Consistency Management for Interactive Peer-to-Peer-based Systems

Laura Itzel
University of Mannheim
Mannheim, Germany
laura.itzel@uni-mannheim.de

Verena Tuttlies
University of Mannheim
Mannheim, Germany
verena.tuttlies@uni-mannheim.de

Gregor Schiele
University of Mannheim
Mannheim, Germany
gregor.schiele@uni-mannheim.de

Christian Becker
University of Mannheim
Mannheim,Germany
christian.becker@uni-mannheim.de

ABSTRACT

Consistency is a crucial requirement for Massively Multiuser Virtual Environments (MMVEs). Such systems provide virtual worlds in which thousands of users can interact in real-time. To realize a consistent world view for all users, a flexible consistency management is required which balances the responsiveness and the level of consistency in the system. In this paper we present an approach for a consistency management for peer-to-peer-based MMVEs. The approach identifies users which actually interact with each other in the virtual world, groups them in *consistency sessions* and synchronizes all users in the session according to a synchronization protocol which is determined at runtime. We discuss the concept of consistency sessions, present algorithms to create and maintain sessions, and analyze what features are still missing and must be added in the future.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Algorithms, Design

Keywords

Consistency, Peer-to-Peer, Synchronization, MMVEs

1. INTRODUCTION

The popularity of Massively Multiuser Virtual Environments (MMVEs) has continuously increased in recent years. Applications like World of Warcraft [1] or Second Life [8] provide a virtual world where thousands of users can interact with each other in real-time. Such systems can realize a number of applications like multi-user online games or virtual meeting rooms. A crucial requirement of such systems is *consistency* [14]. Users which interact in the virtual world should perceive the same consistent world state. For example, if a user wants to transfer an object to another user, the system has to ensure that the object is passed exactly once. The object should neither be lost nor passed several times. This requires synchronization between the typically graphical clients that render the virtual world at the user's computer.

In general, consistency management has to balance performance, i.e., the responsiveness of the system, and the occurrence of inconsistencies. Strict synchronization protocols that realize a high level of consistency usually have a negative impact on the performance as potentially affected clients have to acknowledge every information about changes in the virtual world. In contrast, optimistic protocols can realize a high responsiveness by cutting down synchronization effort thus allowing inconsistencies to occur. This general problem is even more challenging in a peer-to-peer (P2P) architecture, where the execution of the virtual world is distributed among participating nodes and no central unit is used for coordination. Our work is part of the peers@play project [12], a cooperative project of the Universities of Mannheim, Duisburg-Essen and Hannover. The goal of this project is to develop a P2P-based middleware for highly scalable and interactive MMVEs.

In this paper we present an approach towards a flexible consistency management for P2P-based MMVEs. The approach allows to balance the responsiveness and consistency of the system. It is based on two observations: 1) Synchronization is only required for groups of entities which currently interact. For users that do not interact with others a strict synchronization is typically not required. 2) The consistency requirements for the synchronization of peers depend on the actions which are actually performed. Some actions need to be handled with a strict consistency like a money transfer while others may accept inconsistencies in order to provide a high responsiveness.

DISIO 2010 March 15, Torremolinos, Malaga, Spain.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8690
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8690

For consistency management, our approach introduces the concept of consistency sessions. The consistency session enables the synchronization of a group of interactive entities with a specific synchronization protocol. As changes in the virtual world are communicated via update events, this implies a synchronization of update events which are exchanged between the members of the session. In order to determine the particular synchronization protocol, we provide a classification of update events which maps event types to synchronization protocols. This enables the system to select a synchronization protocol from a set of existing protocols like [3] or [4] at runtime based on update events. Following this, we discuss how consistency sessions are built and managed at runtime.

The paper is structured as follows: In Section 2 we describe our system model. Requirements for consistency management are discussed in Section 3. We provide an overview of related work in Section 4 and present our consistency management infrastructure in Section 5. After presenting an example for our approach in Section 6, we close the paper with a conclusion and outlook on future work in Section 7.

2. SYSTEM MODEL

The model of our system is shown in Figure 1. The system consists of a number of computers called *peers*. These peers are connected via a common communication network like the Internet. Each peer has a clock which is synchronized with all other clocks in the system, e.g., by using the Network Time Protocol *NTP*. Furthermore, we assume that peers do not fail, i.e., do not unexpectedly log off the system.

Each peer is responsible for the computation of a single *entity* and a set of *entity copies*. An entity is the formal notion of an *avatar* which is a virtual representation of the user in the virtual world. An entity copy is a proxy for other entities in the system. If an avatar sees another avatar in the virtual world, its entity copy is added to the peer. Figure 1 shows this relationship. Peer P_1 is responsible for the computation of entity A which represents the avatar A in the virtual world. Furthermore, P_1 holds a copy of entity B which is managed on peer P_2. In the following we use the terms avatar and entity synonymously.

