Monitoring Large-Scale Location-Based Information Systems

Hammad Khan, Julien Gascon-Samson, Jörg Kienzle, Bettina Kemme

School of Computer Science, McGill University
Montreal, QC H3A 0E9, Canada
Email: Julien.Gascon-Samson@mail.mcgill.ca,{Joerg.Kienzle,Bettina.Kemme}@mcgill.ca

Abstract—**Monitoring the state of a distributed virtual world is challenging for several reasons: 1) the distributed information must be gathered in real-time without affecting the performance of the information system; 2) in large-scale systems it is impossible for a single node to collect and process all the data; 3) the vast information must be filtered and aggregated according to what the human observer wants to focus on, and 4) the point of interest of the observer can change frequently. In this paper we present and evaluate a non-intrusive monitoring middleware that addresses these challenges by dynamically partitioning the geographic map (e.g., of the virtual world or the game) in terms of map objects and (expected) state changes. We assign a different collector node to each of these partitions to collect and pre-process the data, and forward it to a central monitoring node. Furthermore, we provide mechanisms to efficiently filter and aggregate location changes, the pre-dominant changes in location-based information systems. We describe a specific monitoring setup that takes advantage of the replication model that is common in many virtual worlds and multiplayer games to collect the data. Finally, we present extensive performance results that show the trade-offs between scalability, precision, and real-time performance.**

I. Introduction

Massively multiplayer games (MMOG), where thousands of players are connected to a virtual world, are probably the most well-known example of location-based information systems. In such games, players perform game actions concurrently, changing their own state and location in the game world as well as affecting other game objects and players. Each of the players typically only sees a small part of the world (its vision range), namely the close vicinity of the location in the world the player currently resides in. As the world continuously changes, each of the players has to be informed about the actions of others players in its vision range. Thus, game engines represent a combination of (i) a large distributed information system, where players and game objects represent the state and state changes are requested by many different clients (the players) concurrently, (ii) a replication infrastructure where each object typically has a master copy that accepts updates and many secondary copies residing at the players' clients, and (iii) a large-scale data dissemination infrastructure in order to efficiently send state changes to these secondary replicas.

But games, and virtual worlds in general, are not the only applications that follow such an architecture. Large-scale emergency and rescue endeavors or military actions follow a similar pattern. More recent are urban planning and traffic control environments, where, e.g., vehicles continuously change their location and want updates about other vehicles in their neighborhood. Other applications include cellular network deployments where millions of users are located across a large geographic area, are continuously moving, and are communicating with each other.

In such environments, there are also global stakeholders that require a holistic view of the current game/map/world state. These *observers* typically want to monitor the overall state continuously. Often, the observer wants to have the option to have a full view of the entire world, in which case, information of individual objects must be highly summarized, not only to be visibly understandable but also as the single node showing the data will not be able to hold/receive/process detailed information of all data objects in the system. Additionally, observers want to be able to quickly zoom into different regions with varying zoom factors up to seeing all details in a small area. Both global and local views are, e.g., useful for game developers after they add new features to the game in order to monitor their effect at the global or local stage. As well, system administrators might want to detect any imbalance situations in real-time, e.g., when many players flock at the same time to a central location of the world as this could overload servers quickly. In this case, the system must be able to see that crowding at the global view, as well as provide more detailed views when zooming into the crowded area. Similarly, for traffic control systems, traffic jams or accidents that hold the traffic need to be detected and analyzed quickly.

Monitoring is important not only for location-based information systems, but for any application domain where state can change quickly such as sensor networks or large-scale server systems such as web-server farms. Various monitoring and logging architectures have been proposed in these contexts, and the overall architecture of these systems often follows a similar pattern: (i) in order to handle the large amount of data, the information flow is organized in a hierarchy, where data is collected locally and then streamed upwards a monitoring tree until it reaches a final processing and observer node. (ii) On the way, data is filtered and aggregated. Filtering means that some of the collected data is discarded and only data of interest is retained and forwarded. Aggregation means that many individual data records are summarized and forwarded in a more compact, aggregated format.

Nevertheless, as applications are very different from domain to domain, it is not trivial to transform such a general monitoring pattern into a concrete system for a specific domain. We have identified the following specific requirements for location-based information systems, in particular MMOGs and virtual worlds. 1) Despite having thousands of players (clients), and the world state being distributed among many servers (and clients), the particular state of interest for monitoring must be gathered in near real-time without affecting the performance of the underlying application. 2) The monitoring infrastructure itself must be distributed in order to scale with the underlying application. 3) As the amount of information and the amount of changes per time unit are extremely large, application-specific filtering and aggregation are necessary; this holds in particular for location changes, which are extremely frequent and usually of high interest. 4) The focus of interest for monitoring can change quickly, requiring efficient zoom-in/zoom-out techniques as well as move techniques within the collection, filtering and aggregation framework. In this paper, we present a monitoring framework that fulfills these requirements through various means:

Collecting: Our framework consists of *collector nodes* that observe the state changes, filter and aggregate them, and forward them to a *monitoring node* that merges and post-processes the data. The architecture is scalable as different collector nodes are dynamically assigned to different regions of the virtual world. In particular, if the underlying virtual world implementation supports load-balancing, then our architecture exploits these mechanims for its own load-balancing.

Observing and Filtering: We propose a non-intrusive integration of the monitoring framework by taking advantage of the replication model most multiplayer game engines and virtual worlds implement. By exploiting this replication model, collector nodes can observe updates on objects for which they are responsible without putting hooks into the underlying virtual world implementation. With location changes being one of the most frequent and important state changes in virtual words, we present filtering and aggregation mechanisms that are targeted to handle these location changes in an efficient and holistic way. In particular, objects that are neighbors in the virtual world can be aggregated to groups to provide information at coarser levels. The system ensures that such group aggregation is consistent even if the information is distributed across several collector nodes. We integrated our framework into the massively multiplayer game prototype Mammoth [1] and experimentally analysed the trade-offs between scalability, precision, and real-time performance.

II. Background

In MMOGs, players collaborate or compete in a virtual world. Each player sees a graphical representation of the world and controls an *avatar* which can perform actions, e.g., move, pick up objects, or talk to other avatars. Each player maintains a copy of the (relevant) game state on his node. When one player performs an action that affects the world, the game state of all other players affected by that action must be updated.

