JISA manuscript No.
(will be inserted by the editor)

Unifying Divergence Bounding and Locality Awareness in Replicated Systems with Vector-Field Consistency

Luís Veiga · André Negrão · Nuno Santos · Paulo Ferreira

Abstract Data replication is a very relevant technique for improving performance, availability and scalability. These are requirements of many applications such as multiplayer distributed games, cooperative software tools, etc. However, consistency of the replicated shared state is hard to ensure. Current consistency models and middleware systems lack the required adaptability and efficiency. Thus, developing such robust applications is still a daunting task.

We propose a new consistency model, named Vector-Field Consistency (VFC), that unifies i) several forms of consistency enforcement and a multi-dimensional criteria (time, sequence and value) to limit replica divergence, with ii) techniques based on locality-awareness (w.r.t. players position).

Based on the VFC model, we propose a generic meta-architecture that can be easily instantiated both to centralized and (dynamically) partitioned architectures: i) a single central server in which the VFC algorithm runs, or b) a set of servers in which each one is responsible for a slice of the data being shared. The first approach is clearly more adapted to ad-hoc networks of resource-constrained devices while the second, being more scalable, is well adapted to large-scale networks. We developed and evaluated two prototypes of VFC (for ad-hoc and large-scale networks) with very good performance results.

Keywords Consistency Management · Replicated Objects · Locality-Awareness · Multiplayer Games

1 Introduction

Replication is a fundamental technique employed in distributed systems to improve reliability, scalability, performance and to support disconnected operation. Replication has been widely applied in different contexts such as cooperative work, replicated databases and services, and more recently, distributed multiplayer games, both in ad-hoc, mobile and large-scale networks. However, replication also entails the issue of maintaining consistency among the replicas of an object. Consistency has been addressed both with traditional pessimistic (lock-based) as well and optimistic approaches [27].

Optimistic approaches essentially trade increased availability, reduced latency and bandwidth usage, in exchange for some discrepancy or divergence among replicas, albeit only temporarily. Hence, applications may allow data inconsistencies up to a certain limit defined by application programmers according to the semantics of applications. The criteria for relaxing consistency vary: by divergence between the values of replicas,on a time-basis [1], by applying application based predicates on replica values [15, 30], sequential ordering [14], or with hybrid approaches [30, 31]. Nonetheless, these proposals are inadequate to cope with the dynamics of distributed multiplayer games: consistency requirements change often and quickly throughout the game execution, namely w.r.t. the players' position in the *virtual world*. On the other hand, current middleware for multiplayer games embodies the notion of *locality-awareness* (traceable to [28,20]) but employs very limited consistency models [3] (e.g., strict consistency in part of the game world and none on the rest of it), or use it just to drive load-balancing [8] and network traffic between servers [11].

Current distributed multiplayer games support large numbers of players and maintain large virtual worlds (e.g., Everquest, World of Warcraft). Employing a network of multiple servers is a common approach to improve the performance of commercial games, with the game state (e.g. player positions, maps, scores) being shared among the network nodes. Enforcing data consistency requires additional communication for update propagation and synchronization operations. To mitigate negative impacts on latency and bandwidth, and to ensure scalability, two classes of approaches are followed, frequently combined. First, a game's virtual world is either duplicated or statically partitioned into several mini-worlds [23], each assigned to a different, independent server. Thus, users are confined to a single server at each moment and are unable to interact with players handled by other servers. Partitioned schemes do allow users to move to other partitions, but force them to cross some form of artificial boundary (e.g., portals, tunnels) specially designed for that purpose. Scalability is, therefore, achieved at the cost of interactivity and, possibly, designing freedom. Second, programmers tend to use programming tweaks, low level optimizations and error-

L. Veiga, A. Negrão, N.Santos, P. Ferreira
INESC ID - Rua Alves Redol, 9
1000 Lisboa - Portugal
E-mail: luis.veiga@inesc-id.pt

N.Santos
MPI-SWS
66123 Saarbruecken
Germany

prone message-passing approaches to keep the shared data consistent. As a side effect, software becomes harder to manage and less reliable.

In this paper, we propose and evaluate a new consistency model for replicated objects called *Vector-Field Consistency* (VFC) that unifies optimistic consistency (divergence bounding) with locality-awareness techniques. It allows players to maintain a global vision of the complete game world, without resorting to artificial boundaries (such as portals), while ensuring interactivity through graceful (and bounded) degradation of data consistency. Metaphorically, it operates as a dolphin or submarine sonar using higher frequencies for increased accuracy limited to short ranges, and lower frequencies for wider range scans but with lower precision.