Actions in the virtual world lead to modifications of entities. These modifications are propagated to other peers via *update events*. Each update event has a single *sender* and a set of *receivers*. The sender is the entity which is responsible for the composition and distribution of the event. The set of receivers consists of those entities which are interested in the event. The determination of the interested entities is realized by our interest management [7]. The interest management is part of the middleware which exists on every peer. The middleware provides all required functionalities for a reliable communication of the peers [17]. In the peers@play project, this functionality is provided by the peers@play middleware.

3. REQUIREMENTS

In order to realize an interactive P2P-based virtual world a suitable consistency management infrastructure is required. The consistency management should be able to provide a consistent world state for users of the virtual world. However, a consistency management infrastructure for P2P-based MMVEs faces a number of requirements. In dependence

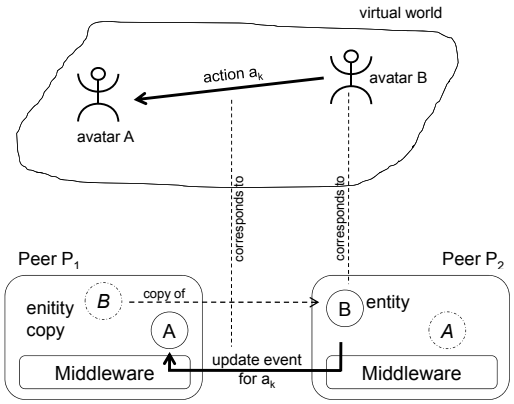

Figure 1: System model

on our previous work [15] we discuss the requirements in the following.

1) Scalability: The peers of interacting users in a virtual world ideally have to provide absolute consistency, i.e., the data held on each peer is synchronized with every other peer in the system. This realizes a consistent world state such that the experience of the virtual world is the same for all users. However, the maintenance of absolute consistency does not scale with an increasing number of users as all have to be synchronized. As we address highly interactive MMVEs the scalability of the system is an essential requirement.

2) Flexibility: In MMVEs users can perform a multitude of different actions, e.g., moving around the avatar or trading with other users. Each of these actions has different consistency requirements when being processed, e.g., the interval in which peers are synchronized. For virtual worlds a consistency management should maximize the responsiveness of the system for a high interactivity while minimizing the number of possible inconsistencies to provide a good experience of the virtual world for all users. Thus, a consistency management for P2P-based virtual worlds should be flexible in the sense that different synchronization protocols can be employed. The approach should be able to select and change synchronization protocols at runtime depending on the entities and the type of interaction in the virtual world.

3) Extensibility: Different MMVE applications may require different consistency models, i.e., the synchronization protocols that realize them. Furthermore, the synchronization protocol enforced for a specific situation may vary for different MMVEs. Therefore, the consistency management must be easily extensible for the developer of an MMVE. Developers should be able to integrate their own consistency model by adding further synchronization protocols.

4. RELATED WORK

In this section we discuss the related work in the field of consistency management infrastructures based on the requirements identified in Section 3. Maintaining a consistent state in a distributed system has been the subject of a lot of research in the past, mostly for databases and distributed simulations. Only a few approaches deal with a consistency management infrastructure for highly interactive MMVEs, most of them are based on a client/server architecture.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8690
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8690

The basis for all approaches for consistency management architectures are consistency models, i.e., the synchronization protocols to realize them. The models presented in the past can be divided into two classes, *conservative* and *optimistic* approaches. Conservative approaches process an event only if the system can guarantee that they will never receive another event that should have been processed before the actual event. For this purpose, conservative approaches delay the processing of the event. Common examples of conservative approaches are Lockstep synchronization [6] or the algorithm of Chandy, Misra and Bryant [3]. Conservative approaches provide a high level of consistency, but due to the delaying of events the system responsiveness is usually low. The state of the virtual world is the same on each peer, but the system waits until the slowest peer has confirmed that there exists no earlier event than the considered one. Conservative approaches are very well suited for actions that need to be highly synchronized like monetary transactions. In contrast, optimistic approaches process events immediately after receiving them. This possibly leads to inconsistencies which optimistic approaches try to correct through techniques such as rollbacks. A rollback is a mechanism to go back to a world state that is known to be consistent and then re-execute the events which produced the inconsistency. Common approaches in this field are Local-lag and Timewarp [10], Trailing State Synchronization [4] or protocols particularly developed for Multiplayer Online Games like [5]. With these approaches the system responsiveness is much higher than with conservative approaches. However, a user cannot be guaranteed that his instantly executed action will not be rolled back in the future. Hence, consistency models based on an optimistic synchronization approach are specifically well suited for highly interactive actions that can be reversed without negatively affecting the user. None of these isolated approaches are able to provide the necessary flexibility to support consistency requirements of a multitude of different actions that are experienced in an MMVE.