A MMOG has to be *scalable* (handling any number of simultaneously connected players), *reliable* (tolerating node or communication failures), and *fair* (treating all players equally despite distribution). It must also provide a consistent view of the world to all players, despite network delays.

A. Game Components

Interest Management (IM) determines for each player the objects she needs to know about, e.g., all game objects and players that are in her vision range. Calculating interest is done within short time intervals, or whenever a player or object changes state, e.g., changes position.

Replication Model: Replication is necessary for state management and update dissemination. Most systems use a primary copy replication model. Each object and player in the system has one primary copy (also referred to as *master copy*) and many secondary copies (also referred to as *replicas*). All updates are performed at the master copy that contains the latest and correct state of the object. The master then broadcasts the state changes to all replicas. Conceptually, the replicas subscribe to the changes that occur at the master copy.

Whenever IM determines that a player or object has entered the vision range of another player P, P needs to receive a copy of that player or object. Once the copy is received, the primary/secondary replication model ensures that P receives all subsequent updates to this player/object. Once the player or object leaves P's vision range, the copy can be discarded. IM is important not only for game semantics, but also to reduce the total number of update messages in the system as every player only needs updates for a small subset of game objects.

B. Distributed Architectures

In single server systems, the server maintains the master copies of all objects and performs IM. All clients connect to the server and receive replicas of the objects they are interested in. Whenever a client makes a change to his avatar, the change request is sent to the master on the server, executed there, and the changes are then propagated to all replicas. Whenever IM on the server determines that an object/player has entered the vision range of a client's avatar, the server must send a replica to the client. However, a single server system can only handle a limited number of players; thus, existing MMOGs typically run on a multi-server, P2P, or hybrid distribution architecture.

In *multi-server architecture*s, the game is typically divided into regions, and each server performs IM and update dissemination for one region. The first approaches proposed static regions [2], [3], [4]. For better load-balancing, newer approaches propose dynamic regions where region sizes can be adjusted, e.g., by splitting and joining regions, or by resizing them [5]. To be more flexible, some approaches split the world into many small fixed-sized regions, referred to as *tiles*, and each server is responsible for many tiles [6]. Load-balancing then involves moving some of the tiles from an overloaded server to a less loaded server. A challenge is that the vision range of players can cover regions residing on different servers when the player resides close to the border of two regions.

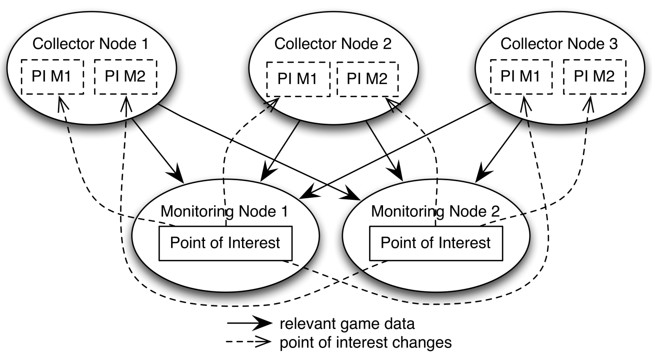


Fig. 1: Proposed Monitoring Architecture

In *peer-to-peer architectures*, the clients are responsible for maintaining the game state and perform IM [7][8][9]. Some approaches follow a super-peer approach where some of the clients take over server functionality, i.e., they are assigned regions for which they perform update dissemination and IM.

In *hybrid approaches*, servers perform some of the tasks while clients perform others [10][11]. For example, IM is done by servers, while object management is performed by clients.

Our implementation is based on the MMOG middleware *Journey [12]*. IM in *Journey* is based on triangular tiles. Each player sees game objects in the tile she resides plus a configurable neighborhood of tiles. Servers take care of IM for a *cell* that consists of a set of tiles, and determine the vision set for each player in that cell. For load-balancing, tiles can be transferred between cells., e.g., if players flock to a small subset of tiles, the cell server responsible for these tiles can transfer some of the tiles to the cell of another server. *Journey* uses the replication model described above. Cell servers have copies of all game objects in their cell plus game objects that are in tiles that are neighbors of their cell, so that they can determine all objects their players might be interested in. The master copies can either also reside on the servers (multi-server) or can reside on some of the clients (hybrid).

III. Monitoring Architecture

Monitoring the state of a virtual world is challenging for several reasons: 1) the distributed information must be gathered in real-time without affecting the game itself; 2) it is impossible for a single node to connect to all player nodes and collect all necessary information; 3) the monitoring architecture must be integrated with the underlying infrastructure in a seemless way; 4) filtering and aggregation must be done in a location aware manner and according to what the monitoring person wants to focus on, and 5) the point of interest of the monitoring person changes frequently. In this section, we discuss our monitoring architecture and how it integrates with the game infrastructure addressing challenges 1-3. Filtering and aggregation are only described at a high level. Section IV provides details on how to perform dynamic and adaptive location-focused filtering and aggregation.

A. Overall Collector/Monitor Architecture

Our proposed monitoring and filtering architecture for virtual worlds is shown in Fig. 1.

Filtering and part of the aggregation is done on the *collector nodes*. Each collector node observes a different region of the virtual world. The number of collector nodes depends on several factors: each collector has limited network connections and bandwidth, which limits the amount of game state updates a collector node can receive; additionally, each collector has limited memory and processing power, which limits the amount of filtering the node can perform. Unfortunately, the many technical aspects that come into play when it comes to network and processing power limitations make it hard to analytically determine the ideal size of the region of the virtual world that should be assigned to a collector. As a result, an adequate size usually needs to be determined by experimentation, and is ideally dynamically configurable.

At run time, the monitoring node and the collector nodes are in constant communication. While the collector nodes continuously forward the filtered data of interest to the monitoring node, the monitoring node communicates changes in *point of interest* to the collectors, so they can adjust the filtering accordingly. The point of interest has 3 attributes: 1) what part of the virtual world the monitoring node currently focuses on, 2) the kinds of data the monitoring node currently wants to know about, and optionally 3) what level of detail of information is requested in terms of accuracy and/or timeliness.

