Improving the Concurrent Updates of Replicated Global Objects in Multi-Server Virtual Environments

Sebastian Bartholomäus, Alexander Ploss, and Sergei Gorlatch
University of Münster, Germany
bartholomaeus@imfl.de,<a.ploss,sgorlatch>@uni-muenster.de

ABSTRACT

We study an emerging class of high-performance virtual environments, called *Real-Time Online Interactive Applications* (ROIA), with such popular examples as Massively Multi-player Online Games (MMOG), interactive simulations, virtual communities, etc. ROIA must handle an enormous number of actions from geographically distributed user processes and present to each user a consistent view of the application state. We address a challenging aspect which influences the application scalability – the simultaneous access of several clients and servers to globally replicated objects. The traditional primary-copy-based state replication mechanism ensures fast read access but shows drawbacks for global objects that are concurrently updated by multiple processes. Our main contribution is to combine the traditional state-based approach with a novel operation-based *update ordering* approach that considers the operation's semantics and allows to use weaker consistency models than the sequential consistency enforced by the primary copy protocol. We implement our improvement of the update concurrency as the *Global Object Management System* (GOMS) and integrate it into the *Real-Time Framework* (RTF) – our platform for the high-level development and efficient execution of ROIA which has been used for implementing numerous applications like real-time online MMOG, crowd simulation, etc. Our experiments with GOMS show significant improvements regarding both the usability and the performance of the concurrent updates to replicated global objects.

Categories and Subject Descriptors

C.2.4 [**COMPUTER-COMMUNICATION NETWORKS**]: Distributed Systems —*Distributed applications*; D.1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming—*Distributed programming*; C.4 [**PERFORMANCE OF SYSTEMS**]: Performance attributes

DISIO 2011, March 21, Barcelona, Spain

DOI 10.4108/icst.simutools.2011.245593

General Terms

Algorithms, Performance, Design, Synchronization

Keywords

Global Objects; Data consistency; Concurrent write; Update ordering; Massively Multiplayer Online Games (MMOG); Real-Time Framework (RTF); Virtual Environments

1. INTRODUCTION

This paper studies an emerging class of high-performance virtual environments, called *Real-Time Online Interactive Applications* (ROIA), with such popular examples as Massively Multi-player Online Games (MMOG), interactive simulations and virtual communities. ROIA must handle an enormous number of actions from geographically distributed clients and present to each user a consistent view of the application state. To be attractive for users, ROIA have to provide Quality of Service (QoS), including high interaction rates and low response times, which requires scalability over multiple resources (servers) that process the application state in a distributed manner.

We address the probably most challenging aspect which influences the application scalability – the simultaneous access of several clients and servers to replicated objects. The used model of consistency between the replicas of a global object determines the degree of concurrency in the accesses to this object while guaranteeing the correctness with respect to the application logic. The degree of concurrency plays a critical role for the QoS of online games because it ultimately determines the degree of immersion which in turn is an important factor for attracting customers.

We start by analysing the traditional *primary copy protocol* for updating global objects and show its drawbacks both in performance and usability for the area of distributed virtual environments. We consider an alternative *update ordering* approach, originally suggested in [1], which enables performing many updates concurrently; it utilizes the semantics of particular client operations to reduce unnecessary delays.

We implement our improvements of the update concurrency as the *Global Object Management System* (GOMS). We integrate GOMS into the *Real-Time Framework* (RTF) [3] – our platform for the high-level development and efficient execution of ROIA which has been used for implementing numerous applications including real-time online MMOG, crowd simulation, etc. Our experiments show significant improvements regarding, both, the usability and the application performance.

2. ROIA DEVELOPMENT USING RTF

We consider the arguably most challenging example of ROIA – Massively Multiplayer Online Games (MMOGs) which support potentially thousands of users playing together simultaneously. MMOGs simulate large virtual worlds and must provide for the users a fast-paced game experience and high interaction rates. To allow for high interactivity and timely updates of the application, ROIA use the *real-time loop* as their operation model [3]. Fig. 1 illustrates the real-time loop of an MMOG server which consists of three major steps: The server receives user actions from the clients (1. in the figure), computes a new application state according to the application logic (2.), and sends the new state to all clients and other servers (3.).

Each ROIA server process has to compute a new application state in a very small time frame to allow for (soft) real-time interactions between the users. A common approach for ROIA is to organize processing in so-called time *ticks*. At the beginning of each tick, a new iteration of the real-time loop is started. If a loop iteration is finished before the end of the tick, the start of the next iteration is delayed until the start of the next tick. Game servers usually have the rates of 4 to 50 ticks per second, resulting in ticklengths between 20 ms and 250 ms. If some consecutive ticks take more time than the defined ticklength, the application becomes "choppy" and less responsive, thus substantially disturbing the user's experience and immersion.