A number of different architectures has been presented to maintain consistency in MMVEs. As already mentioned most of them are based on a client/server architecture. Lu et al. [9] propose a consistency control mechanism that allows the application to select between three different levels of consistency, depending on, e.g., system load. The FIT-Gap framework [2] allows states in an MMVE to be associated with different replication models, depending on their consistency requirements. Both approaches are based on a client/server architecture.

In contrast to these centralized approaches, we propose a decentralized approach based on a peer-to-peer architecture. Pellegrino *et al.* [13] propose a hybrid architecture where a central arbiter is responsible for maintaining consistency. The central arbiter is listening to all messages that are sent, detects inconsistencies and resolves them. With this approach only optimistic event synchronization approaches with rollback techniques are applicable. McCaffery *et al.* [11] also provide the system with a managing unit for consistency, but not as a passive part of the system. They propose a spatial partitioning of the virtual world, and each part of the system resolves state consistency issues independently by the node that is responsible for the specific partition. Finally, HyperVerse [16] proposes another P2P-based architecture for maintaining consistency in a decentralized MMVE. The virtual world is divided into world partitions with different density (*weight*) of users. A multi-level consistency model, elastic consistency, is proposed with three different consistency levels. This model adapts the consistency level dependent on the actual partition weight. In the last two approaches the partitioning of the virtual world is not dynamic and it is not dependent on the actual consistency requirements of specific actions but dependent on spatial aspects. In conclusion, none of the existing approaches meet all three requirements which we discussed in Section 3. Consequently, we will present our approach that satisfies the three requirements in the following.

5. OUR APPROACH

In this section we present our approach to consistency management for interactive P2P-based systems. The central idea of the approach is to handle consistency for groups of entities which actually interact in the virtual world. An interaction is defined as an action of an avatar in the virtual world which concerns another avatar. Whether or not a certain action is an interaction depends on the specific application. For example, some applications may consider watching another avatar as an interaction while others may not. The interaction can be identified by the update events which are exchanged between the entities which represent the avatars. Consequently, we group the set of entities and their update events in a *consistency session*. As multiple groups with interacting entities will exist in the system, multiple consistency sessions will exist in parallel. In a consistency session, all entities and all exchanged update events are synchronized. The specific synchronization protocol is selected by our system at runtime. It depends on the actions which are actually performed as different actions may have different consistency requirements.

Figure 2 shows the assignment of avatars to consistency sessions. The avatars A_1 and A_2 have been combined in consistency session cs_1 as they interact with each other. Another consistency session (cs_2) has been established with a larger group of avatars – A_3 through A_7 – which also interact with each other. The interaction can be a direct interaction as between avatars A_5 and A_7 or can be an indirect interaction as between A_6 and A_7. Each session is synchronized with a synchronization protocol which is selected by the system. The protocols which are used for cs_1 and cs_2 can differ. For example, while cs_1 could employ the Lockstep synchronization [6], cs_2 could synchronize using Timewarp [10]. The figure also shows two avatars which are not part of any session. As they do not interact with any entities in a session they will receive update events but are not synchronized with other entities.

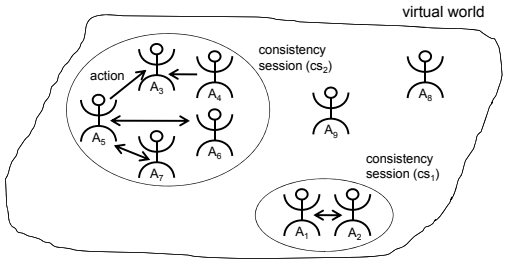

Figure 2: General concept of consistency sessions

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8690
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8690

In the following, we give a detailed introduction to consistency sessions in Section 5.1. The classification of update events is described in Section 5.2. Finally, we discuss the life cycle of consistency sessions and explain how update events are actually handled in our system in Section 5.3.

5.1 Consistency Sessions

The consistency for interacting entities is managed in a consistency session. A consistency session consists of a set of interacting entities and the update events they exchange, i.e., the actions they perform in the virtual world. Each session is managed by a single *session coordinator*. The session coordinator is a specific peer which hosts an entity in the consistency session and is responsible for the session. Its tasks are the life cycle management of the session and the administration of entities which join or leave the session. Furthermore, the session coordinator acts as a unit which actually enforces a selected synchronization protocol and synchronizes the entities respectively. For example, depending on the synchronization protocol it determines the order in which update events must be processed and ensures that update events are executed at each peer in this order. If node failures are likely to occur in the system, an efficient backup strategy for the session coordinator is required. As we assume that a peer does not fail without warnings this strategy is subject to future work.