VFC selectively and dynamically strengthens/weakens replica consistency based on the ongoing game state while elegantly managing i) how the consistency degree *changes* throughout game execution w.r.t. each player, and ii) how the consistency requirements are *specified*. First, by employing locality-awareness techniques, VFC considers that throughout the game execution, there are certain 'observation points' we call *pivots* (e.g. the player's position) around which the consistency is required to be strong and weakens as the distance from the pivot increases. Since pivots can change with time (e.g. if the player moves), objects' consistency needs can also change with time. Second, it provides a 3-dimensional vector for specifying consistency degrees, where each dimension bounds the replica divergence in *time* (delay), *sequence* (number of operations) and *value* (magnitude of modifications) constraints. Game programmers (or even game designers) can parameterize VFC by specifying both the pivots and the consistency degrees according to game logic.

The advantages of VFC are manyfold. First, it is flexible and easily perceived by programmers: pivots and consistency degrees are intuitive settings allowing game programmers to specify consistency requirements for a wide range of game scenarios. Second, VFC allows user experience to proceed within acceptable parameters in the sense that, to the players, game rules are being abided to, and users are provided with complete and relevant information (e.g., immediate surroundings, opponents' scores) to make sensible game decisions. Also, by intelligently selecting the critical updates to send and postponing the less critical ones, VFC is efficient in the utilization of resources, it reduces network bandwidth usage and masquerades latency. Thus, for each particular game, programmers are able to specify the consistency requirements that enable a more efficient use of the network by tolerating bounded inconsistencies that do not jeopardize the overall game state and the players' experience.

This paper is organized as follows. Section 2 describes the VFC consistency model. Section 3 presents a meta-architecture to enforce the VFC model and its instantiation in two architectures: centralized and distributed with partitioning. Section 4 describes the main implementation details of two prototypes (for ad-hoc and large-scale networks). Section 5 presents and discusses the experimental results obtained. Section 6 surveys the relevant related work, and Section 7 closes with some conclusions.

2 Consistency Model

In VFC, objects are positioned within a *virtual world*, an abstraction of an N-dimensional space. Without loss of generality, we consider the virtual world to be 2-dimensional. In many games these abstractions map immediately to the game semantics; for example, in the Pac-Man game, the virtual world is a 2-dimensional maze populated with objects such as avatars, ghosts and dots. Each node of the network has a local *view* consisting of a full local replica of the virtual world. Each view may have bounded inconsistencies. VFC characterizes how these inconsistencies are managed.

Broadly, VFC offers bounded divergence guarantees, stronger than eventual consistency but weaker than strict consistency. In eventual consistency, delays in update propagation are essentially unbounded, uncommitted updates may be subject to reordering in logs, and replicas may experience oscillation in values. In VFC, updates are state-transfers, the latest update received completely precludes or supersedes previous ones, even if some have been omitted. Updates are serialized at servers and propagated to clients. Updates received by clients are never applied out of order; therefore replicas never diverge (they may lag) and players never experience game going "backwards in time". Strict consistency requires global synchronism or pessimistic locking regarding data reads with severe performance and scalability penalties. VFC is by design an optimistic approach, while ensuring that lasting or unbounded divergence, among data replicas, never takes place.

The remainder of this section describes the two main ideas underlying the VFC model: *consistency zones* describe how the consistency of object replicas varies in each view (see Section 2.1), and *consistency vectors* characterize the consistency degrees (see Section 2.2). Section 2.3, proposes two generalizations of the basic VFC model and systematizes the parameters for setting VFC from the game programmers' viewpoint.

2.1 Field-Generated Consistency Zones

Within a particular view, object consistency depends on its distance to a *pivot* (P). The pivot characterized by a position in the virtual world and it can move over time. A pivot can be an object (e.g. the Pac-Man player) or just a function (e.g. an editor cursor). Figure 1.a illustrates a virtual world populated with objects o_1, o_2, o_3, o_4 and o_5. The pivot (o_5) is highlighted with a star.

By analogy with the electric ($\overrightarrow{E}$) and the gravitational ($\overrightarrow{G}$) fields, a pivot generates a 'consistency field' determining the consistency of each object as a function of the

a. Conceptual consistency zones. **b.** Simplified consistency zones.

Fig. 1 Consistency zones centered on a pivot within a virtual world.

a. Node A view. **b.** Node B view.