B. Architecture Scalability Discussion

CPU use, memory use, network bandwidth and number of network connections are the limiting factors at the collectors and at the monitoring node. Our proposed architecture can deal with these constraints as follows:

Adding additional collectors: Adding collectors to the system reduces the size of the virtual world that a collector is in charge of, which effectively reduces the number of game objects that the collector needs to handle. This can address CPU overload, memory use, incoming bandwidth and network connection limitations at the collector, but increases incoming bandwidth and number of connections on the monitoring node due to the additional collectors.

Partitioning collectors according to kind: Instead of only relying on space partitioning to determine what game objects a collector is in charge of, the *kind* of game object can alternatively or additionally be used. E.g., for a given region of the virtual world, a collector could be in charge of monitoring the position of the elves, while another keeps track of the position of the orcs. This can address CPU overload, memory use, incoming bandwidth and network connection limitations at the collector, but increases number of connections on the monitoring node to handle the additional collectors.

Adjusting the filtering: CPU use and outgoing bandwidth on the collector, as well as incoming bandwidth on the monitoring node can be decreased by adapting the filtering granularity and frequency. Coarsening the granularity of the filtering, i.e., aggregating game data more aggressively, decreases the precision of the information that is forwarded to the monitoring node. Lowering the filtering frequency decreases the timeliness of the data available at the monitoring node. It depends on the game, precision and real-time requirements of the observer if adjusting the filtering is an option.

Adding intermediate collectors: Fig. 1 shows one layer of collectors. If the virtual world is so big that the number of collectors needed to monitor its state exceeds the number of connections that a monitoring node can support, then our architecture needs to run in a multi-layer configuration: an additional layer of collectors is introduced – the "supercollectors". The idea is that while collectors observe game object updates, the supercollectors observe collector updates. The supercollectors then filter that information further and forward it to the monitoring node. This technique can address number of connection limitations both on the collector and monitoring nodes, but decreases the timeliness of the data available to the monitoring node because all game state updates now additionally flow through the supercollectors.

C. Integrating Monitoring into the Underlying Infrastructure

This subsection describes how we integrated our monitoring approach into *Journey*. We believe that integration with other game engines that follow a similar architecture (and many do), will be conceptually very similar.

a) Game World Partitioning: To determine a suitable partitioning of the virtual world for monitoring purposes it was natural to reuse the regions that *Journey* already creates in order to distribute IM to servers. The IM partitions are created in such a way that the servers are powerful enough to do IM for all the game objects in the partition: the servers must have enough network capabilities to be able to register for replicas of all game objects in the partition, and they must have enough processing power to do IM for them. The requirements for collectors are similar: they must have enough network capabilities to be able to receive updates for all game objects in the region they are monitoring, and they must have enough processing power to filter the collected data. If there is a node powerful enough to be a region server, then there should also be a node that can act as a collector for that same region. Consequently we integrated monitoring into *Journey* by assigning per default one collector node for each IM region.

b) Data Collection: Since the data dissemination mechanism that sends updates from masters to replicas is highly optimized in game engines, the obvious way for collector nodes to be made aware of game state updates in their region is to create replicas of all game objects of a region on the collector. The nodes holding the master objects then send all relevant game state object updates directly to the collector node, just like for any other game client. With this, we rely on the optimized data dissemination mechanism of the underlying engine, and the monitoring framework does not need to reimplement this important functionality.

c) Collector $\leftrightarrow$ Monitor Communication: We also chose replicated objects to communicate between the collectors and the monitoring node. We encapsulate the filtered data that needs to be communicated to the monitoring node in so-called *monitor objects* (MOs). Each collector has a certain number of MOs, and a monitoring node registers for replicas of all MOs it is interested in. As a result, whenever a collector node updates the state of one of its MOs, the replication module automatically sends an update message to the replica on the monitoring node. By controlling how often a collector node updates its MOs, we can control how much data and how often data is sent to the monitoring node.

The monitoring node needs to communicate the current point of interest to all collector nodes. To this aim, the state of the point of interest, i.e., the kind of information that is being monitored and the currently observed area of the virtual world, is stored in a *view object*. The monitoring node has the master copy, which is updated whenever the human observer changes the camera position or zoom factor, or selects different game data to be monitored. All collector nodes have replicas of the view object, and consequently are immediately informed of any changes to the observer's interest.

a) Maximization of Monitoring Precision: The design of our monitoring architecture makes it possible to maximize the precision of the gathered data given the maximum available network bandwidth for a monitoring node. Since each collector updates its MOs at a given frequency, the amount of data sent from one collector node to the monitoring node is maximally equal to $\#\mathrm{MOs}/\mathrm{updateInterval} \times \mathrm{bytesPerUpdateMessage}$. Given the number of collectors needed to monitor a virtual world, the ratio $\#\mathrm{MOs}/\mathrm{updateInterval}$ is determined as:

$$\frac{\#\mathrm{MO}}{\mathrm{updateInterval}} = \frac{\mathrm{availableBandwidth}}{\#\mathrm{collectors} \times \mathrm{bytesPerMessage}} \quad (1)$$

A tradeoff decision must be made: the more MOs are used, the longer the update interval must be; if less MOs are used, updates can be sent in shorter intervals. The ideal configuration depends on the current point of view, i.e., on the kind of information gathered and the part of the world that is in focus.

b) Multiple Monitoring Nodes: By using replicated objects for communication, it is easy to support multiple monitoring nodes. Each monitoring node has its own view object. Each collector node has several view object replicas. Collectors then create a separate set of MOs for each view, and performs for each set the mapping from game state object to MO according to the point of interest of the corresponding view.

While this works for a small number of monitoring nodes, it is not feasible when the number of monitoring nodes reaches the hundreds or thousands due to resource limitations on the collectors. In this case, additional optimizations such as the four scalability strategies discussed above – adding additional collectors, partitioning collectors according to kind, adjusting filtering, and adding intermediate collectors – can be applied. An additional strategy that can be used is to configure the collectors to only keep replicas of the view objects of monitoring nodes that are relevant to them. In *Journey* this can be done using the provided IM service. A collector announce that it is interested in view objects only if their view region intersects with the region that the collector is monitoring. IM then ensures that the collector holds a replica of a view object only as long as it fulfills that criteria. This optimization results in receiving less view updates, decreases CPU usage on the collector and reduces the incoming bandwidth and number of connections from monitoring nodes.

c) Load Balancing: Journey provides support for load balancing by dynamically adjusting the size of the region assigned to a server [13]. The virtual world is partitioned into many small tiles, and a server region is defined by a set of connected tiles. The load of a server depends directly on how many game objects are in its region: the server has replicas of all objects and therefore receives all state updates, and must perform IM for all players. When a server experiences high load, e.g., because a significant number of players decide to gather in the region handled by the server, *Journey* transfers some of the border tiles of the region of the overloaded server to an adjacent server.