The specific synchronization protocol which is employed for the session depends on the actions which are performed between entities in the session. Each action in the virtual world maps to a synchronization protocol with which the corresponding update event has to be handled. In order to realize this, the application developer must map each action to a certain synchronization protocol as we discuss in detail in Section 5.2. Furthermore, he must provide a classification for the available synchronization protocols in order to allow our system to compare the consistency level of different protocols. This classification should be done with respect to different factors like the resulting average responsiveness of the system by using a specific protocol and the risk of inconsistencies. In a consistency session the most restrictive consistency level with respect to those factors is applied which is required by any action, i.e., update event in the session.

Consistency sessions are dynamic structures. Entities can be added to an existing session if they interact with other entities which are already part of the session. Furthermore, entities can be removed from the session, e.g., if no interaction could be detected for a predefined time interval. A consistency session is terminated if no more interaction can be registered, i.e., all update events have been executed and no new events are registered in the session within a predefined timeout. Other group management issues are discussed in more detail in Section 5.3. Each entity can only be part of a single consistency session. If an entity is not a member of a consistency session, it will receive required update events, but is not synchronized with the other entities in the system. Furthermore, update events can belong to at most one consistency session. As we discuss in Section 5.2 some actions in the virtual world may not require a consistency management. Consequently, their update events do not belong to a consistency session, if none of the interacting entities are part of a consistency session. In case, an update event is sent between two different consistency sessions, these sessions are merged which we explain in detail in Section 5.3.

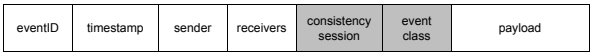

Figure 3: Update event

The concept of consistency sessions satisfies the first and the third requirement which we discussed in Section 3, namely scalability and extensibility. A consistency session is a dynamic structure that grows and shrinks with the number of interacting entities. Multiple consistency sessions may exist in the system in parallel. Due to these reasons and as consistency is managed by session peers themselves, our approach scales with an increasing number of entities in the system. In fact, special cases exist in which our approach needs to integrate more meta information about the session, e.g., the number of entities in the session. For example, for continuously growing sessions which do not shrink the use of a strict synchronization protocol does not seem reasonable even if required by the taken action. In this case meta information could be used in order to determine the trade-off between consistency and responsiveness. However, a detailed analysis of these cases and how they should be handled is subject to future work.

The third requirement – flexibility – is also satisfied by the consistency sessions concept. The consistency session uses the synchronization protocol with the highest consistency level that is required by actions performed within the session. As the protocol is action-dependent, the approach can flexibly adapt the consistency level for the session. Finally, the mapping of actions to synchronization protocols also satisfies the requirement of extensibility. The application developer can easily extend the mapping by new synchronization protocols as long as he provides a classification for all employed synchronization protocols. The selection of a specific protocol is realized by our system at runtime.

Having presented the general concept of consistency sessions we address update events in the subsequent section. We describe necessary extensions for the notion of update events and discuss their classification for our approach.

5.2 Event classification

An update event is the formal description of an action performed in the virtual world. It is sent from a sender to a set of receivers which are interested in the event. Figure 3 shows the components of an update event. The components which are emphasized have been added for consistency management. The original update event consists of five components, an *eventID*, a *timestamp*, a *sender*, a set of *receivers* and a *payload*. The *eventID* is a globally unique identifier which is generated by the middleware when the event is created. The *timestamp* denotes the time at which the event was created. Recall that our system model assumes all peers to have synchronized clocks. The timestamp is needed for maintaining consistency, i.e., message ordering. The sender and the receivers of the event are stored in the fields *sender* and *receivers*. The sender is represented as a tuple consisting of the entity id and the peer which hosts the entity. The *receivers* field holds a list of such tuples. The *payload* contains the actual instruction for processing the performed action.

For the purpose of consistency management we have extended the event notion by two more components, *session* and *class*. *Session* denotes the consistency session the en-

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8690
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8690

tity is part of. As we discussed, an event can only be part of a single session. Thus, the *session* field holds the id of the respective session and the id of the session coordinator which is responsible for the session.

The *class* field holds the synchronization protocol with which the update event has to be handled. The specific synchronization protocol depends on the action which is taken in the virtual world and which is modeled in the update event. The event class is automatically assigned by the middleware when the update event is generated. Thus, the middleware must know to which event class every defined action maps. We consider this as a task of the application developer as the specific actions and the decision with which synchronization protocol each action has to be handled are application specific. As an example, in most cases a fighting situation in a virtual world would be handled with a strict synchronization protocol such as Lockstep synchronization. Another example is a situation where two users are trading non valuable goods. In this case, many applications would probably use some weaker synchronization protocol such as Timewarp. Situations where an avatar just looks at another would probably not be synchronized at all.

