An Architecture Supporting Large Scale MMOGs

Chandana Ghosh
Institute for Simulation & Training
3100 Technology Pkwy
Orlando, FL 32826
cghosh@ist.ucf.edu

R. Paul Wiegand
Institute for Simulation & Training
3100 Technology Pkwy
Orlando, FL 32826
wiegand@ist.ucf.edu

Brian Goldiez
Institute for Simulation & Training
3100 Technology Pkwy
Orlando, FL 32826
bgoldiez@ist.ucf.edu

Troy Dere
RDECOM-STTC
12423 Research Pkwy
Orlando, Fl 32826
troy.dere@us.army.mil

ABSTRACT

The growing popularity of large-scale, highly interactive virtual reality systems such as massively multiplayer online games (MMOGs) necessitates highly robust and efficient architectures. Distributed implementations are common, but they must deal with challenges such as supporting very large numbers of closely interacting users, the need to maintain robustness in the face of hardware failure, balancing the processing load, reducing user latency, and minimizing thrashing effects caused by movement between servers. Although a number of existing techniques address each of these independently, there are no unified methods that attack these problems cohesively.

We present methods to simultaneously address these critical challenges—a novel approach and associated software design intended for distributed high performance computing facilities in which the world is divided into a regular lattice of overlapping cells (providing redundancy), which are dynamically assigned to servers within the High Performance Computing (HPC) (facilitating load balancing). We believe this architecture can be applied to non-spatial cells. This architecture is currently being implemented in a test bed for further experimentation.

Categories and Subject Descriptors

I.6.8 [**Computing Methodologies**]: Simulation and Modeling—*types of simulation*

General Terms

Algorithms, Design, Performance

Keywords

Distributed Simulation, MMOG, Partitioning, Redundancy

1. INTRODUCTION

Large-scale, highly interactive virtual realities (VR) such as those popularized by massively multiplayer online games (MMOGs) permit growing numbers of interacting users to become immersed in a shared, virtual world. Beyond games meant purely for entertainment, expansion of these ideas into serious game technology [1] presents unprecedented training opportunities for the military [15] and emergency services communities. One example is the ability to conduct large-scale multi-team training, virtual training simulations of complex and real-world scenarios involving a host of different personnel working with different organizations with different specialized goals (e.g., paramedics, firemen, police, and military personnel in first response to some natural or manmade event). Finally, an infrastructure supporting large-scale interactions between heterogeneous users and equipment offers new opportunities and paradigms for social networking [19].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DISIO 2010, March 15, 2010, Torremolinos, Malaga, Spain.

Copyright 2010 ICST, ISBN 78-963-9799-87-5.

Advances n communication and computational infrastructures have led to increased sophistication of these digital virtual environments (DVEs) and interactions within them—complexity demanded by many applications such as the multi-team training example just mentioned. Additionally, increased scale of user interaction is of interest to entertainment [18] and commerce [3].

An expanded user base necessitates highly scalable and reliable software, as well as hardware solutions capable of handling smooth front-end and back-end execution. To achieve this, game and virtual reality systems rely more and more on distributed back-end solutions, where game and VR server responsibilities are divided among a number of interconnected machines. Distributed solutions introduce a number of difficulties, and four key requirements of large-scale, distributed VR systems are:

1. **Robustness to hardware failure** – Distributed systems should avoid single-point-of-failure problems where part of the world state is lost with a server failure.

2. **Balancing of processing load** – Dynamic entity behaviors demand responsive methods for shifting processor load to conserve computational resources.

3. **Mitigation of thrashing** – Distributed systems should avoid repeated inefficient transitions of objects between resources.

4. **Reduction of user latency** – Game systems must avoid unacceptable latencies between the user clients and the servers.

Currently these requirements are addressed separately by a number of technologies; however, our interest is in how these four can be achieved by a single, cohesive solution. Our current work has led to the development of a new software architecture intended for use with distributed high performance computing facilities. This architecture addresses these concerns by dividing the world into overlapping cells (providing redundancy and mitigating thrashing), which are dynamically assigned to servers within an HPC (facilitating load balancing). The design provides

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8731
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8731

seamlessness and continued connectivity. The HPC orientation of our design does not yet consider external network issues.

This paper introduces a preliminary study and design of this architecture. Our analysis considers the redundancy, load-balancing and thrash-mitigation effects of certain geometric variations in lattice structure, as well as how such variations impact dynamic cell allocation. User latency is a vital concern and addressed by our architecture; however, it is not the focus of this paper. Additionally, we describe a spatial division of the world, but our ideas can be extended to cases where cells are organizational or social in nature. We conclude with requirements for effective dynamic cell allocation algorithms and suggest some plausible simple solutions.

2. RELATED WORK

Distributed MMOGs and other multi-server, large-scale VR systems use a variety of methods to address the four desired capabilities mentioned above. Multi-server solutions must first address the question of how to divide server responsibilities, which typically involves dividing users or other world components among different machines. These choices lead to different advantages and disadvantages.

