, S. Seitz, and C. Becker, "Adaptive update propagation for low-latency massively multi-user virtual environments," in *Proceedings of the 18th International Conference on Computer Communications and Networks (ICCCN)*, 2009.

[5] F. Heger, G. Schiele, R. Sueselbeck, and C. Becker, "Towards an interest management scheme for peer-based virtual environments," in *Proceedings of the 1st International Workshop on Concepts of Massively Multiuser Virtual Environments (COMMVE)*, 2009.

[6] S.-Y. Hu, "Spatial publish subscribe," in *Proceedings of the 2nd International Workshop on Massively Multiuser Virtual Environments (MMVE) at IEEE Virtual Reality (IEEE VR)*, 2009.

[7] M. Varvello, F. Picconi, C. Diot, and E. Biersack, "Is there life in second life?" in *Proceedings of the 4th ACM International Conference on emerging Networking Experiments and Technologies (ACM CoNEXT)*, 2008.

[8] S.-Y. Hu, S.-C. Chang, and J.-R. Jiang, "Voronoi state management for peer-to-peer massively multiplayer online games," in *Proceedings of the 4th IEEE International Workshop on Networking Issues in Multimedia Entertainment (NIME)*, 2008.

[9] A. Bharambe, J. R. Douceur, J. R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang, "Donnybrook: Enabling large-scale, high-speed, peer-to-peer games," in *Proceedings of the ACM SIGCOMM*, 2008.

[10] World of Warcraft: http://www.worldofwarcraft.com.

[11] Second Life: http://www.secondlife.com.