Our system provides a list of event classes where each class is associated with the implementation of a specific synchronization protocol like [6], [10]. The list of protocols can be extended or changed by the application developer. The developer then maintains a mapping of event classes to synchronization protocols particularly for its own application. Each event class must be handled with one specific synchronization protocol, but a synchronization protocol can be applied for various event classes. Thus, the concept of event classification allows the application developer to flexibly enforce different synchronization protocols in different situations in his MMVE. Having presented the general concepts – consistency sessions and event classification – for our approach we discuss the life cycle management of consistency sessions and how update events are actually handled.

5.3 Update event handling

The handling of update events requires the creation of new consistency sessions and the modification of existing ones. The specific handling depends on the state of the sender and the states of the receivers in the system. In the following, we discuss the possible combinations of sender and receiver states in our system. We explain the creation of consistency sessions and discuss how an update event that has a sender and receivers in different consistency sessions leads to a merge of the concerned session.

In our approach, an entity can have three different states with respect to consistency management, namely *no session*, *session member* and *session coordinator*. The *no session* state denotes that the entity is not a member of a consistency session. The *session member* state indicates that the entity is part of a consistency session, but is not the session coordinator. The last state, in contrast, expresses that the entity is part of a session and is also the coordinator of the session. With respect to an update event which is sent from a single sender to a set of receivers nine state combinations can occur in the system. Figure 4 depicts three of the nine combinations for an entity which sends an update event and is not part of a consistency session. In the figure, the bounding box depicts a part of the virtual world with a number of interacting entities. The open grey boxes depict consistency

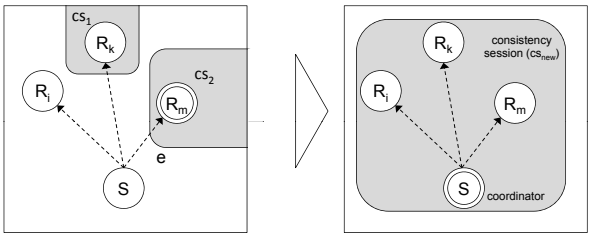

Figure 4: Sender state: no session

sessions which are likely to have more members.

The figure shows a sending entity S which sends an update event e (indicated by the dashed arrow) to three different receiving entities, R_i, R_k and R_m. An update event which is sent will arrive at a receiving entity which can be in one of the three – *no session* (R_i), *session member* (R_k) and *session coordinator* (R_m) – possible states. However, at the time the event is composed the sending entity is not aware of the state of a receiver. Thus, it creates a new consistency session cs_{new} and nominates itself as session coordinator. The new session is associated with the synchronization protocol that maps to the event class of e. After creating the session, S sets the session information in the update event and sends it to the receivers.

When R_i receives the event it detects that it is not part of a consistency session. Consequently, it adds itself to the consistency session by sending a registration event to the session coordinator which is specified in the received event. In addition to the entity id this registration event contains the state of the receiving entity R_i (*no state*). In the second scenario, e is received by the entity R_k which is already part of consistency session cs_1. In this case a session merge with cs_{new} is required as an update event can only belong to a single session. Thus, R_k notifies the coordinator of session cs_1 that a session merge is required and forwards the session information – session id, coordinator and synchronization protocol – which is included in the received event. In case the receiving entity is the session coordinator – which is the third scenario (R_m) in Figure 4 – the receiving entity can directly enter the session merge process.

The session merge basically comprises the selection of a session coordinator for the merged session, the choice of the synchronization protocol and the transfer of management data from the old coordinators to the new one. Furthermore, each session member is notified of the new coordinator and the new synchronization protocol. In our approach the sending entity S is always selected as the new session coordinator. This is due to the reason that the sender and all receivers can agree on a new session coordinator without further negotiation as each event has a single sender even if multiple sessions need to be merged. In the case of merging multiple sessions a registration event can arrive at an entity, that is no longer the session coordinator of the session due to the first merge. Thus, the receiver of the registration event has to forward the registration event to his actual session coordinator enabling the second session merge. Note, that the choice of a new session coordinator may not be optimal in the current system. If a session comprises a larger set of entities it may be reasonable to nominate the coordinator of the largest session as the new session coordinator. On one

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8690
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8690

hand this could avoid a large transfer of management information and may only require to integrate the sender in the session. On the other hand, this may cause some overhead in order to select the best possible coordinator. Furthermore, the amount of session coordinator changes should be minimal as a change always produces a high load, e.g., used bandwidth. However, the investigation of the trade-off and a selection strategy that results from the analysis is subject to future work.

The synchronization protocol used in the merged session is always the most restrictive protocol amongst all protocols which were used in the original sessions before. If a less restrictive protocol would be chosen, inconsistencies could occur that are not tolerable for some actions that were originally handled with a conservative consistency protocol. In addition to adding new members to a session, entities are removed if they have been inactive – have not sent any update events – for a specific amount of time. If a session consists of a single member or no more update events are sent within a given time interval, the session is closed by the session coordinator. If the session coordinator logs off the system, he nominates another session member as session coordinator.