The method of sharding [18] maintains separate instances of the complete game world on different servers, where each server supports different sets of users (e.g., as in World of Warcraft [4]). This approach only partially addresses single-point-of-failure issues since a server failure entirely destroys the game experience for some users while leaving other users unaffected. Load balancing and latency reduction techniques can be addressed by effective allocation of users to servers. Users and objects do not move between servers, so there are no back-end thrashing effects caused by migrations between machines; however, this disjunction comes at the cost of severe constraints on the types and quantity of interactions that are possible.

Slightly more complex approaches are geographically constrained partitioning and zoning methods, where the game world is partitioned into geographically distinct cells, with each cell assigned to a separate server (e.g., as in Second Life [12]). Here, users can interact with any object or user, though movement between cells requires migration between servers. Migration raises several challenges, such as how to handle edge effects at cell boundaries and how to balance server processing loads and manage server-client latency problems. Simply relying on user dispersal to address these issues is infeasible given known traffic patterns in typical MMOGs [10, 5]. One solution is to use avatar generated workloads to manage load balancing [13], though balancing in this way means that serving requests from each user requires reading objects from all servers.

Zoning can be approached less severely by partitioning the world into small cells and assigning the cells to individual servers [2, 14]. The aim is to address load balancing by distributing equal number of users to the partitions while simultaneously minimizing inter-partition costs, though cell geometry and migration constraints differ among approaches. Such solutions lack seamless migration across cells; however, thrash mitigation can be approached by creating buffer regions at cell boundaries and defining (different) check-in and check-out edges to trigger shifts in server responsibilities [7, 8]. Still, without redundancy such systems are not fault tolerant.

Alternatively, publisher-subscriber methods [17] relay messages to users, and processing workload can be broken down into tasks, each of which can be scheduled on any server (e.g., as implemented within Project Darkstar [16]). Game objects can be stored in a fully transactional database for synchronization and consistency purposes. Unfortunately, this solution does not yet scale well in multi-server settings [6]. Another method to address scalability and single-point-of-failure is the well-known Peer-to-Peer (P2P) paradigm [11]. In this design, some server functions are distributed among clients enabling users to communicate with each other without going through a server. An example [9] puts forth a hybrid architecture that combines ideas from P2P and zoning for increased scalability. Limitations in authentication, security and persistence, make most P2P impractical solutions for large-scale MMOGs.

We strike a balance between these approaches. In our zone-based approach, users and objects can migrate throughout the DVE. Even so, seamlessness and fault tolerance are achieved via a natural redundancy mechanism enabled by overlapping cells.

3. AN OVERLAPPING, ZONE-BASED ARCHITECTURE

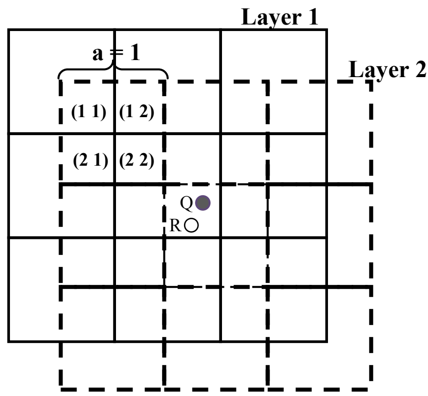
We divide the virtual world into a lattice of small, overlapping cells (zones)[1]. Cells can have different geometric shapes (e.g., square, hexagonal), and their pattern of overlap may differ to form lattices with different types of connectivity. Cells must completely tessellate the region of interest. The cell overlap scheme serves three important functions in our architecture: it provides a mechanism for redundancy to protect against server crash, it facilitates a system for seamless migration across cell boundaries, and it allows load balancing methods to control the balance between reads and writes of object data among the servers. Details follow.

Cells are pre-distributed amongst the servers for processing. Updating responsibilities for each *entity* (object and user data in the environment) are assigned to a cell based on the location of the object in the world, and the server managing that cell can be considered the *master host* of such objects. In addition, other cells will have redundant, read-only copies of the entity, and other servers managing such cells can be considered *slave hosts* for the entity. Thus each server can be considered both a master host for some entities and a slave host for other entities. There are redundant copies for all entities, but only a single server is permitted to write changes of the entity's state.

This master-slave relationship is determined by the overlap: cells may access local, cached slave-hosted copies of entities when the entity resides in a region of overlap with another cell. The system is not permitted to host all copies of a given entity on a single server, which entails certain regional constraints for server-cell assignment. These constraints differ based on cell geometry, and some will be discussed in detail in the next section. Figure 1 illustrates what we designate as a loose square cell overlap pattern.