The Real-Time Framework (RTF) is used to hide the low-level details (state distribution between servers, communication and serialization, etc.) from the application developer. Modern MMOG feature vast landscapes inhabited by thousands of virtual citizens (avatars). It is impossible for a single server to provide the computational power and bandwidth required to simulate a coherent game world of this size. Therefore, RTF natively supports practically proven parallelization strategies for MMOG development: zoning, zone-replication, and instancing. Parallelization strategies distribute computation load on multiple servers while keeping for the user the illusion of a coherent virtual world: although players are connected to different servers, they still have the impression of playing in one common world.

Zoning divides the game world into zones, mostly cuboid areas. Fig. 2 illustrates four zones, connected by several portals. All entities located within one zone are processed by the server responsible for this zone. Portals allow entities to travel from one zone to the other. If an avatar (an entity controlled by a client) moves through a portal from one zone

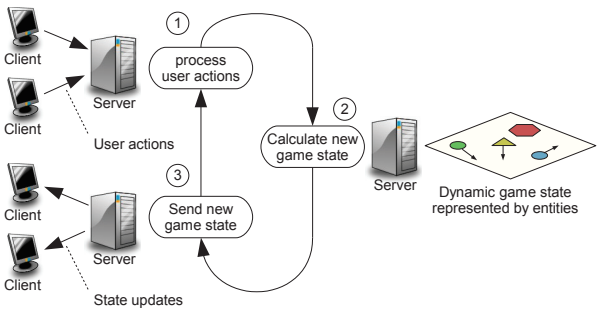

Figure 1: Real-time loop in an online game

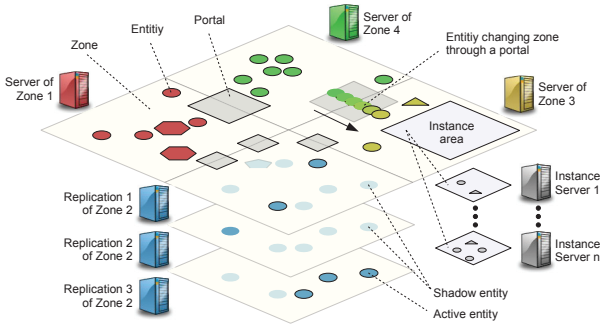

Figure 2: The RTF parallelization strategies: zoning, zone-replication and instancing

to another, RTF transparently migrates the client's connection to the other server: the user remains unaware of being transferred to another server when changing zones.

Zone-replication distributes processing of the same zone among multiple servers: in Fig. 2 three servers cooperatively process zone 2. Each server is responsible for some part of the overall data and replicates that part to the other servers [3]. Entities for which a server is responsible are called *active entities* on this server; other entities are called *shadow entities* on this server.

Instancing creates multiple copies of special parts of the game world. Fig. 2 shows how a small instance area in zone 4 is processed in separate copies by different servers [3].

3. GLOBAL OBJECTS AND CONSISTENCY

As many contemporary game engines, RTF supports the *sequential consistency* model for multiple server processes [2]. This is achieved by the *Primary Copy Protocol* (PCP) which defines one copy of an object (entity) to be the *primary copy*. Only the process that owns the primary copy (*primary process*) can change its state; the entity is called *active* for this process. When using the zoning and instancing parallelization approaches, in most cases only the primary copy of an entity exists. But if an entity is in a replicated zone or a portal, then other server processes may hold a shadow copy of the entity which they are not allowed to update directly.

Fig. 3 illustrates the replica update paths when using PCP. If an entity is updated, the primary process sends the update to all other server processes that own a shadow copy of the entity. In turn, these processes inform all their clients that are interested in the entity. The bottom line is that PCP prohibits concurrent updates entirely since the primary copy acts as a sequencer for all update operations: if another process wants to update the entity, it has to forward the operation to the process that owns the primary copy. Consequently, all server processes "see" the same order of state updates for the object, which is the main idea of sequential consistency.

As client processes (avatars) can travel between zones and thus may be connected to servers holding different copies of the state, sequential consistency cannot be guaranteed for these processes. Due to the network effects, a migration can connect a client to a server containing object replicas in an "older" state than the one the client has already seen. This violates sequential consistency from the client's perspective.