Analogous to the removal of entities, the session coordinator can also decrease the consistency level. It can choose another synchronization protocol if all events that require a high level synchronization protocol have been processed. This proceeding avoids that all sessions are handled with the highest consistency level after some time. However, the development of efficient algorithms for group management and the flexible selection of synchronization protocols is subject to future work.

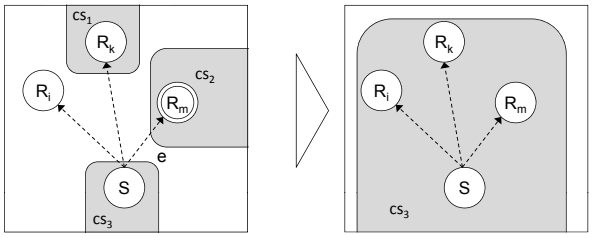

Figure 5: Sender state: session member / session coordinator

The possible combinations for the case when the sender is already part of a consistency session is shown in Figure 5. The sender is depicted as a normal session member. However, the handling of a sender which is the session coordinator does not differ a lot from the handling of a sender which is a normal session member. Thus, we discuss the cases together in the following.

Analogous to the first three combinations which we previously discussed, a sender can send an event to a receiver which does not belong to a consistency session (R_i), which is a normal member of a consistency session (R_k) or which is the coordinator of a consistency session (R_m). In the first case, the sender S sends an update event to R_i containing information about the session it is part of. As R_i is not part of a consistency session it adds itself to the session by sending a registration event with his state *no session* to the session coordinator of cs_3. In the second scenario, S sends the update event e to the receiver R_k which is already part of a consistency session. In this scenario a session merge is required as the update event is sent in between two different sessions. The receiver R_k notifies the session coordinator of cs_1 that a session merge must be realized forwarding information about session cs_3 which is contained in the update event. Then the session coordinator of cs_1 performs the session merge with the coordinator of cs_3. The process does not differ if S is the session coordinator of cs_3 as the receiver R_k will directly contact the session coordinator of cs_3. The new session coordinator will be the coordinator of the existing session cs_3 because this coordinator is known to all receivers as we previously discussed. The merged session will be synchronized with the synchronization protocol that provides the highest level of consistency among the protocols used in the merged sessions. In the last two possible scenarios, the sender S – in the role of a normal session member or a session coordinator – sends the update event e to the coordinator R_m of session cs_2. Again, a session merge is required for this case. R_m notifies the session coordinator of cs_3 that a session merge is required and the merge is performed.

Once entities belong to the same consistency session they can exchange messages without the overhead of creating new sessions or merging existing ones. Each update event which is exchanged in the session is also sent to the session coordinator. The session coordinator can now enforce the synchronization protocol which is assigned to the session. If it changes the synchronization protocol, it notifies the member of the session. Consequently, each entity that receives an event first determines which synchronization protocol has to be applied. It always acts on the higher level protocol, e.g., delays the execution of the event until the session coordinator informs it to do so. As every entity is aware of the employed synchronization protocol which is provided by the middleware the selected protocol can be realized.

```
Input: an update event (u)
Data: an entity sending or receiving u (e)
begin
    if e is sending entity then
        if e.state = no session then
            init new session s;
            s.sessioncoordinator = e;
        end
        set session and sessioncoordinator for u;
        send u to all its receivers;
    else if e is receiving entity then
        if e.state = no session then
            add e to session of u;
            send registration event to session coordinator of
            u;
        else
            send own session information to session
            coordinator of u;
        end
    end
end
```

Algorithm 1: Handling an update event

The algorithm that realizes the described handling of update events is given in Algorithm 1. The algorithm is executed on the hosting peer of an entity e that acts either as a sender or as a receiver of the update event u. If the entity wants to send an entity, the first part of the algorithm is executed. The algorithm checks whether e is in the state *no session*. If that is the case, a new session is created and

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8690
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8690

the sending entity e nominates itself as session coordinator. After the creation of the new session, the update event u is sent to all its receivers. If e is in state *session member* or *session coordinator*, no new session has to be created and u can be sent immediately. As already discussed, the session id and the corresponding session coordinator are sent to the receivers via an update event.