The load on the collector node is also directly dependent on how many game objects are in a monitoring region, since, just like servers, the collector needs to have replicas of all game objects in the region. Thanks to the fact that we chose to align server regions and collector regions, the load balancing done by *Journey* is also taking care of preventing collector overload.

IV. Filtering and Aggregation

There are two filtering techniques that can be applied at the collector. One way to reduce information is to not forward every change in state, but to only pass along the game state changes once they are significant enough (which needs to be determined based on game semantics) or once a minimal time period has elapsed since the previous update. This unfortunately introduces an artificial lag between the actual game state and the one forwarded to the monitor. The other way to reduce information is to aggregate changes by calculating statistical data about the observed state changes, and to only forward that summary data to the monitor. Which aggregation algorithm works well depends on game semantics.

This section focuses on how our architecture can be configured to monitor a very common kind of game data: positions of players. Keeping track of player distribution in a virtual world is essential, e.g, to foresee performance problems when players flock to a region of the virtual world, to observe what paths the majority of players are taking to complete the game goals, or to monitor the behaviour of computer controlled players.

A. Player Groups

To reduce the amount of player position updates that need to be communicated, our position collector nodes use both filtering techniques described above: 1) the information quantity is reduced by sending updates of the group positions in fixed time intervals, and 2) the information quantity is reduced by mapping the players in the virtual world to a fixed number of groups based on player proximity. Only the position of the group (i.e., the average of the positions of all the players in the group), the radius and the size of the group are communicated. This effectively limits the amount of bandwidth used between a collector and the monitoring node for forwarding position updates: the number of update messages sent is maximally equal to $\#\text{groups} \times \text{updateFrequency}$.

To monitor player positions, we designed a *Group Monitor Object* (GMO) that represents a group of players, i.e., it stores the number of players that are part of the group, the radius of the group, and the group centre, i.e., the average position of the players. At every update cycle, the collector node updates the state of the GMOs based on the received position updates of players in the monitored region.

One decision that needs to be made is how to assign players to groups. The assignment could be done on every update cycle, thus minimizing the sum of all position errors. Unfortunately, this strategy results in situations where, due to small player movements, on each update cycle, a given player is likely to be assigned to a different group. This behaviour might be adequate if the monitoring node wants to display high-level player density information, but fails if the intent is to be able to monitor player group behaviour. The strategy that we decided to adopt assigns players to the optimal GMO based on their position *when they enter the area of interest*. Once assigned, the player remains with the same GMO, unless she moves considerably away from the group centre.

To do that, the following algorithm is used. First, based on the current view object, it is determined what part of the world is currently being observed. Then, the current set of players that are in that "area of interest" is determined. This set is compared with the set of players that are already assigned to GMOs. Players that have newly entered the area of interest are flagged as such, players that have left the area of interest are removed from the mapping, and GMOs that are as a result left unused are marked as "free". Next, the used GMOs are compared in pairs to check whether they are close to each other. If yes, then the two groups are combined into one, freeing one of the GMOs. Then, for each player that is already mapped to a MO, the algorithm determines if the player has moved too far from the group centre. If yes, the player is removed from the group and flagged to be reassigned. Finally, for each player that needs to be (re)assigned, we check if the closest GMO is within a given maximum distance. If this is the case, then the player is added to this GMO. Otherwise, if there are still free GMOs, the player is assigned to a new GMO, thus starting a new group. In case there are no more GMOs, the player is added to the closest GMO as a last resort, even though its centre might be farther than the maximum distance.

B. Additional Filtering at the Monitoring Node

One might think that it is always best for the monitor to display all information received. In certain situations, though, this leads to an uneven level of detail of displayed information. This can be bothersome for aesthetic reasons, but can also interfere with the interpretation of the data.

For instance, when a human observer pans through the world at a fixed zoom level, she might at a given time observe a region that is handled by a single collector, whereas at other times she might be observing a region that is covered by multiple collectors. Since each collector uses a fixed amount of MOs, the two situations differ significantly in the amount of information received at the monitoring node. For example, when monitoring player positions and x GMOs are used per collector, the monitor receives maximally x group positions when observing a single-collector region, whereas it receives maximally nx group positions when observing a region that is handled by n collectors. This results in an inconsistent level of detail when panning. When moving from a view that is covered by a single collector to a view covered by two collectors the number of groups shown on the screen could double. Also, since grouping of players is done at the collectors, all players within a group belong to the region handled by the collector. As a result, if, e.g., a group of 10 players is standing close together inside a region handled by a single collector, the monitoring node would receive position information consisting of one group of 10 players. If, however, the same 10 players would be standing in the same configuration on the border of two collector regions, the position information received at the monitoring node would consist of two groups of 5 players. A human observer might think in the later situation that there are no groups of players that have more than 5 players, which is not accurately reflecting the actual state of the virtual world.

To guarantee a consistent level of detail of information regardless of the current point of view of the observer, the monitor needs to perform additional filtering of the information received. The nature of this filtering depends, of course, on the kind of information that is observed, the specific filtering algorithm that is used on the collectors, and what the monitored information is eventually used for. In the case of monitoring player positions, the monitoring node needs to detect when the view of the observer covers more than one collector region. If this is the case, the monitoring node needs to analyze if groups of players that are located at the region borders need to be merged. In total, in our system we decided that the number of groups displayed in the view should not exceed the number of GMOs on a single collector.

V. Experiments

Mammoth [1] is a testing and research framework for MMOGs. It allows researchers to conduct experiments in a controlled but real-world like environment. The *Journey* middleware described in Section 2 is part of *Mammoth*, but *Mammoth* offers more, e.g., easy replacements of components, an infrastructure to add non-player characters (NPCs) and to define complex AI algorithms that determine their actions. With the help of these NPCs, we can conduct real-world experiments (as opposed to mathematical simulations) involving hundreds of player machines, and gather the results using mechanisms built into *Mammoth*.