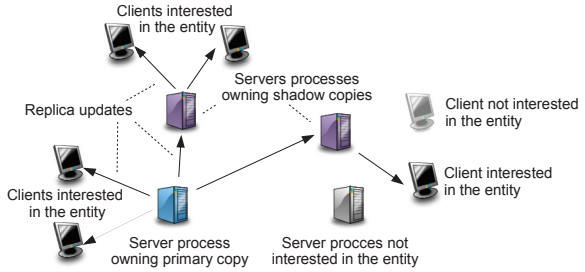

Figure 3: Replica update paths using PCP

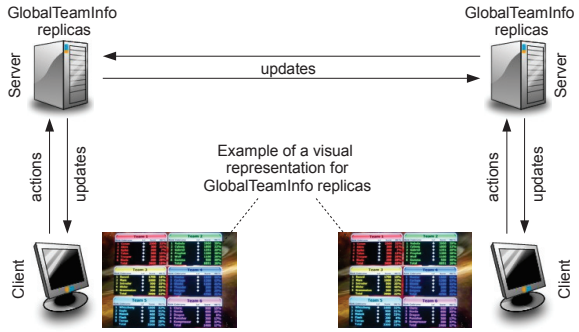

Figure 4: Replicated GlobalTeamInfo objects

The primary copy protocol is still suitable if only few processes modify an object and the majority just read it. In this case, each object can be allocated to a single server process that performs all updates on the object. In RTF's combination of parallelization mechanisms, this holds for all entities located in a particular zone: only the server assigned to the zone needs access to the objects in that zone. However, in many applications there exist *global objects* which are updated by many or even all server processes.

Example.

An example of global objects in online computer games are *teams*, i.e. groups of players that cooperatively play the game and compete for points. Each player sees a list of all teams, their current members and the teams' points. Therefore, the game application has to provide for each team a "global team information object" (`GlobalTeamInfo`). All server processes update the `GlobalTeamInfo` objects because players can join, leave or score points for their team. Fig. 4 depicts a scenario where two `GlobalTeamInfo` replicas are located on server processes. Clients send actions to their servers which in turn update the replicas and distribute updates on them.

RTF offers global objects (also called *globals*): in contrast to an ordinary entity, a global is always replicated to *all* participating server processes. The process that created a global, owns the primary copy and is the only process that can update this global. All other processes own shadow copies and can read the global's state, but are not allowed to update it. RTF's current approach to overcome this limitation is to send so-called *forward messages* to the primary process. Whenever a process owning a shadow copy of a global wants to update the global's state, it sends a forward message containing information about the action to be performed to the primary process (see Fig. 5). The primary process receives the forward message and updates the global, whose new state is then replicated to all other processes using the RTF's default replication mechanism. Like in the PCP, the process owning the primary copy acts as a sequencer for all actions that require update on the global and thus ensures sequential consistency.

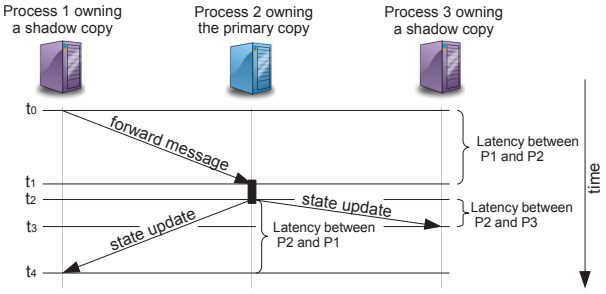

Figure 5: Forward message is sent to the primary copy; state update is distributed to other processes

Drawbacks of the primary copy approach.

The PCP is efficient for entities which are only updated by a single process or a small group of processes(the latter, e.g., when in a replicated zone or travelling through a portal). However, global objects have to be updated by all server processes, and all but one of them have a shadow copy of a global object, i.e., they have to send forward messages to update the global. This enforces sequential consistency, but causes a delay of approximately double latency between the process that updates the global and the process that holds the primary copy, until the final state is seen by all server processes (see Fig. 5): at time t_0, process P_1 forwards an update operation to the primary process P_2. The result of that update operation is seen at t_3 in P_3, and at t_4 in P_1.

Another drawback of the primary copy approach is that application developers are burdened with designing forward messages and corresponding handling code for every operation on global objects. Additionally, they have to check if the current process is the primary process for every update on a global object. This can become very time consuming.

Towards a new approach.

Basically, there are two approaches to updating copies of a global object in a distributed system: using either the *state* of the object's attributes or some *operations*. In a *state-based* system, an update for a replica contains the new states of some or all of its attributes. Once a primary copy is updated, its new state is propagated to all shadow copies that eventually see the new state. In an *operation-based* system, an update contains the serialized form of an operation and its parameters. This operation is performed on all replicas. In contrast to a state-based system, an operation-based system allows to exploit the operations' semantics to relax the consistency constraints.

Fig. 6 illustrates the classical state-based approach by means of a simplified version of the `GlobalTeamInfo` object which only contains the team points, and provides the `increasePoints` and `decreasePoints` operations. Process 2 (P_2)

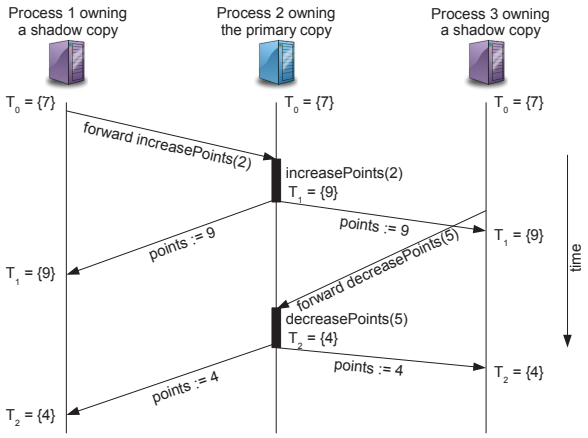

Figure 6: Increase and decrease operations on a GlobalTeamInfo object in the state-based solution.