If the entity e is a receiver of the update event u, the algorithm also determines the state of e in a first step. In the case e is in state *no session*, e adds itself to the session of u. For this purpose it sends a registration event to the session coordinator of u's session. If e is in state *session member* or *session coordinator*, it has to notify the session coordinator of u of its state and that a session merge is required. Thus, it has to send a registration event containing its own session information and its own state – *session member* or *session coordinator* – to u's session coordinator.

```
Input: a registration event (r)
Data: a session (s) of an update event (u)
Data: the session coordinator of s (sc)
begin
    if r.state = no session then
        add sender of r to s;
    else if r.state = session coordinator then
        merge s and r.session;
    else if r.state = session member then
        send s.sessioninformation to session coordinator of u;
        merge s and r.session;
    end
end
```

Algorithm 2: Session management after an update event

The actual merge of the session is managed by the session coordinator u belongs to. As u may have multiple receivers, multiple sessions may have to be merged. Algorithm 2 describes how the coordinator of the session that contains u processes the registration events. Thus, when u's session coordinator receives a registration event, it first checks the contained state. If the state is *no session*, it adds the sender of the registration event to the session it manages. If the received state is *session coordinator* it performs a session merge for its own session and the received session. It nominates itself as the session coordinator of the new session as already discussed. If the contained state is *session member*, it sends the information about its own session to the session coordinator and notifies it about the session merge before performing the actual merge. Having presented the general concepts for consistency sessions, their management and update event handling at runtime we demonstrate our concepts based on an example in the following section.

6. EXAMPLE

In this section we give an example for our approach. We consider a virtual world with four possible activities, namely *trading of flowers*, *trading of fruit*, *transferring money* and *moving around*. In the example system, two synchronization protocols – Timewarp and Lockstep Synchronization – are available. Obviously we consider Lockstep synchronization as the higher level synchronization protocol since it pursues a conservative approach. In the example flowers and fruit are considered to be low prized goods with unlimited availability so that trading these goods has no high consistency requirements. Thus, *trading of flowers* and *trading of fruit* is mapped to Timewarp. In contrast, the transfer of real money is considered to be a critical action with high consistency requirements where no inconsistencies are tolerable. Thus, it is mapped to Lockstep synchronization. Finally, the event class for simple position updates, e.g. moving around a users avatar, is mapped to *no consistency*, so it is not necessary to synchronize such events. Hence, such update events do not trigger the creation or change of a consistency session. The event classes, associated actions and their mapping to synchronization protocols is shown in Figure 6.

event class	*action*	*synchronization protocol*
1	trading of flowers	Timewarp
2	trading of fruit	Timewarp
3	transferring money	Lockstep Synchronization
4	moving around	no synchronization

Figure 6: Example mapping

In the example we step through all stages of event handling. The example situation is shown in Figure 7. The left part of the figure illustrates the initial situation, the right part shows the situation after the entity E_d has received the update event e for processing a money transfer. In the initial situation, four users, respectively their avatars represented as entites, – E_a, E_b, E_c and E_d – are interacting. The entities E_c and E_d are trading some flowers (depicted as a dotted arrow denoted with *trade*). As already discussed, the trading action for flowers is mapped to Timewarp. As update events have been sent for the trading action between E_c and E_d both entities are in the same consistency session $cs1$ together with the respective update event. In this initial situation, E_c is the session coordinator of cs_1. Hence, E_c is in state *session coordinator* and E_d in *session member*.

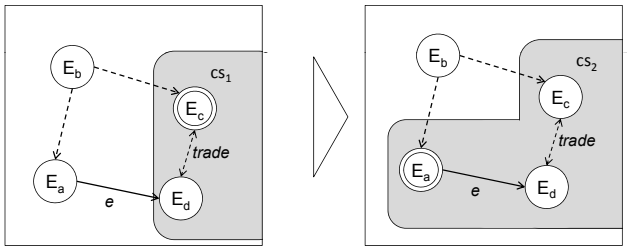

Figure 7: Example situation

The entity E_b is moving around in an area where the entities E_a and E_c can see it. The result is, that it sends update events about its position changes to those two entities (depicted as dashed arrows). Since moving around is mapped to no consistency, these position updates do not trigger creating or changing a consistency session. This means, that E_b is in no consistency session and therefore is in the state *no session*. The state of E_c does not change by the position updates sent by E_b, so E_c remains in state *session coordinator*. Also, the state of E_a remains the same (*no session*).

Now we look at event e, that is an update event for a money transfer from E_a to E_d. As already discussed, we consider a money transfer to be a critical action that is mapped to the conservative synchronization approach Lockstep synchronization, as inconsistencies or rollbacks cannot be tolerated for this action. First, the sending entity E_a executes Algorithm 1. As E_a is in state *no session*, it creates a

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8690
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8690

new session cs_2 with synchronization protocol Lockstep synchronization and nominates itself as the session coordinator of this session. After creating the new session, it sends the update event e for the money trading action to its receivers. In our example, the event only has one receiver E_d.

In the next step, the update event e is received by E_d, so E_d executes Algorithm 1. The received event contains the session id of the newly created session $cs2$ and its session coordinator E_a. First, the algorithm checks, in which state E_d is. In our example, E_d is in state *session member*, because it is part of the consistency session $cs1$. As sender (E_a) and receiver (E_d) of the update event are in different sessions, a session merge is necessary. Thus, E_d sends a registration event to the session coordinator of cs_2 given in e. This registration event contains the session information - session id, session coordinator and synchronization protocol - of his own session cs_1 and his own state (*session member*).