[1] While zoning is currently spatially oriented, we anticipate extensions to non-spatial partitions, such as social or group hierarchies

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8731
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8731



☐ Cells assigned to Server1 – Server1 is master to object Q and slave to object R

⬚ Cells assigned to Server2 – Server2 is master to object R and slave to object Q

Figure 1. Loose Square Cell Overlap Pattern

The simplest rule for cell-to-server assignments is described as follows. In Figure 1, lattice constant a = 1, where, the lattice constant of any layer of cells is defined as the distance between the center of a cell and the center of its adjoining cell. Adjoined cells are those that share any common edge entirely. Each cell is divided into 4 quadrants, (11), (12), (21), (22). The cells in both layers are strictly maintained to be of exactly the same shape and dimension, while their spatial origin is set apart by a distance $\Delta x = ½ . a$ *and* $\Delta y = ½ . a$ with respect to each other, to generate the overlapping pattern. As can be seen in Fig. 1, any cell from a particular layer is overlapped partially by 4 cells from the overlapping layer, on its 4 quadrants respectively, thus, acquiring 100% of total overlap. If Server1 serves as the master to this cell, it acts as a slave to its 4 overlapping cells.

Update responsibilities of entities are transferred between servers as those entities are repositioned throughout the world. Such decisions are made by using the overlap region between cells to define check-in and check-out boundaries for entities moving between cells. We define an *Area of Interest* (AoI) around an entity, and when that AoI intersects the master cell's boundary, management of the entity is shifted. Figure 2 below illustrates this idea with square cells.

The value of the AoI radii of all objects can either differ or be the same. Experimentation will help determine optimum values and optimum ratios between cell dimensions and AoI radii. Seamless transition of objects between cells is made possible by using the check-in and check-out boundaries, which determine when a server should begin to act as the object's master. However, the velocity of the object plays a crucial role in providing smooth transition from one server to the other. This is due to the requirement that the data update needs to be quicker than the object velocity. If this is not true the radius of the AoI must be increased by some δ (Fig. 2 (b)) to accommodate such issues. The value of δ, and factors on which it depends, is another area for experimentation.

4. ANALYSIS

Early prototypes of this architecture provide some encouraging results. Additional analyses are found below.

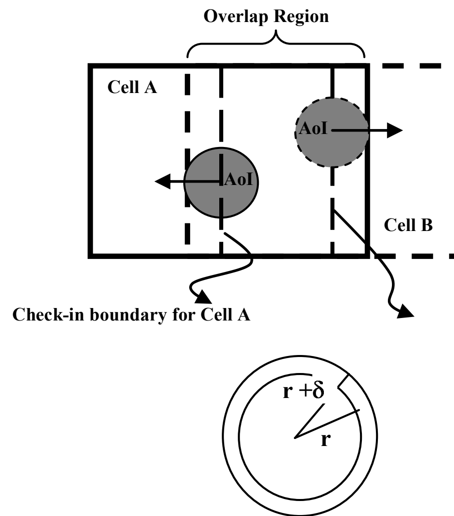


Figure 2. (a) Check-in and Check-out boundaries in overlapping cells, (b) Area of Interest with increment

4.1 Cell Graph Structures

We distinguish between cells that overlap and cells that neighbor one another. Overlapping cells share some region of coverage, while neighboring cells share an adjoining boundary. For example, the tiles on a checkerboard neighbor one another; however, if a new tile is placed on the point where four checkers tiles meet, the new tile would overlap the four checkers tiles.

The partitioning of the world is determined by cell geometry,

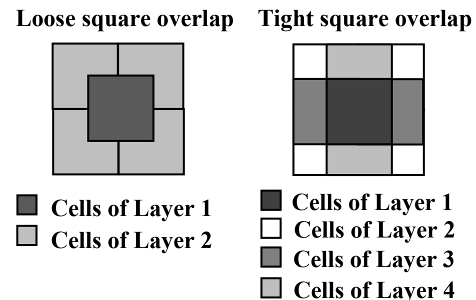


Figure 3. Example of loose and tight square overlap

cell size, and the overlap pattern of cells. We discuss only divisions of square cells for this paper, though the concepts generalize to other cell shapes (e.g., hexagons). Likewise, cells may be overlapped in many ways, but we focus on two choices: *loose overlap* and *tight overlap*. In Fig. 3, we see that for the loose overlap, any region of the game world is shared by 2 cells (shown by the hashed region), each belonging to a different layer and different server. Whereas, for the tight overlap, any region is shared by 4 cells each belonging to a different layer and different server.

It is useful to visualize these structures in graph form. There are two graphs: one for neighboring cells and one for overlapping cells. We are primarily concerned with the latter, in which each vertex in the graph represents a cell and edges between two vertices indicate where the two corresponding cells overlap. We can further refine this representation by including the server assignment information as the color of a vertex. We call such

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8731
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8731

graphs the cell allocation graph (CAG), Figure 4 below illustrates loose and tight overlapping CAG examples. The dotted squares show the portions of the CAG that correspond to their equivalent representation using cells in Figure 3, for the loose and tight overlaps respectively.