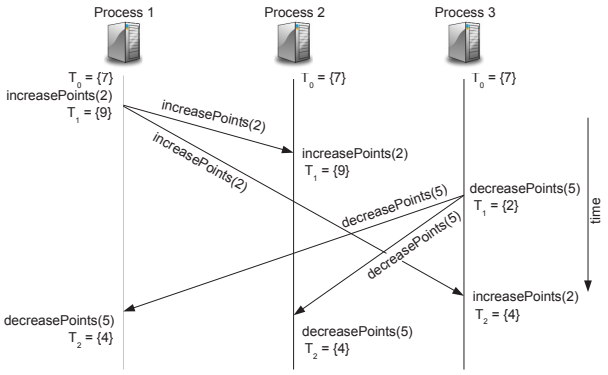

Figure 7: Increase and decrease operations on a `GlobalTeamInfo` object in operation-based approach

owns the primary copy of a `GlobalTeamInfo` object. The team default state T_0 is $\{7\}$, which means it has seven points. A player's action on process P_1 increases the points for the team by 2, while another player's action on P_3 decreases them by 5. Both processes own a shadow copy of the global object. Therefore, forward messages for the increase and the decrease operations have to be sent to P_2. P_2 executes the forwarded operations in the order they arrive, and propagates the new values to P_1 and P_3 using RTF's state-based replication mechanism.

This example raises the question, if the strict model of sequential consistency is required for this kind of object with this kind of operations? The answer is no, because it does not matter in which order the increase and decrease operations are performed. As long as each operation is performed on every process, the `GlobalTeamInfo` replicas will be eventually consistent. This is completely acceptable for such kind of information in an online game.

The knowledge about the commutativity of the increase and decrease operations obviates the need for sequencing the modifications. Instead, processes can modify the global concurrently as illustrated in Fig. 7. Therefore, we can switch to the operation-based approach: each process directly calls the team's increase or decrease operation to change the local state and broadcasts the operation and its parameter to all other processes. The final state of all replicas is consistent, as soon as all processes have executed all update operations. This solution performs better than the state-based approach in terms of performance and fault-tolerance: the process initiating the update can immediately execute the operation. For all other processes, the time until the update arrives is equal to the latency between the sender and the receiver, instead of the double latency as in the state-based case. Additionally, there is no primary copy as a single point of failure.

We use a non-blocking update protocol: update operations return immediately, no matter whether the operations can be applied directly or some synchronization has to be done. The consequence of this approach is that for some update operations on global objects, the processes invoking them will not see the result of their own update operation immediately, which is acceptable: the new approach does implicitly what is done explicitly by ROIA developers in the traditional approach (sending forward messages and waiting for the state update from the primary copy).

The operation-based approach requires custom messages to broadcast operations and their parameters. Ideally, ROIA developers should not have to implement anything except the global object. Therefore, serialization, deserialization and execution of custom operations have to be hidden from the developer to improve the usability of global objects.

4. UPDATE ORDERING AND GOMS

To implement an efficient operation-based replication, we use operation semantics to choose between different ordering constraints for the execution of update operations, following the ideas of [1]. While consistency models are defined once for all operations on the whole distributed memory, ordering constraints are defined for sets of operations, based on their pairwise relations. Therefore, various ordering constraints can be used within the same distributed system.

A replica group is a group of processes which all hold a replica of the same object. Assuming a n-replica group, $R^n = \{R_1, ..., R_n\}$, and an operation set OP_{set}, we follow [1] in defining the following ordering constraints.

The FIFO ordering constraint "$\rightarrow$": *If two updates u_1 and u_2 originate from the same replica R_i and are sent to the replica group, and if u_1 is executed before u_2 at original replica, then $u_1 \rightarrow u_2$, iff u_1 is executed before u_2 at all other replicas in the group.*

There is no constraint regarding the execution of update operations sent from different processes.

The Causal ordering constraint "$\prec$": *If R_i executed update u_1 originated from R_j before sending out update u_2, then $u_1 \prec u_2$, iff u_1 is executed before u_2 at all replicas.*

The causal ordering constraint "$\prec$" is used for two consecutive causally related operations. Causal ordering extends FIFO ordering by taking multiple senders into account. Updates from the same process are considered potentially causally related and are, therefore, executed at all processes in the same order (FIFO). For updates from different processes, a potential causal relation is defined by the happened-before relation which assumes a causal relation between each pair of consecutive operations on an object

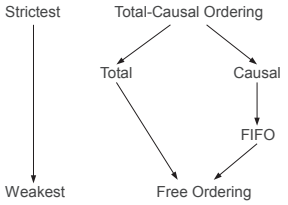

Figure 8: Relations between ordering constraints

regardless of their semantics.