The session coordinator of cs_2 is E_a. The hosting peer of this entity is responsible for the merge of the two sessions. To accomplish that, Algorithm 2 is executed by E_a as the session coordinator of session cs_2. This algorithm checks the state given in the registration event. In our example, this state is *session member*. Thus, E_a sends a notification that both sessions will be merged to the session coordinator of the corresponding session (E_c). The synchronization protocol used for the merged session cs_2 is Lockstep synchronization as this is the highest level protocol used in the original sessions before merging. Now, all update events and entities within the consistency session $cs2$ can be synchronized and executed, managed by the new session coordinator E_a.

7. CONCLUSION AND FUTURE WORK

In this paper we proposed a flexible consistency management which is based on consistency sessions. A consistency session is built from a number of entities which interact in the virtual world. The members of the session are synchronized using a synchronization protocol which is determined by our system at runtime. Currently, we are implementing the discussed concepts with a number of existing synchronization protocols. Furthermore, we are working on group management algorithms in order to avoid sessions which only increase and do not shrink. In future work we plan to develop efficient algorithms for the selection of the session coordinator with respect to resource aspects such as bandwidth and actual system load.

8. REFERENCES

[1] Blizzard Entertainment. http://www.worldofwarcraft.com.

[2] A.-G. Bosser. A framework to help designing innovative massively multiplayer online games interactions. In *Technologies for E-Learning and Digital Entertainment*, pages 519–528. Springer Berlin/Heidelberg, 2006.

[3] K. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, Sept. 1979.

[4] E. Cronin, B. Filstrup, A. R. Kurc, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. In *1st workshop on Network and system support for games (NetGames 02)*, pages 67–73, New York, NY, USA, 2002. ACM.

[5] S. Ferretti. A synchronization protocol for supporting peer-to-peer multiplayer online games in overlay networks. In *Second international conference on Distributed event-based systems (DEBS '08)*, pages 83–94, New York, NY, USA, 2008. ACM.

[6] T. A. Funkhouser. Ring: a client-server system for multi-user virtual environments. In *1995 symposium on Interactive 3D graphics (SI3D 95)*, pages 85–ff., New York, NY, USA, 1995. ACM.

[7] F. Heger, G. Schiele, R. Süselbeck, and C. Becker. Towards an interest management scheme for peer-based virtual environments. In *1st International Workshop on Concepts of Massively Multiuser Virtual Environments (COMMVE 09)*, March 2009.

[8] Linden Lab. http://www.secondlife.com/.

[9] T.-C. LU, M.-T. LIN, and C. LEE. Control mechanism for large-scale virtual environments. *Journal of Visual Languages & Computing*, 10:69–85, 1999.

[10] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-lag and timewarp: providing consistency for replicated continuous applications. *IEEE Transactions on Multimedia*, 6(1):47–57, Feb. 2004.

[11] D. J. McCaffery and J. Finney. The need for real time consistency management in p2p mobile gaming environments. In *2004 ACM SIGCHI International Conference on Advances in computer entertainment technology (ACE 04)*, pages 203–211, New York, NY, USA, 2004. ACM.

[12] Peers@Play Project. http://www.peers-at-play.org.

[13] J. D. Pellegrino and C. Dovrolis. Bandwidth requirement and state consistency in three multiplayer game architectures. In *2nd workshop on Network and system support for games (NetGames 03)*, pages 52–59, New York, NY, USA, 2003. ACM.

[14] G. Schiele, R. Sueselbeck, A. Wacker, J. Haehner, C. Becker, and T. Weis. Requirements of peer-to-peer-based massively multiplayer online games. In *Seventh International Workshop on Global and Peer-to-Peer Computing*, 2007.

[15] G. Schiele, R. Sueselbeck, A. Wacker, T. Triebel, and C. Becker. Consistency management for peer-to-peer-based massively multiuser virtual environments. In *1st International Workshop on Massively Multiuser Virtual Environments at the IEEE Virtual Reality (MMVE 08)*, Reno, Nevada, USA, 2008.

[16] H. Schloss, J. Botev, M. Esch, A. Höhfeld, I. Scholtes, and P. Sturm. Elastic consistency in decentralized distributed virtual environments. In *4th International Conference on Automated Solutions for Cross Media Content and Multi-channel Distribution (AXMEDIS 08)*, Florence, Italy, 2008.

[17] R. Sueselbeck, G. Schiele, S. Seitz, and C. Becker. Adaptive update propagation for low-latency massively multi-user virtual environments. In *18th Internatonal Conference on Computer Communications and Networks, 2009 (ICCCN 09)*, pages 1–6, Aug. 2009.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8690
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8690