The ratio between the cell size and the size of the space

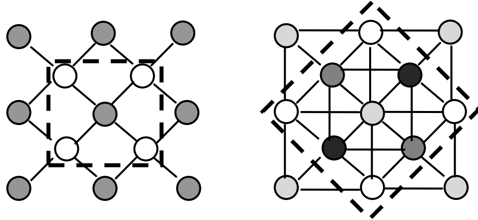


Figure 4. Cell Allocation Graphs for loose and tight square cell partitioning

determines the resolution with which the system is capable of load balancing, but it also impacts latency and redundancy. When object AoIs are too large with respect to the cell resolution, the radius of the AoI may span several cells, and the system will either have to make remote calls rather than rely on slave-hosted caches, or be configured with more sophisticated overlap strategies that permit greater read access to object information. When interaction regions are very small relative to cell size, then overlap and redundancy strategies will be less useful since objects within the cell will seldom interact with objects in other cells. In practical applications, the interaction region will vary between different objects and during execution. A good general lower bound for cell size can be obtained given an average size for the interaction region as follows. Consider again Figure 2, above and a hypothetical case where the diameter of the interaction region is precisely half the width of a cell and the object is centered in the overlap region. If the object moves slightly in the direction toward the overlapped cell, the check-out boundary will be triggered, and it will be migrated. If it then immediately backtracks, it will quickly trigger a migration back. Thus, it is best if cell widths are larger than half the diameter of the average AoI.

This design allows two ways for achieving load balancing. The first being load distribution from a congested cell, served by a server, to its overlapping cells, served by different servers. The second is by using the Dynamic Cell Allocation Algorithm (DCAA). Server load is measured as the number of dynamic entity processes being handled in a given period of time. However, load on a server can also be generated by other processes, such as creating, acquiring and deleting entity responsibilities. An appropriate metric to measure combined load needs to be devised.

4.2 Entity Motion

As previously mentioned, entity velocity is an important consideration trading off cell size and AoI. Below, a discussion of entity dynamics and thrash mitigation methods is provided for the simplest case of a loose square cell overlap pattern.

In the loose overlap each square cell in a layer is overlapped by four other cells in the overlapping layer. For convenience of description, let the type of cell filling indicate the server to which they have been assigned, such as in Figure 5, Cell A → server Gray (layer 1), cells B and E → server Stripes (layer 2) and cells C and D → server Dots (layer 2). The four quadrants of cell A are labeled as (11), (12), (21) and (22).

For an object, z, in quadrant (12), either server Gray or server

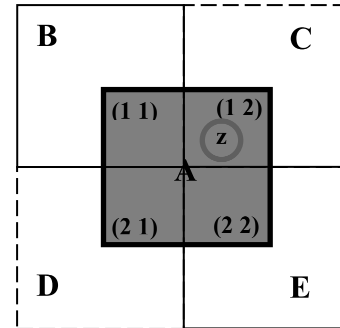


Figure 5. Quadrants in the loose square cell overlap pattern

Dots could be its master server. Figure 6 shows the different stages of the object motion as it moves from quadrant (12) to quadrant (11). For this scenario, it is preferable for the Gray server to act as the object's master. The four stages for the entity motion appear below in Figure 6, followed by discussion.

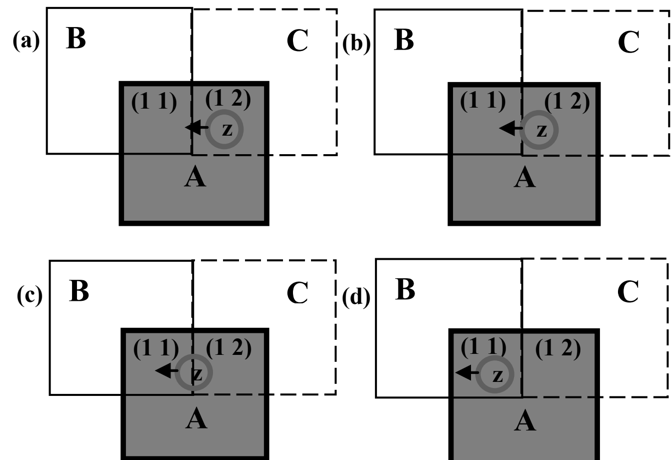


Figure 6. Motion of object across overlapping cells

In stage (a), assume Gray to be the master and Dots to be the slave server. Gray updates Dots on the movement of object z. In stage (b), the AoI of z touches the border of cell B and C, and slave copies of the object files have to be created in server Stripes. Gray has to now update Dots as well as Stripes. In stage (c), the AoI of z crosses the boundary of cell C, therefore Gray keeps updating Dots and Stripes. In stage (d), the object is entirely inside cell B, and Gray has to now update only Stripe.

The main contributing cost in this entire scenario, occurs in stages (b) and (c), where, Total Cost = Cost of creating slave copies of object in server Stripes + Cost of updating servers Dots and Stripes by server Gray. A hysteresis type of design could mitigate some of this cost and help minimize thrashing.