We have implemented our monitoring architecture in *Mammoth* on top of *Journey* to assess our approach by running a set of real-world experiments. These experiments were conducted at the School of Computer Science labs, McGill University, using over 100 separate machines to run the clients, collectors, monitors and IM servers. All machines run Linux, have a 2.8 GHz dual-core processor or better, and at least 4GB of RAM.

In all experiments, the human players were replaced with NPCs, which were either controlled by external commands (for the load balancing experiment in Section V-E) or were allowed to wander randomly from waypoint to waypoint (for all other experiments). The rationale for using NPCs is provided in [14], which shows that, given correct starting conditions and positions, NPCs can approximate the actions of human players and therefore provide realistic results in experiments. Wanderer NPCs change destinations on average at a frequency configured for each experiment. Each destination change results in a state update message that is propagated from the master to the replicas. The game engine then updates the position of the master player object every 50 ms according to the player speed until the destination is reached.

A. Scalability – Delay due to Redirection

The game information – player positions in our experiment – goes through many stages before it reaches the monitoring node. Any change in position on the node that hosts the master player object is broadcast to all replicas, one of which is at the collector node. There, the player position is mapped to a GMO, and the state of the GMO is updated. That change is then forwarded to the GMO replica located on the monitoring node. This indirection introduces a delay, which can be divided into the time taken due to additional communication and the time spent filtering on the collector, i.e., assigning the players to groups and updating the group position.

The first experiment was conducted to determine the scalability of our approach, reporting on the delay introduced by our architecture as the number of players increases. We instrumented a monitoring node to not only register for the GMOs of the collectors, but also to register for the player replicas directly. In *Journey*, every state modification on a player object is tagged with a serial number. We instrumented the collectors to add these serial numbers to the GMO when players are mapped to it. This makes it possible to measure on the monitor the time delay between receiving the original position update from the player replica, which is sent directly from the player node to the monitor, and the corresponding update to the tagged GMO replica sent from the collector node. To measure only the delay due to communication, we configured the collector node to propagate position changes immediately. We ran the experiment on a *Mammoth* map with 1000 initially inactive, uniformly distributed players. We launched 4 *Journey* IM servers, each one responsible for approximately 1/4th of the virtual world. We successively started client machines, which take control of the players, instructing them to move from waypoint to waypoint, changing destination on average every 5 seconds. We repeated the experiment with 3 configurations: one, two and 4 collectors. Each collector was configured with 20 MOs, and we recorded at least 100 measures before adding additional players.

Fig. 2 presents the time (in ms) that elapses on the monitor between receiving a state update directly from the master player node vs. receiving the state update through the GMO from the collector. For each configuration, the average delay time and the 95th percentile is shown. According to [15], a delay of 300 ms is acceptable for MMOGs, and hence

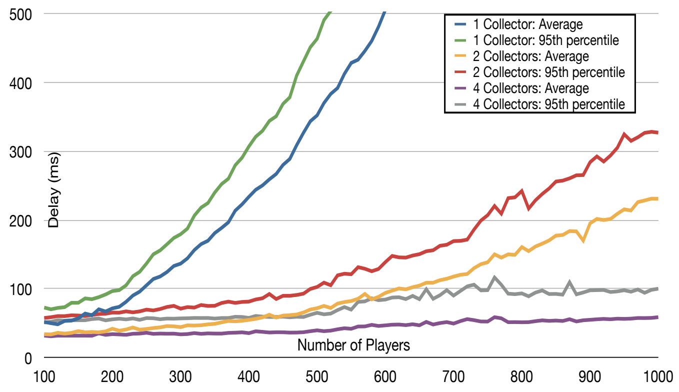


Fig. 2: Delay due to Filtering / Indirect Communication

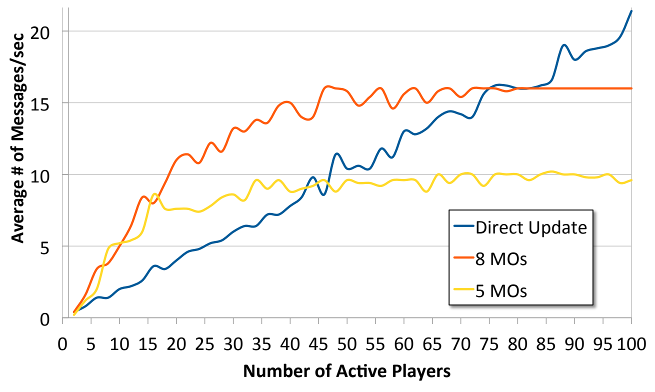


Fig. 3: # Update Messages received on Monitoring Node

the minimum acceptable delay for monitoring an MMOG is probably of the same order of magnitude, maybe slightly greater. Our results show that with 1 collector we can support 400 players without exceeding 300 ms for 95% of the clients. With 2 collectors we can handle 900 players, and with 4 collectors and 1000 players we measured average delays of 58ms, with 95% of the delays being under 100 ms. We were not able to reach higher player numbers, as the CPU usage and network bandwidth on the instrumented monitoring node saturated for higher player numbers, because it had to additionally receive and process the updates of the players in order to be able to take the measurements.

It has to be pointed out here that the collector and the monitor were located on the same LAN. As a result, the network delay caused by the additional message sending is small. In a WAN setting, the delay would increase accordingly.

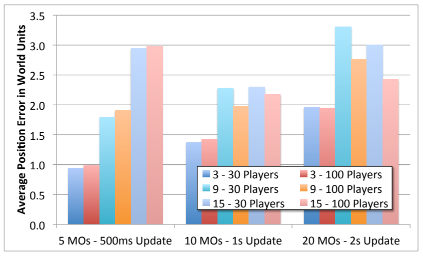
The CPU and network bandwidth used on the collector increases linearly with the number of connected players. In the single collector configuration at around 220 connected players, the CPU load due to the execution of the mapping algorithm that maps all players to 20 GMOs starts to influence the overall delay, and in the long run prevents the system to scale above 500 players. In the 2 collector setting, each collector has to deal with only half of the players on average. Also, there are now 40 GMOs available to group players, and as a result the delays introduced by the mapping algorithm only start affecting the delay at around 440 connected players. With 4 collectors, the mapping algorithm load does not prevent the system to scale to 1000 players.