The Total ordering constraint "$\leftrightarrow$": *For two updates u_1 and u_2 sent from R_i and R_j, it holds $u_1 \leftrightarrow u_2$ iff when one replica executes u_1 before u_2 then the other replicas execute u_1 before u_2 as well; vice versa: when one replica executes u_2 before u_1 then the other replicas execute u_2 before u_1.*

In general, total ordering does not imply FIFO ordering. The ordering of updates can be arbitrary, even for updates from the same process, as long as all processes see the same order. However, for our approach we assume that total ordering also ensures FIFO ordering.

The Total-Causal ordering constraint "$\Rightarrow$": *If two updates u_1 and u_2 originate from R_i and R_j, respectively, then $u_1 \Rightarrow u_2$ iff $u_1 \prec u_2$ and $u_1 \leftrightarrow u_2$.*

The combined *total-causal* ordering constraint ensures that total- or causal-ordering is performed depending on the semantics of each pair of updates.

As Fig. 8 shows, ordering constraints differ regarding the strictness of the consistency model they ensure. Total and causal ordering are orthogonal concepts, so combining them results in an even stricter ordering constraint.

Operation relations: Idea and example

The basic idea of update ordering is to choose the weakest possible ordering constraint for every pair of operations on a replicated object. To determine the weakest possible ordering constraint, the *operation relations* (i.e., the inter-operation semantics) have to be analysed. For two operations op_1 and op_2, we analyse the relation between any two updates, u_1 from op_1 and u_2 from op_2.

The Commutative relation "$\|$": *Two updates u_1 and u_2 are called commutative ($u_1 \| u_2$), iff the effect of executing (u_1, u_2) equals the effect of executing (u_2, u_1).*

The commutative relation is obviously symmetric.

The Conflicting relation "$\bowtie$": *Two updates u_1 and u_2 ($u_1 \neq u_2$) are called conflicting ($u_1 \bowtie u_2$), iff the effect of executing (u_1, u_2) is always different from that of executing (u_2, u_1).*

The conflicting relation is also symmetric.

The Caused-by relation "$\preccurlyeq$": *An update u_2 is caused-by another update u_1 ($u_1 \preccurlyeq u_2$), iff $u_1 \prec u_2$, and u_2 can be affected by u_1.*

A causal relation is traditionally defined by the simple happened-before relation. But the happened-before relation causes some operations to be treated as causally dependent, although they are unrelated from the semantical point of view. The caused-by relation excludes such operations which under no condition have an effect on the following operations. The caused-by relation is asymmetric.

Example: Caused-by relation.

In the case of teams, the `join(player)` and `leave(player)` operations on the `GlobalTeamInfo` object are completely unrelated to setting the team's color, as well as increasing and decreasing its score. The caused-by relation reflects this fact, in contrast to the happened-before relation which assumes that all these operations are potentially causal dependent. To illustrate the caused-by relation, this example focuses on the semantics of the `join(player)` and `leave(player)` operations. Only the player himself decides if he joins or leaves a team; a player can only join a team if he is not already member of a team; a player can only leave a team he is currently member of. All operations which do not obey these rules are invalid: the application logic checks these properties before a join or leave operation is called.

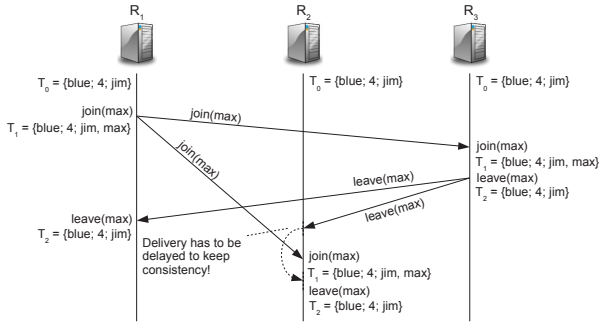

Figure 9: Join and leave operations for the same player are in the caused-by relation.

The execution of a join operation at a replica can influence the result of the subsequent execution of a leave operation *for the same player* as illustrated by Fig. 9. If R_2 receives the leave(max) operation from R_3 before the join(max) operation from R_1, the leave operation is based on, then R_2 has to delay the execution of the leave operation to remain consistent. Join and leave operations invoked concurrently on two replicas can be executed concurrently, because they cannot be issued by the same player, due to the assumption that client migration does not incur an overtake of the operations issued at the original replica. It follows that `join(player)` $\preccurlyeq$ `leave(player)` and `leave(player)` $\preccurlyeq$ `join(player)`.

Determining Ordering Constraints.

For a given set of operations OP_{set} on a global, we define four subsets of OP_{set}, regarding the ordering constraints:

- $OP_{Commutative}$ contains all operations of OP_{set} that are commutative with all other operations in OP_{set}. These operations can be sent and executed using the FIFO-ordering constraint.
- OP_{Causal} contains all operations of OP_{set} that are in a caused-by relation with at least one operation in OP_{set}. These operations can be sent and executed using the Causal-ordering constraint.
- OP_{Total} contains all operations of OP_{set} that are in a conflicting relation with at least one operation in OP_{set}. We call these operations total: they can be sent and executed using the Total-ordering constraint.
- $OP_{Total-Causal} = OP_{Total} \cap OP_{Causal}$. Operations in $OP_{Total-Causal}$ can be sent and executed using the Total-Causal-ordering constraint.