If the same scenario were to be considered, but with server Dots being the master in stage (a) and server Gray being the slave server, the contributing cost would again occur in stage (b),

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8731
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8731

where, Total Cost = Cost of creating master copies of the object in server Stripes + Cost of updating servers Gray and Dots by server Stripes. The only difference between these two possibilities is the handover of master status from Dots to Stripes in the second scenario as opposed to *no master status handover* by server Gray in the first scenario. These are likely insignificant in terms of Total Cost. However, once the object touches the boundary between cells B and C and chooses to oscillate across this boundary, in scenario 1 these oscillations will be free of thrashing since server Gray is the master and the object is entirely within cell A. Considering this situation, scenario 1 is the more favorable option.

The above description is for entity motion in the horizontal direction from quadrant (12) to quadrant (11). Extending this idea for objects in motion in all three directions, a simple set of rules can be formulated for the most favorable option that minimizes total cost. This is summarized in Table 1.

Table 1: Favorable Server-Motion allocation for Cost minimization

For direction of motion of object	Master server of object should be that server which is the master of the cell that holds the object in quadrant
→ ↓ ↘	(1 1)
← ↓ ↙	(1 2)
→ ↑ ↗	(2 1)
← ↑ ↖	(2 2)

Note that the above argument addresses object thrashing effects when in motion across boundaries that are formed by *edges of neighboring cells of the same layer*. However, when the object motion is diagonal across the intersection point between one horizontal and one vertical edge of the two different layers, oscillatory motion in this direction, at these points, cannot be free of thrashing unless any pair of the two neighboring cells of either layer, those that share an edge at this intersection, are assigned to the same server. There are 8 such cases that could occur in the case of the loose square overlapping cells, as in Figure 7. Similarly for the loose hexagonal overlap motif, a set of rules can be formulated and 6 cases exist for which thrashing is unavoidable unless the pair of neighboring cells in either layer hosting this movement are assigned to the same server.

4.3 Server Crash

A natural fault tolerance is accomplished in our design because of the redundancy provided by overlapping cells. When a server crashes, any entities over which it had master control are already replicated on at least one other server—the system must merely transfer master control and needn't seek data from a crashed machine. A preliminary analysis of this follows.

Figure 8 is a simple visualization of cells allocated to various servers and their mutual communication channels. Standard

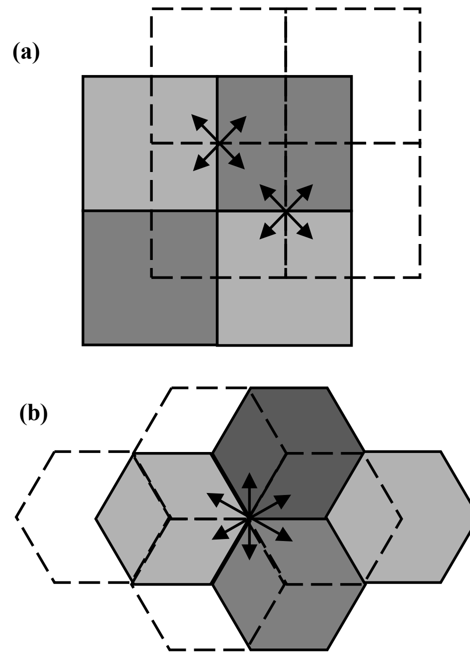


Figure 7. Object motion subject to thrashing for neighboring cells assigned to different servers in (a) a loose square overlap pattern and in (b) a loose hexagonal overlap pattern

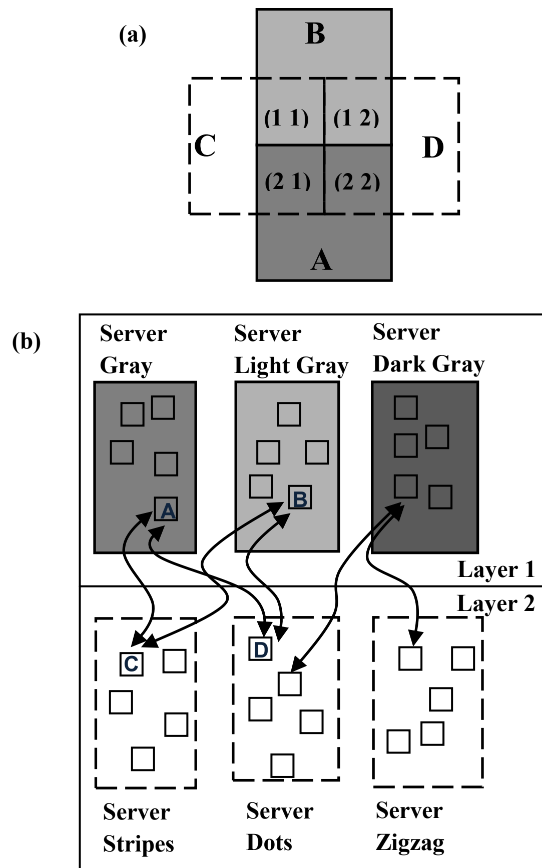


Figure 8. (a) Cell Overlap Pattern, (b) A Cell-to-Server allocation favorable for crash handling