B. Scalability – Limiting Update Messages

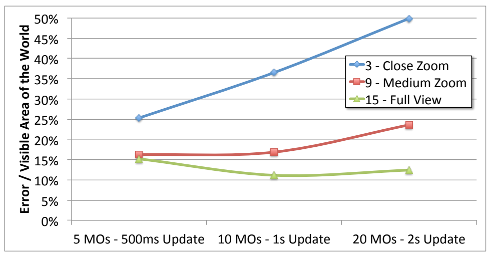
The second experiment verifies that our architecture limits the network bandwidth required to send updates to the monitor. We ran three separate experiments: one with a monitor that is directly connected to all players; one where position updates are filtered by a collector that has 5 GMOs, and one where position updates are filtered by a collector with 8 GMOs. For the tests involving collectors, we configured the mapping algorithm to run twice per second. Each experiment was started in a virtual world that contains 100 non-moving players. Then, every 10 seconds, a Wanderer NPC would connect and take control of one of the immobile players. The movement frequency was set to an average of one change in direction every 5 seconds. Each change in direction results in an update message sent from the node running the NPC to the monitor (experiment 1) or the collector (experiment 2 and 3). As the players starts moving, we measured the number of update messages that were received on the monitor. The update count was taken every 5 seconds, and then averaged to compensate for the randomness of the player movements.

The measurements for this experiment are shown in Fig.3, which plots the number of moving players against average updates per second. The line labeled “Direct Update” represents the results for experiment 1. As expected, the message count on the monitor increases almost linearly as more players are added. Furthermore, the number of update messages is approximately 1/5th of the number of moving players, which corresponds to the 5 second average delay between change of direction that was set for the NPC.

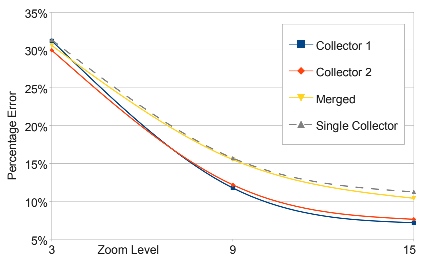
For experiments 2 and 3, the received update messages on the monitor are coming from the collector, and therefore contain filtered data. The numbers show that in both experiments initially the number of messages received by the monitor increases twice as fast as for experiment 1. This is explained as follows: when a moving player that was first mapped to one GMO is reassigned to a different GMO, the states of both GMOs change and hence two update messages are sent to the monitor in the following time slice. Since the players were distributed uniformly across the world, the maximum number of GMOs contain moving players very early on during the experiments. However, as more players start moving, GMOs are being shared, which slows down the increase in the number of update messages. Once a sufficiently large number of players are connected (approx. 50 in our experiments), each GMO changes state at each iteration of the mapping. However, the maximum number of updates is limited by the number of GMOs divided by the update delay between iterations. Since for both experiments the update delay was fixed at 0.5 seconds, the maximum number of update messages that is generated is twice the number of GMOs, i.e. 10 messages/s for experiment 2 with 5 GMOs, and 16 messages/s for experiment 3 with 8 GMOs.



(a) Position Errors vs Zoom Level / Configurations



(b) Relative Error / Visible Area of the World



(c) Error in Player Position Before and After Merging

Fig. 4: Accuracy – Error in Player Position

C. *Accuracy – Error in Player Position*

Our third experiment investigates the inaccuracy of player positions reported on the monitor due to the mapping of multiple players to a single GMO and due to the time delay. The experiment also provides insight on how to determine the optimal tradeoff between #GMOs and update frequency.

We ran several experiments, some on a map with 30 players, some on a map with 100 players, using different areas of interest at the monitoring node, i.e., areas with a diameter of 15 world units, 9 world units, and 3 world units. For our test map, 15 corresponds to a full view of the map (completely zoomed out), 9 shows 35% of the map (medium zoom), and 3 displays 4% of the map (close zoom). We ran the experiments using different configurations of the collectors, but making sure that the maximum used network bandwidth was constant at 10 messages / second for every experiment. We therefore used: 5 GMOs with 0.5s update interval, 10 GMOs with 1s update interval, and 20MOs with 2s update interval. We instructed our NPCs to move on average every 3 seconds at a speed of 0.6 units / second.

We measured the average error in player positions, i.e., the difference between the actual player position and the position of the centre of the group to which the player was mapped to. In the case where a player should be visible on screen but was not mapped to a GMO yet, the error was set to be the maximum viewable distance.

The results are shown in Figure 4a. The first fact to observe is that the difference in the average error in all experiments does not depend on the total number of players in the map. The reason for this is that the size of the GMOs in terms of the area of the map covered is roughly the same in both cases, and only the number of players mapped to a GMO differs. But since we take the average of all the errors in player positions, the number of players in the GMO is not relevant.

We also observe that within each group of measurements for a given configuration, the average error increases with the diameter of the viewed area. The difference in accuracy between the closest zoom level and the full map view is the biggest for the 5 GMO configuration. This makes sense, since at the closest zoom level there are often just very few players visible, and hence 5 GMOs with fast update yields the best results. In contrast, 20 GMOs with slow update is not ideal at the closest zoom level, shown by the high average error reported at zoom level 3 for the 20 GMO configuration. In this case, most GMOs are unused, and the error stems mostly from the slow update interval. However, having more GMOs pays off at higher zoom levels. In our experiment, the 10 GMOs with 1s update interval performs best at full world view.

Although our experiments show that the error increases as the view is zoomed out, the actual perceived error for an observer on the monitor is the error *relative to the view size*. Fig. 4b plots the ratio $\mathrm{error}/\mathrm{visibleArea}$, i.e., the perceived accuracy, for the 5 GMO, 10 GMO and 20 GMO configurations. The results show that error ratio in general is lower for higher zoom levels, since the movement of players with respect to the visible area of the world is small. It also shows that for a close zoom level, the shortness of the update interval is most important for achieving high accuracy. For higher zoom levels on the other hand the number of GMOs becomes important too. However, 10 GMOs and 1s update interval outperforms the 20 GMOs and 2s update interval configuration, which suggests that at some point the error resulting from slow updates cancels the benefits of having more GMOs.