Due to the definitions of $OP_{Commutative}$ and OP_{Total}, the following assertions hold: $OP_{Commutative} \cup OP_{Total} = OP_{set}$ and $OP_{Commutative} \cap OP_{Total} = \varnothing$ (see [1]). A total operation is not necessarily in the conflicting relation with each operation in OP_{Total}. Commutative pairs of total operations can be executed in arbitrary order.

Example revisited: operation sets.

The example based on the `GlobalTeamInfo` object consists of the following set of operations:
OP_{set} = { `join(player)`, `leave(player)`, `setColor(color)`, `increasePoints(amount)`, `decreasePoints(amount)` }.

The inter-operation relations are as follows: 1) `increasePoints` and `decreasePoints` are commutative to all operations in OP_{set}; 2) `setColor` is conflicting with itself, but commutative to all other operations in OP_{set}; 3) `join` is in a caused-by relation to `leave` and vice versa; 4) `join` and `leave` are commutative to all operations in OP_{set}.

The last statement is true because all *valid* pairs of leave and join operations are commutative to each other. Valid pairs are operation pairs which can occur based on the application's semantics of the join and leave operations.

The operation sets for the example are:

$OP_{Commutative}$ = { `join(player)`, `leave(player)`, `increasePoints(amount)`, `decreasePoints(amount)` }
OP_{Total} = { `setColor(color)` }
OP_{Causal} = { `join(player)`, `leave(player)` }
$OP_{Total-Causal}$ = $\emptyset$

Global Object Management System (GOMS).

The Global Object Management System (GOMS) implements the described update ordering approach. We will refer to the global objects using the update ordering approach as *managed* globals, and to RTF's usual global objects using the state-based approach as *default* globals. With GOMS, we introduce the class `Global`, a new abstract base class for all managed globals.

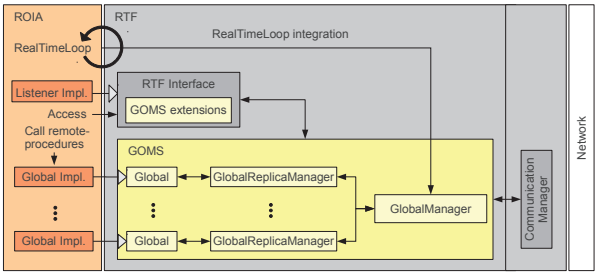

Figure 10: Integration of GOMS into RTF

Fig. 10 depicts a high-level component diagram of GOMS and its integration in RTF. In order to use a managed global in GOMS, the application developer creates a new class derived from `Global`, implements the update operations in special functions (so-called remote functions) and defines their pairwise relations using the constraints described above. GOMS facilitates the implementation of remote functions by providing a set of macros which generate code for the transmission and execution of updates at remote servers. The managed globals are registered in GOMS by using the GOMS extensions of the RTF API. The GOMS, in turn, distributes local invocations of remote functions to all replicas, considering the required ordering strategy for the specific operation. To realize the Total-ordering constraint, *unique sequence numbers* (USNs) are generated for these updates at a dedicated process in the system. The USN orders all updates of particular operation

GOMS is implemented in C++ which is the language of choice in many ROIA including online games. Its intersection with RTF is reduced to a minimum, which allows to use it side by side with the RTF's default Globals. GOMS uses RTF for communication between processes.

5. EXPERIMENTAL EVALUATION

For evaluation of GOMS, we use an online game prototype called RTFDemo: players are represented by robots moving on a plane, arranging teams, and shooting at each other. Fig. 11 depicts a screenshot of the RTFDemo with teams: the Score-Table shows the Team-points, as well as the individual points of all connected players. The new `GlobalTeamInfo` object is a managed `Global` with multiple update operations with several relations among them. RTFDemo creates a team for each zone defined in the game world. Players automatically join the team of the zone their avatar is created in; they obtain points for hitting robots of other teams, and lose points for hitting robots of their team. Destroying a robot scores extra points for the player and a point for his team. For each team, this information is stored in a `GlobalTeamInfo` and displayed at each client.

Figure 11: A managed GlobalTeamInfo implementation based on the RTFDemo.

We analyse the performance of GOMS by comparing to RTF without GOMS. An additional test application was implemented: it evaluates the *update delay time (UDT)*, for the default and managed globals using the different ordering constraints. The UDT for an action at a process is the time between the action being sent from a client and the time when the process executes the update representing the effect of the action.