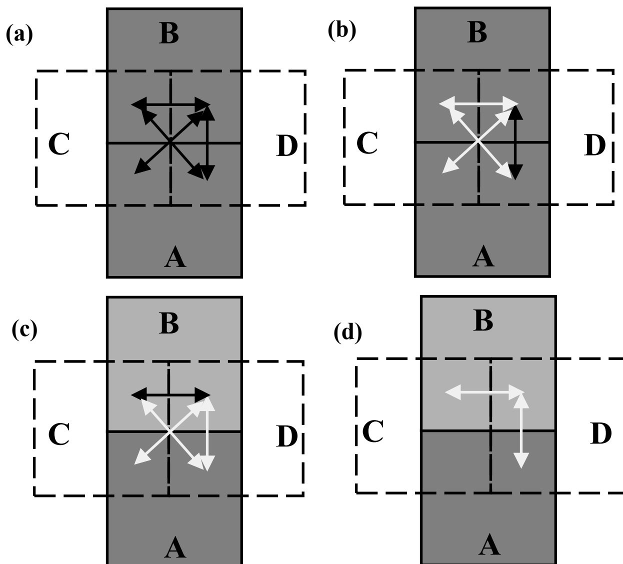
Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8731
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8731

message passing interfaces can handle communication between servers for transfer of object data permissions, update notifications, etc.

In the cell overlap pattern in Figure 8(a), cells C and D are being overlapped by cells A and B. Figure 8(b) shows the corresponding diagram of the server farm with cell allocations, which illustrates communication between the respective servers.

In the event that server Light Gray crashes, cell B (which overlaps cells C and D in quadrants (11) and (12)) gets stranded. Server Gray, which acts as master to cell A (and also overlaps cells C and D but in different quadrants (21) and (22)) could serve as master to cell B. Provided that server Gray has allowable load levels, this might be the most efficient assignment, considering that server Gray already has an open channel of communication between itself and servers Stripes and Dots (because Cell A overlaps cells C and D). Thus, in the above scenario, any server that meets the criteria of the cell allocation problem discussed in section 4.4 can be called to act as master to cell B. However, server Gray is preferable since it already is a master to cell A, which overlaps regions of cells C and D facilitating speedier object file transfers. Therefore, the cells should be assigned to servers in such a way as to meet the above criteria that neighboring cells should be preferably assigned to different servers.

For loose square overlap, there are just four fundamental motifs of cell-to-server assignments that exist between neighboring and overlapping cells, as shown in Figure 9.



Cells in Layer1 are in solid lines and cells in Layer2 are in dashed lines

Figure 9. (a) Neighboring cells in both layers assigned to same server, (b) Neighboring cells in Layer1 only assigned to same server, (c) Neighboring cells in Layer2 only assigned to same server, (d) Neighboring cells in both layers assigned to different servers

The inferences drawn from all four cases with respect to load balancing and thrashing are tabulated in Table 2 and Table 3. The tables clearly show that a reasonable trade-off has to be

Table 2: Favorable Server allocation for Thrash mitigation

Cell to server assignment	Directions of object motion			
	Free of thrashing when		Subject to thrashing when	
	A Layer1 server is the master	A Layer2 server is the master	A Layer1 server is the master	A Layer2 server is the master
9(a)	↕ ↔ ⤡ ⤢	↕ ↔ ⤡ ⤢	-	-
9(b)	↕ ↔ ⤡ ⤢	↕	-	-
9(c)	↔	↕ ↔ ⤡ ⤢	-	-
9(d)	↔	↕	⤡ ⤢	⤡ ⤢

made between these two factors of thrashing and load balancing while designing the Cell Allocation Graphs. These decisions can be made based on the specific game engines.

4.4 Dynamic Allocation

Dynamic cell allocation methods between servers must accomplish a multitude of things with reasonable efficiency. They must accommodate our restriction that the system cannot host all copies of a given entity on a single server. Additionally, allocations should provide some level of load balancing among servers. Finally, there may be ways to mitigate some types of thrashing effects, as discussed in the section above, while designing the Dynamic Cell Allocation Algorithm (DCAA). These must be done appropriately given different problem properties, including the cell graph structure factors discussed above, and issues such as the behavior and interaction of objects within the system. Considering also the heterogeneity of host servers, the algorithm has to account for variable server capacities and dynamic communication patterns within the network. These factors impact both latency and load.

Dynamic cell allocation is a type of constrained multi-object optimization problem. Redundancy requirements establish primary constraints, and load balancing, latency reduction, and thrash-mitigation objectives that must be optimized.

There is no single allocation method that will suit all situations, but some general observations can be made regarding the tradeoffs between cell structures that are fundamental to designing an effective partitioning scheme.