D. *Accuracy – Merging at the Monitoring Node*

The purpose of this experiment is to demonstrate that the observer experiences a varying degree of detail when multiple collectors are used depending on the region that is observed, but that a consistent level of detail can be ensured if additional filtering is performed on the monitoring node as discussed in Subsection IV-B.

For this experiment, Mammoth was setup to run with four servers, and a total of 100 randomly moving NPC players that are equally distributed within the world. In the first experiment, the monitoring system used two collectors, each using 10 MOs and each responsible for two IM regions. At the monitoring node, the group information received from the 20 GMOs was reduced by continuously merging the two closest groups until there were only 10 groups left. The average percentage error between the actual and the reported player position was recorded at each of the collectors, and at the monitoring node after merging. The percentage error was calculated as the error in world units divided by the maximum visible distance from the centre of the view. This error was calculated for the three different zoom levels presented in the previous experiment.

The blue and red lines in Figure 4c represent the average error at each of the collectors. We observe that given the uniform distribution of players, the results for each collector

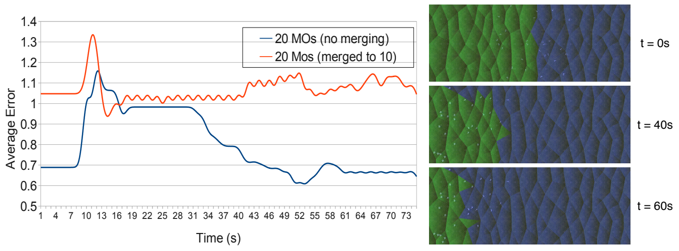


Fig. 5: Effects of Load Balancing on Average Error

are very similar. The yellow line shows the percentage error after the 20 groups have been reduced to 10 groups at the monitoring node. At zoom level 3, we observe that there is no difference before and after the merging. The reason for this result is the fact that at such close zoom level very few players are actually visible, and therefore not all the GMOs at the collectors are used. Since the merging only takes into account the GMOs that are actually used, and since the number of such GMOs is actually already less then 10, no merging takes place. As expected, the result of merging at higher zoom levels increase the average error.

To prove that merging at the monitoring node guarantees visual consistency, a second experiment was conducted with identical setup but using only a single collector for the entire world (and hence no merging at the monitoring node was necessary). The results of this experiment are shown as a grey dotted line in Figure 4c. We observe that the average percentage error when using a single collector is nearly identical to the results obtained after merging groups from multiple collectors.

E. Scalability and Accuracy – Load Balancing

As the virtual world being monitored is divided among multiple collectors, and the players in that virtual world move from one region to another, the load on the individual collectors can change significantly. In cases where a large number of players gather in a small area of the world – a phenomenon commonly referred to as *flocking* – a collector could potentially get overloaded. Fortunately, a load balancing mechanism already exist in *Mammoth* that adjusts IM server regions as players move within the world. This experiment shows that, since in our design collector regions are linked to IM server regions, the *Journey* load balancing also updates the collector regions to ensure that the number of players handled by each collector is balanced. This prevents the incoming bandwidth of collectors from being overloaded. The experiment also illustrates why additional filtering at the monitoring node is necessary to guarantee uniform level of detail of displayed information.

The experiment involved running Mammoth with two IM servers and two collectors with 10 GMOs each. As a result, the world is partitioned into two regions, shown in green and blue in Figure 5. 40 NPCs that are distributed uniformly in the central region of the world are then instructed to move to the left, which results in increasing the load on the green server (and consequently the associated collector). The evolution of the region topology (at 0 seconds, 40 seconds and 60 seconds) is illustrated in Figure 5 on the right. The average position error between the group position and the actual player position was continuously measured on the monitoring node while the *Mammoth* load balancing mechanism readjusted the regions assigned to the servers (and hence to the collectors) to contain again roughly the same number of players.

The results are presented on the left side of Figure 5. The blue line depicts the average error of the group data received from the collectors on the monitoring node compared to the actual player positions. We can observe that as most of the players move to the region covered by a single collector, the average error increases sharply. The reason for this increase is attributed to the fact that in the balanced case all 20 GMOs are being used in the filtering process for the 40 players. However, when players move to a single collector region, only 10 of the GMOs can be used. We also observe that as load balancing readjusts the cells to distribute the players evenly, the error decreases slowly. The screen shots shown in Figure 5 were taken at the beginning of the experiment, after 40s (while the load balancing mechanism is readjusting the size of the regions) and after 60s (when the regions stabilized again).

As explained in section IV-B, if a uniform level of detail of monitored information is more important than accuracy, additional filtering needs to be done at the monitoring node. The red line in Figure 5 depicts the average error when merging on the monitoring node is enabled for visual consistency. The numbers show that in this case the accuracy is much more consistent while the players move across collector regions, because the number of groups displayed to an observer is at most 10 even if multiple collectors are active. The short increase in error at 11s is due to the fact that the so far immobile players all started to move at the same time.

VI. Related Work

To the best of our knowledge, there has been no prior work on monitoring approaches for virtual worlds. However, monitoring for distributed systems in general has been studied in detail, and our approach has been inspired by some of the existing solutions.

[16] proposes a technique for gathering telemetric data, such as performance statistics and resource usage, in distributed systems using the Simple Network Management Protocol. The paper introduced the concept of intermediate level managers (ILM) that gather data from multiple nodes (similar to our collector nodes), and report this data to the top level manager (TLM) (which corresponds to our monitoring node). The TLM is also responsible for informing the ILMs of the type of data that needs to be collected.

Matthew et al. [17] propose a system called *Ganglia*. It is a generic solution for collecting any type of data from a distributed system, which arranges all nodes in the distributed system in a tree structure. Leaf nodes produce the data, and higher level nodes aggregate this data before sending it to parent nodes. A similar hierarchical model is provided in [18], where the system is divided into control agents, responsible for controlling and collecting data from the underlying system, supervisory agents, responsible for informing the control agents about the intended goal, and expert agents, which are used to handle exceptional situations such as malfunctions.

The Chukwa monitoring system [19] was developed as a mean of collecting and analyzing very large amounts of data (in the scale of hundreds of Gigabytes per day) produced by map-reduce calculations. Chukwa's nodes are organized in a tree structure. Adapters are deployed on the nodes which are being monitored to extract relevant data. In case the interest changes, the adapters need to be replaced. Agents run on the monitored system and use the adapters to collect data from the data source and send it to the collectors. Collectors receive data from multiple agents and write it to a big sink file along with the metadata. These files can later be read for analysis. Twitter has developed a similar unified logging infrastructure [20] that collects logs in a scalable and reliable manner using hierarchically connected log servers.