Fig. 2 Two views of the same virtual world.

distance between the object and the pivot. Thus, pivots generate *consistency zones*, iso-surfaces, ring shaped, concentric areas around them, such that the objects positioned within the same consistency zone are enforced the same consistency degree. For example, in Figure 1.a, pivot P is in the center of four consistency zones labeled z_i, where $0 \leq i \leq 4$. Objects o_2 and o_3 are enforced the same consistency degree since they are in zone z_3.

Each consistency zone maps to a *consistency degree* (c_i) of a *consistency scale*. A consistency scale $C = \langle c_1, \ldots, c_n \rangle$ is an ordered set of c_i, each specifying the consistency to be enforced within zone z_i. The property $c_i > c_{i+1}$ holds, meaning that c_i enforces stronger consistency than c_{i+1}. Thus, consistency zones are arranged monotonically; consistency degrees become weaker as the distance to P increases. In Figure 1.a, darker consistency zones impose stronger consistency requirements. For example, if P represents the player and the other objects are ghosts of the Pac-Man game, each ghost's consistency weakens as it is farther from the player. Specification of consistency degrees is detailed in Section 2.2.

Consider λ_i the radius of the outer circumference of z_i. We define z_i as follows: i) if $i = 1$ then z_1 is the circle of radius λ_1, ii) if $i > 1$ then z_i refers to the area enclosed between z_i and z_{i-1} (a ring). Thus, if a pivot P is surrounded by n consistency zones, it is necessary and sufficient to specify λ_i to all i where $1 \leq i < n$. The consistency zone z_n refers to the area beyond the circumference of radius λ_{n-1}. This is represented by vector $Z = [\lambda_1, ..., \lambda_{n-1}]$. Since it is computationally more expensive to determine if an object is within a radial surface, we define consistency zones as concentric squares instead of concentric circles, as depicted in Figure 1.b. Also, λ represents not the radius of the outer circumference, but half the side of the outer square (or its apothem). For example, consistency zones of Figure 1.b are defined by $Z = [1, 2, 3]$ and objects are distributed by the following zones: $\{o_1, o_5\} \rightarrow z_1$, $\{o_2, o_3\} \rightarrow z_2$, $\{o_4\} \rightarrow z_3$.

Determining the consistency degree of an object depends on its relative position w.r.t. the pivots. Thus, the same object may have different consistency degrees in different views. Figure 2 illustrates this by depicting the views of two nodes, A (Figure 2.a) and B (Figure 2.b), respectively, with pivots P_A and P_B. Both pivots generate the consistency zone pattern $Z = [1, 2, 3]$. Hence, for example, $o_2 \rightarrow z_2$, in A, while $o_2 \rightarrow z_4$ in B. This implies that o_2 consistency is stronger in A than in B, which is expected since o_2 is closest to a pivot in A.

In a game with two (or more) pivots, the definition of the consistency zones and corresponding consistency degrees is obviously a relevant issue that depends on the game semantics. For example, when two pivots, each for a different player, have their outer consistency zones with very small intersection, this may lead each player to observe slight intermittence (e.g., interaction or battles among non-player entities), although no differences in outcome. Depending on their relative distances and movement, each player may observe such interactions (individual actions)

with different lag/delay, with different detail (e.g., number of shots), but without lasting divergence and, therefore, no significant impact in player's decisions and game semantics. This can be prevented with a consistency zone definition (and consistency degrees) that takes into account the game semantics.

2.2 Consistency Degree Vectors

VFC describes the consistency degrees as 3-dimensional *consistency vectors* $\kappa = [\theta, \sigma, \nu]$. κ bounds the maximum objects divergence in a particular view, i.e. between the objects latest updates and their replicas in that view. In short, for each object o, κ bounds the staleness of o in a particular view. Each dimension is a numerical scalar defining the maximum divergence of the orthogonal constraints *time* (θ), *sequence* (σ), and *value* (ν)[1], respectively.