Using tight overlap carries increased communication burden and (perhaps) unnecessarily excessive redundancy, but they provide

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8731
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8731

less severe constraints for the allocation algorithm. Loose, square overlap structures provide the minimum replication of all objects in the environment and thus reduce the communication costs of the redundancy since fewer copies are exchanged between servers, but finding allocations that provide redundancy is algorithmically more difficult. We use a bit of graph theory to help explain why this is so.

Consider the cell allocation graph for a square-cell, loose overlap partitioning. Dynamic allocation methods must assign each cell to a server, and the constraint that permits complete redundancy implies that no two overlapping cells be on the same server. This is equivalent to a graph coloring problem, which is NP-hard in the general case for any graph. Cell allocation graphs are a subset of possible graphs, and an example is sufficient to show that these particular graphs admit efficient assignment solutions: visualize the graphs as two interleaved levels, divide the server list into two mutually exclusive lists, and assign servers from one group to one level and the remaining servers to the other level. In this way, connected nodes cannot be served by the same server and the smallest set of servers that can admit such a solution is known (two, as in the left graph of Figure 4).

Table 3: Load distribution in the event of a server crash

Cell to Server Assignment	√, *n* – Functional Server w/ Max. Load of *n* Quadrants × – Crashed Server NA – Not Applicable			
	Gray	**Light Gray**	**Stripes**	**Dots**
9(a)	√, 4	NA	×	NA
	×	NA	√, 4	NA
9(b)	√, 2	NA	×	√
	√, 2	NA	√	×
	×	NA	√, 2	√, 2
9(c)	√, 2	√, 2	×	NA
	×	√	√, 2	NA
	√	×	√, 2	NA
9(d)	√	×	√, 1	√, 1
	×	√	√, 1	√, 1
	√, 1	√, 1	×	√
	√, 1	√, 1	√	×

Though simple algorithmic solutions that satisfy the redundancy constraints are possible when using a loose overlap, to optimize for other factors (e.g., load balance), the allocation algorithm must search within a quite sparse subset of graphs with valid assignments (no overlapping cells on the same server). This implies that the larger multi-objective optimization problem may be difficult when the server-cell ratio is small.

When using tight overlap, the redundancy requirement creates a less severe mandate: It is merely necessary that an object be replicated on at least one other server. Server-cell allocation in this context is no longer a traditional graph coloring problem, but rather a form of weak graph coloring [22]. Solving this optimization problem means the definition of a *valid* assignment is related to the notion of a k-partially proper coloring: a t-coloring of a graph $G = \langle V, E \rangle$, where t is the number of colors (servers) that must be assigned to the vertices (cells), is k-partially proper if every vertex $v \in V$ has at least $\min\{\deg(v), k\}$ neighbors with different colors, where $\deg(v)$ denotes the degree of v. In the case of graph coloring (and loose overlap cell assignment), k is the maximum degree of any vertex in the graph. In the case of tight overlap, any combination of north and south, or east and west overlapping cells will completely cover a given cell. This means a subset of 3- and 2-partially proper colorings admit valid cell assignments. Additionally, the four diagonal overlapping cells completely cover a cell, thus an even larger subset of 5- and 4-partially proper colorings admit valid assignments. If there are six or more overlapping cells with a different assignment than the given cell, then the redundancy constraint must be met since at least two must be the north/south or east/west pair. This means that all k-partially proper colorings with $k > 5$ represent valid cell assignments. The redundancy requirement for tight overlap is a substantial relaxation of the coloring constraint from the loose overlap case.

In fact, the space of server-cell assignments that confirms full redundancy when using tight overlap is much larger than it is for the loose overlap case, perhaps easing optimization. We will employ a combination of formal and empirical analysis to help develop guidelines for algorithm designers for how to select cell geometry given different server and problem configurations.

5. CONCLUSIONS & CURRENT STATUS

Our goal is to address, using distributed computing, solutions for large-scale, highly interactive VR environments that provide four critical capabilities (redundancy, load balancing, thrash mitigation, and latency reduction) in a cohesive, unified manner. In this paper, we present a promising new architecture that combines ideas from existing methods that independently address these issues. A prototype of this system currently serves as a test bed to explore our ideas for finding unified answers to these complex questions.

In this paper, we discuss some of our analyses of the architecture. From this we know that cell resolution and overlap are the critical representational research points, and that dynamic cell allocation can be seen as a constrained multi-objective optimization problem, where the primary constraint is related to a generalization of the graph coloring problem. Experiments involving various cell graph structures, allocation algorithms, and simulations of varying complexity of interaction patterns are currently under development within this test bed.