Fig. 12 shows the test setup consisting of three zones without zone replication, instancing or portals. Zones are as-

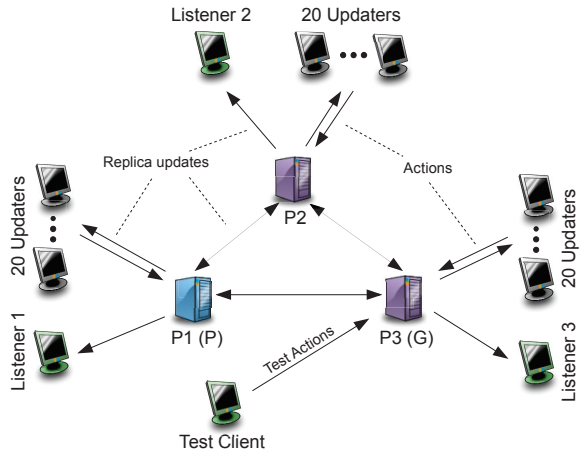

Figure 12: Measuring UDT (Update Delay Time)

signed to processes P_1, P_2 and P_3, correspondingly, each running on its own machine. Process P_1 is the primary server (P) for the state-based default globals, P_3 is the generator (G) for unique sequence numbers (USNs) for the total ordering approach of GOMS. Each server runs on a separate machine, and the clients run on four other machines, up to 20 on each machine.

Each test was run with 20 global objects and 20 clients (Updaters in the figure) per server, summing up to 60 global objects and clients. Each group of 20 clients was started on its own machine and connected to one of the servers. The server processes were started with a ticklength of 40 ms, while the client processes have a ticklength of 20 ms. The clients send an update message to their server each 50 ms, updating a random global.

To measure the update delay times, every 250 ms a special test-client sends a test action to its server, signed with the system time when the action was initiated. On the same machine, three listener clients are started, each connected to one of the three servers. Once the action arrives at the server, its start-time is written to the global object and distributed as a global update. The listener clients detect these special updates and measure their individual update delay times. The tests run for about 2.5 minutes, resulting in over 500 measured values per listener.

To simulate different real-time loop saturations, an adjustable *sleep* phase is integrated into the server real-time loop. This simulates complex calculations and load resulting from non-global objects. To measure the impact of saturation on the performance of the particular state replication and ordering strategies, additional tests were performed with different loads of 0%, 25%, 50%, 75% and 100% of the servers' ticklength. The resulting individual UDT for the three listener clients are averaged to a single value.

Figures 13 to 15 show the average UDT for the different real-time loop saturations. As a reference, each diagram shows the measurements for the default globals using the state-based approach: for a client connected to the primary server (DEF_P in the figures), and for a client connected to a non-primary server (DEF_NP). The two test cases for default globals show an ascending average UDT for situations with an increasing load because updates on default global objects are distributed at the end of each tick. The longer a tick lasts, the later the updates are sent. As expected, the updates of a client connected to a non-primary server of a default global show a higher UDT than the updates sent to the primary server of the global.

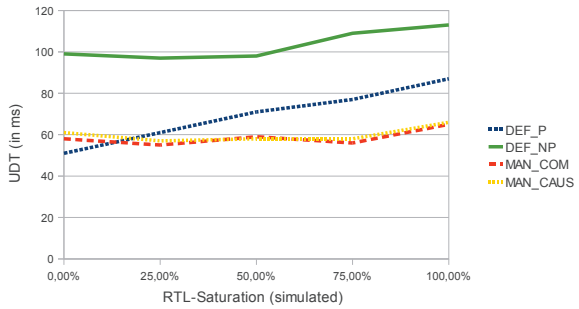

Figure 13: Average UDT for commutative (MAN_COM) and causal (MAN_CAUS) updates vs. state-based approach (DEF_P and DEF_NP)

Fig. 13 shows that commutative (MAN_COM) and causal updates (MAN_CAUS) perform better than the default globals even if the servers ticklength is exceeded regularly. The reason is that update operations are not delayed until the end of the tick, but sent out as soon as possible. The transmission of the global updates by GOMS is independent of the real-time loop thread.

Additionally, commutative updates are always applied at the start of the next server tick, such that complex calculations in the application real-time loop do not effect the UDTs of commutative operations. Causal updates might have to be delayed to the next tick if caused-by related updates are missing at the current process. The results show only slightly higher UDTs for causal updates, which indicates a low number of delays for causal updates.

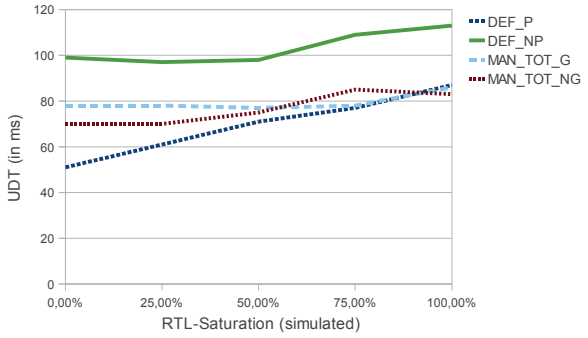

Figure 14: Average UDT for total updates vs. the state-based approach (DEF_P and DEF_NP)

As shown in Fig. 14, total operations (MAN_TOT_G for a client connected to the sequence number generator, and MAN_TOT_NG for a client connected to another server) are also slightly affected by the real-time loop saturation. A total operation, which cannot be applied due to missing

preceding updates, has to be buffered until the next tick. This is reflected in the slightly ascending UDTs for higher saturations. The average UDT is settled between the values of DEF_P and DEF_NP.