- *Time* – Specifies the maximum time a replica can be without being refreshed with its latest value, irrespective of the number of updates performed in-between. Consider that $\theta(o)$ provides the time passed from the last replica update. The *time* constraint κ_θ enforces that, at any time, $\theta(o) < \kappa_\theta$. This scalar (not necessarily integer) quantity measures time in seconds.
- *Sequence* – Specifies the maximum number of lost replica updates, i.e. updates that were not applied to a replica. Similarly, consider that $\sigma(o)$ indicates the number of lost updates. The sequence constraint κ_σ enforces that, at any time, $\sigma(o) < \kappa_\sigma$. The unit is the number of lost updates.
- *Value* – Specifies the maximum relative difference between replica contents or against a constant (e.g. top-value). Consider that $\nu(o)$ provides this difference. The value constraint κ_ν enforces that, at any time, $\nu(o) < \kappa_\nu$. The unit of variation is a percentage. It captures the effects of updates (i.e., their impact or importance) on the object's internal state and is implementation dependent (e.g. it may reflect a drift or discrepancy regarding the player's score or the player's life charge/energy).

The overall maximum divergence is obtained by the disjunction of all the κ vector dimensions. For example, consider the consistency vector $\kappa = [0.1, 6, 20]$. Hence, at maximum, replicas are outdated in $\kappa_\theta = 0.1$ seconds or $\kappa_\sigma = 6$ lost updates or with a $\kappa_\nu = 20\%$ variation in the replica internal state. To indicate the least possible requirements, i.e. no requirements on that dimension, we use '.' (mathematically, this symbol represents '∞'). For example, $\kappa = [0.1, 6, .]$ imposes no consistency constraints whatsoever regarding the replica internal state.

In VFC, consistency degrees are specified by κ vectors. In order to specify a consistency scale obeying $c_i > c_{i+1}$ with κ_i and κ_{i+1} vectors, the condition $\kappa_{i+1} > \kappa_i$ must hold, i.e. for every $\kappa_{i+1_u} \geq \kappa_{i_u}$ and there is at least one v such that $\kappa_{i+1_v} > \kappa_{i_v}$, $u, v \in \{\theta, \sigma, \nu\}$. For example, $C = \langle [0.2, 2, 10], [0.2, 5, 10] \rangle$ is a valid consistency scale: $[0.2, 2, 10]$ stands for a stronger consistency degree than $[0.2, 5, 10]$ because the number of admitted lost updates is higher in the latter (5) than in the former (2) and the other dimensions are equal. Also, we define $\kappa_M = [., ., .]$ as the highest consistency degree, and $\kappa_m = [0, 0, 0]$ as the lowest consistency degree, such that $\kappa_m \leq \kappa_i \leq \kappa_M$.

[1] Although in modern Greek, the *vee* sound is written using the letter β, we prefer to use the letter ν, for its resemblance with the latin v.

Parameter	Description
O_i	Subset of objects that the consistency specification refers to. O_i are exclusive meaning that for every two ϕ_i and ϕ_j of ϕ, if $o \in O_i \Rightarrow o \notin O_j$. Moreover, for every object o, there must be a ϕ_i such that $o \in O_i$.
Z	Consistency zone vector Z specifying how to draw the consistency zones around the pivots. It is $\#_Z$ sized and specifies $\#_Z + 1$ consistency zones.
C	Consistency scale characterizing the consistency degrees for applying into the consistency zones. It is $\#_C$ sized with $\#_C = \#_Z + 1$ consistency degrees.
V	Set identifying the pivot objects for each view of the virtual world.

Table 1 Table describing the ϕ parameters of VFC.

2.3 VFC Generalization

In this section we introduce two generalizations allowing a broader utilization of the VFC model: *multi-pivot* and *multi-zones* generalizations. The multi-pivot generalization admits more than one pivot per view. Figure 3 illustrates such a case, with two pivots P_1 and P_2 in the same view. Objects are assigned the consistency degree w.r.t. the closest pivot.

In other words, VFC supports game semantics usually employed in some *role-playing games* and *real-time strategy* games, where a (human) player may have multiple avatars by controlling several game entities (e.g., characters, soldiers), turning their locations into important points in the virtual world; each one would be a pivot around which stronger consistency may be required (when compared to other areas of the virtual world).

The multi-zones generalization allows different sets of objects to be characterized differently w.r.t. their consistency requirements. For example, in Pac-Man, objects standing for ghosts and for rooms may be characterized with different consistency requirements. Thus, n sets of objects may be assigned specifically: i) consistency zones, ii) consistency degrees, and iii) pivots. Specification of each set is designated by ϕ_i, where $1 \leq i \leq n$; ϕ refers to all ϕ_i. Figure 4 shows an example of two object set specific settings ϕ_1 and ϕ_2. The former characterizes objects $\{o_1, o_2, o_4, o_5\}$. The latter characterizes objects $\{o_3, o_6, o_7, o_8\}$. Both have the same pivot but different consistency zone specifications.