Already, we can say the following. When the server-cell ratio is relatively high, a loose overlap cell structure will often be a better choice. Resolving the redundancy constraints will be less challenging and the dynamic cell allocation method can concentrate on quick, approximate solutions that minimize latency, load imbalance, and thrashing. Often simple divisions will be sufficient: the server list is divided in half and the two "layers" of graph are treated separately and within-layer

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8731
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8731

assignments that keep local groups of neighboring cells on the same server. In such cases, balancing can take the form of resizing these local clusters relative to one another, and when the surface boundaries of those local groups is as small as possible, the number of objects migrated as a result of re-balancing is minimized. When the server-cell ratio is very small, though, more thought into cell structure and allocation is needed, and the complete multi-objective optimization problem may be impractical. Tight overlap in cells may be needed to meet redundancy constraints, and load balancing events may need to be reserved for extreme cases of imbalance. More attention should be spent, then, on redundancy and latency.

Currently, we are implementing this architecture in software hosted on a 600+ core system. Once prototyping is complete we will be attaching the front end of a simple game/virtual environment and conducting experimentation aimed at assessing and enhancing runtime performance; determining trade-offs on cell size, entity performance, and AoI; and experimenting with the DCAA. We are developing example simulations within the Java-based multiagent simulation framework MASON [20].

6. ACKNOWLEDGEMENTS

This work was funded a part of the Army RDECOM STTC project *High Performance Computing for Simulation Training Systems* (N61339-07-C-0107).

7. REFERENCES

[1] C. Abt. Serious Games. Viking Press, New York, 1970.

[2] D. Ahmed and S. Shirmohammadi. A microcell oriented load balancing model for collaborative virtual environments. In Proceedings from the 2008 Virtual Environments, Human-Computer Interfaces and Measurement Systems, pages 86–91, 2008.

[3] R. Arakji and K. Lang. Avatar business value analysis: A method for the evaluation of business value creation in virtual commerce. Journal of Electronic Commerce, 9(3):207–218, 2008.

[4] Blizzard Entertainment. World of warcraft.http://www.worldofwarcraft.com.

[5] K.-T. Chen, P. Huang, and C.-L. Lei. Game traffic analysis:an MMORPG perspective. In Proceedings of the InternationalWorkshop on Network and Operating Systems Support for Digital Audio and Video, pages 19–24, 2005.

[6] Darkstar discussion forum. Scalability and performance of multimode PDS. http://www.projectdarkstar.com/forum/ ?topic=829.msg5678#msg5678, September 2009.

[7] J. de Oliveira and N. Georganas. Velvet: An adaptive hybrid architecture for very large virtual environments. Presence: Teleoperators and Virtual Environments, 12(6):555–580, 2003.

[8] F. Glinka, A. Ploss, J. Mueller-Iden, and S. Gorlatch. Rtf: A realtime framework for developing scalable multiplayer online games. In Proceedings of the 6th ACM SIGCOMM Workshop on Network and System Support for Games, pages 81–86, 2007.

[9] I. Kazem, D. T. Ahmed, and S. Shirmohammadi. A zone based architecture for massively multi-user simulations. In Proceedings of the 2007 Spring Simulation Multiconference, pages 149–156, 2007.

[10] J. Kim, J. Choi, D. Chang, T. Kwon, Y. Choi, and E. Yuk. Traffic characteristics of a massively multi-player online role playing game. In Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games, pages 1–8, 2005.

[11] K.-C. Kim, I. Yeom, and J. Lee. Hyms: A hybrid mmog server architecture. IEICE Trans. on Information Systems, (12):2706–2713, 2004.

[12] Linden Lab. Second life. http://secondlife.com/.

[13] P. Morillo, J. Orduna, M. Fernandez, and J. Duato. Improving the performance of distributed virtual environment systems. IEEE Transactions on Parallel and Distributed Systems, 16(7):637–649, 2005.

[14] B. Ng, A. Si, R. Lau, and F. Li. A multi-server architecture for distributed virtual walkthrough. In Proceedings of the ACM Symposium on Virtual Reality Software and Technology, pages 163–170, 2002.

[15] M. Proctor, M. Bauer, and T. Lucario. Helicopter flight training through serious aviation gaming. JDMS: Applications, Methodology, Technology, 4(5), 2007.

[16] Sun Microsystems Laboratories. Project Darkstar**.** http://www.projectdarkstar.com/

[17] J. Waldo. Scaling in games & virtual worlds. Queue, 6(7):10–16, 2008.

[18] S. Webb,W. Lau, and S. Soh. NGS: an application layer network game simulator. In IE '06: Proceedings of the 3rd Australasian conference on Interactive entertainment, pages 15–22, Murdoch Univ., Australia, 2006. Murdoch Univ.

[19] M. Yee. The demographics, motivations, and derived experiences of users of massively multi-user online graphical environments. Presence: Teleoperators and Virtual Environments, 15(3):309–329, 2006.

[20] S. Luke, G.C. Balan, L.A. Panait, C. Cioffi-Revilla, and S. Paus. MASON: A Java multi-agent simulation.

[21] Library. In *Proceedings of Agent 2003 Conference on Challenges in Social Simulation*.

[22] F. Kuhn. Weak graph colorings: distributed algorithms and applications. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures,* pp. 138-144, 2009.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8731
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8731