At the other spectrum are monitoring systems such as [21] that automatically trace events in the system, and cluster and order them so that they can determine dependencies (e.g., determining all execution related to a single external client request). These solutions often focus on the complex matching algorithms and scalability issues in log collection and processing are not the major concerns.

VII. Conclusion

In this paper we presented and evaluated a monitoring framework that fulfills the requirements for monitoring large-scale, location-based information systems. We propose to partition the space into regions and delegate the gathering of state updates for each region to a collector node. The collector node filters and aggregates the gathered game state updates according to the current point of interest of the observer, and forwards them to the monitoring node that merges and post-processes the data it receives.

We illustrated our monitoring framework in the context of MMOGs, and proposed a non-intrusive integration of the monitoring framework into the game by taking advantage of the replication model most multiplayer game engines and virtual worlds implement. By exploiting this replication model, collector nodes can observe all updates on the objects for which they are responsible without the need to put hooks into the underlying virtual world implementation. Replicated objects are also used for the communication between collectors and monitoring nodes by mapping the relevant game state updates to a fixed set of monitor objects. By carefully choosing the number of monitor objects on each collector and the frequency at which their state is updated, it is possible to achieve good accuracy and timeliness of data while bounding the maximum network bandwidth required for monitoring. To illustrate the filtering and data aggregation, we presented the design of monitor objects that gather player position data.

We summarized the results of an extensive set of performance measurements obtained by running several real-world experiments on our implementation. The experiments showed that 1) The delay introduced by using dedicated collector nodes to filter the game state updates is acceptable for real-time monitoring of MMOGs. 2) Our approach is capable of bounding the network bandwidth used by each collector node. 3) The proposed architecture scales well: the number of players that the system can handle is proportional to the number of collector nodes in the system. 4) The ideal $\#MOs/updateInterval$ ratio depends on the kind of data that is observed and the current point of interest. For instance, when observing player positions, the ratio resulting in the highest data accuracy depends on the zoom level used on the monitoring node. 5) Load balancing for collectors can be achieved by adjusting the size of the observed regions.

References

[1] J. Kienzle, C. Verbrugge, B. Kemme, A. Denault, and M. Hawker, "Mammoth: A Massively Multiplayer Game Research Framework," in *ICFDG*. New York, USA: ACM Press, April 2009, pp. 308 – 315.

[2] W. Cai, P. Xavier, S. Turner, and B. Lee, "A scalable architecture for supporting interactive games on the internet," in *Parallel and distributed simulation workshop*. IEEE, 2002, pp. 60–67.

[3] T.-Y. Hsiao and S.-M. Yuan, "Practical middleware for massively multiplayer online games," *IEEE Internet Computing*, vol. 9, no. 5, pp. 47–54, 2005.

[4] M. Varvello, S. Ferrari, E. Biersack, and C. Diot, "Exploring second life," *IEEE Transactions on Networking*, vol. 19, no. 1, pp. 80–91, 2011.

[5] R. Balan, M. Ebling, P. Castro, and A. Misra, "Matrix: Adaptive middleware for distributed multiplayer games," in *Middleware 2005*. Springer, 2005, pp. 390–400.

[6] B. Technologies, "Bigworld 2.0," http://www.bigworldtech.com/, 2011.

[7] B. Knutsson, H. Lu, W. Xu, and B. Hopkins, "Peer-to-peer support for massively multiplayer games," in *INFOCOM 2004*, vol. 1, 2004.

[8] T. Iimura, H. Hazeyama, and Y. Kadobayashi, "Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games," in *Netgames*. ACM, 2004, pp. 116–120.

[9] A. Bharambe, J. Pang, and S. Seshan, "Colyseus: A distributed architecture for online multiplayer games," *NSDI*, 2006.

[10] D. Ahmed, S. Shirmohammadi, and J. de Oliveira, "A hybrid P2P communications architecture for zonal MMOGs," *Multimedia Tools and Applications*, vol. 45, no. 1, pp. 313–345, 2009.

[11] L. Chan, J. Yong, J. Bai, B. Leong, and R. Tan, "Hydra: a massively-multiplayer peer-to-peer architecture for the game developer," in *Netgames*. ACM, 2007, pp. 37–42.

[12] A. Denault and J. Kienzle, "Journey: A massively multiplayer online game middleware," *IEEE Software*, vol. 28, no. 5, pp. 38–44, 2011.

[13] A. Denault, C. Cañas, J. Kienzle, and B. Kemme, "Triangle-based Obstacle-aware Load Balancing for Massively Multiplayer Games," in *Netgames*. ACM, 2011, pp. 1 – 6.

[14] J. Boulanger, J. Kienzle, and C. Verbrugge, "Comparing interest management algorithms for massively multiplayer games," in *Netgames*. ACM, 2006, pp. 1–12.

[15] L. Pantel and L. C. Wolf, "On the impact of delay on real-time multiplayer games," in *NOSSDAV*. ACM, 2002, pp. 23–29.

[16] R. Subramanyan, J. Miguel-Alonso, and J. A. B. Fortes, "A scalable snmp-based distibuted monitoring system for heterogeneous network computing," in *Supercomputing '00*. IEEE, 2000.

[17] M. L. Massie, B. N. Chun, and D. E. Culler, "The Ganglia distributed monitoring system: Design, implementation and experience," 2004.

[18] V. V. Tan, D.-S. Yoo, J.-C. Shin, and M.-J. Yi, "A multiagent system for hierarchical control and monitoring," *Journal of Universal Computer Science*, vol. 15, no. 13, pp. 2485–2505, 2009.

[19] A. Rabkin and R. Katz, "Chukwa: a system for reliable large-scale log collection," in *LISA'10*. USENIX Association, 2010, pp. 1–15.

[20] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy, "The unified logging infrastructure for data analytics at twitter," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1771–1780, Aug. 2012.

[21] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang, "vpath: Precise discovery of request processing paths from black-box observations of thread and network activities," in *USENIX'09*. USENIX Association, 2009, pp. 19–19.