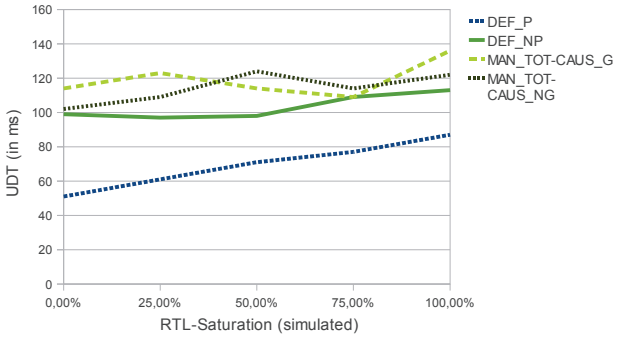

Figure 15: Average UDT for total-causal updates vs. the state-based approach (DEF_P and DEF_NP)

Fig. 15 shows that total-causal updates (MAN_TOT-CAUS_G and MAN_TOT-CAUS_NG) are strongly affected by the RTL-saturation. While total updates can be sent immediately when their USN is determined, the transmission of total-causal updates has to be delayed until they can be applied at the local process. As the applicability of updates is checked only once per tick, a total-causal update can be delayed for one or even multiple ticks.

Our evaluation proves that update ordering is a preferrable alternative to the state-based approach. Although the evaluation has revealed a potential to improve the UDT for processes sending total operations, the operation-based total ordering approach still performs better than the state-based approach for non-primary processes. The update ordering approach has also been proven as partially independent of the real-time loop saturation, resulting in shorter UDTs for situations with high load, i. e., 100% loop saturation.

6. CONCLUSION AND RELATED WORK

This paper studies the consistency of replicated global objects under concurrent update operations in a soft real-time environment implemented on multiple servers. We demonstrate the drawbacks of the traditional state-based update mechanism which result in performance losses and restricted usability. We implemented an operation-based approach which allows the application developers to achieve more concurrency and thus improve users' immersive experience.

The problem of consistency in distributed environments including online games has been actively investigated recently. While some authors consider the consistency of client-side part of single-server online games [7], we rather focus on the more challenging problem of concurrent update operations on global objects in multi-server environments. Previous work [5, 6] to improved the correctness of the state-based approach in multi-server online games while preserving correctness. The time warp approach of [5, 6] delays the delivery of state updates to ensure correctness, whereas [4] also considers the correlation of updates operations and proposes a rollback-based algorithm which improves the performance for unrelated updates. The latter approach is similar to the update ordering approach [1] which we adopt in our paper. The important advantage of our implementation (GOMS) is that it allows to adjust the consistency and performance of replicated global objects to the semantics of the application operations, thus providing more flexibility to the application developer. Furthermore, application developers can choose which approach - update ordering using GOMS or state-based replication - is the most suitable for their application.

We implemented the update ordering approach in the Global Object Management System (GOMS) and integrated it into our Real-Time Framework (RTF) which is widely used for developing different kinds of interactive distributed environments. The experimental evaluation of the various replication mechanisms and ordering constraints proved that commutative, causal and even total operations are performed significantly faster, as compared to the state-based approach using the primary copy protocol.

7. REFERENCES

[1] Wanlei Zhou, Li Wang, Weijia Jia, "An analysis of update ordering in distributed replication systems", in Future Generation Computer Systems, pp. 565-590, 2004.

[2] Coulouris G., Dollimore J., Kindberg T., "Distributed Systems Concepts and Design", Addison Wesley, 2005.

[3] Glinka F., Ploss A., Gorlatch S., Müller-Iden J., "High-Level Development of Multiserver Online Games", Int. Journal of Computer Games Technology 2008, Article ID 327387.

[4] Ferretti S., Roccetti M., "Fast delivery of game events with an optimistic synchronization mechanism in massive multiplayer online games", in Proceedings of ACE '05, pp. 405-412, 2005.

[5] Mauve et al., "Local-lag and timewarp: Providing consistency for replicated continuous applications", in IEEE Transactions on Multimedia 6(1), pp. 42-57, 2004.

[6] Müller J., Gössling A., Gorlatch S., "On correctness of scalable multi-server state replication in online games", in Proceedings of NetGames 2006, 2006.

[7] Li F., Li L., Lau R., "Supporting continuous consistency in multiplayer online games", in Proceedings of MULTIMEDIA '04, pp. 388-391, 2004.