Summary. In order to specify the consistency requirements, game programmers need to provide the VFC ϕ settings by describing individual object sets ϕ_i. Each ϕ_i

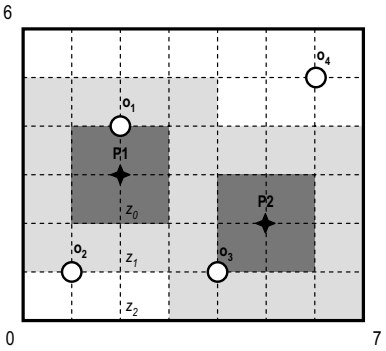


Fig. 3 Multi-pivot generalization.

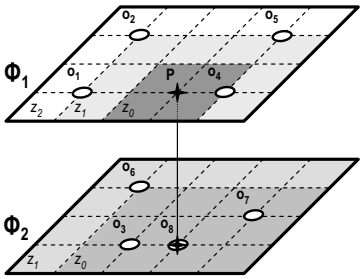


Fig. 4 Multi-zones generalization.

setting is described by $\phi_i = [O_i, Z, C, V]$, where $O_i \subseteq O$. Table 1 summarizes these parameters. As an example, the ϕ settings relative to Figure 2 can be described by $\phi_1 = [O, Z, C, P]$, where $O = \{o_1, o_2, o_3, o_4, o_5, o_6\}$, $Z = [1, 2, 3]$, $C = \langle \kappa_m, [., 1, .], [., 2, .], \kappa_M \rangle$ and, finally, $V = \{A \rightarrow \{o_6\}, B \rightarrow \{o_5\}\}$. In this example, there is a single object set ϕ_1.

3 Architecture

VFC is aimed at supporting the design of multiplayer distributed games both for ad-hoc and for large-scale networks. Based on the model previously described, we present a generic meta-architecture that can be instantiated in these two scenarios according to their specific characteristics. Both specific architectures enforce VFC by managing the game state between the network nodes and provides programmers with the adequate means to parameterize VFC according to game semantics.

3.1 Generic VFC Meta-Architecture

The VFC model is general enough to contemplate a wide set of architectural solutions. Thus, we provide a generic VFC meta-architecture (see Figure 5) that allows several instantiations; in particular, and without loss of generality, we focus on both centralized and (dynamically) partitioned architectures: i) a single central server in which the VFC algorithm runs, or b) a set of servers in which each one is responsible for a slice of the data being shared (e.g. a part of the game scenario). The first approach is clearly more adapted to ad-hoc networks of resource-constrained devices while the second, being more scalable, is well adapted to large scale networks.

Thus, for ad-hoc networks, the solution follows a single-server multiple clients architecture.[2] Upon the establishment of the ad-hoc network, one of the nodes becomes the server which is responsible for enforcing VFC. Naturally, the server device may also act as a client allowing all nodes to participate in the game.

For large-scale networks, the virtual world is partitioned into different, but not independent, regions handled by distinct servers. Players can move freely around the virtual world, transparently switching between regions of different servers and are able to seamlessly interact with players located in other regions. Each server enforces VFC for the region under its responsibility. In addition, each server communicates, through a subscription protocol, with other servers to apply VFC for players located in others, contiguous or not, regions. This solution is particularly suited to massively multiplayer online games (MMOG) in which a large number of players interact through an extensive virtual world, shared over a wide area network. Players control an entity (the avatar) that represents them in the game's virtual world. Avatars can move across the game map and interact with each other according to the instructions given by the human player through some input device (e.g., a keyboard or a mouse). Players can also find several objects (e.g., health items, food, weapons,...) and computer controlled characters (e.g., NPCs - non-player characters). Each avatar has its own state that comprises several properties like position, health, abilities and owned items. Interactions with other avatars or objects may change both its state and the others'.

Thus, in both architectures, the server (in a centralized solution) or the servers (in a partitioned solution) have a coordinating role regarding data management: write-lock management, update propagation and VFC enforcement. The client-server protocol is orchestrated by the Session Manager components of each server. Communication is performed between clients and the corresponding server on a star like topology using the services of components Network Layer and Serialization Layer. The Client Manager in the server administers client data and enforces the consistency model through two components, the Session Manager and the Consistency Management Block (CMB). The Object Pool Manager manages the game objects and encapsulates the stored data, performing every operation on it on behalf of the other components.

[2] The rationale for this choice is mainly due to the limitations of wireless communication technology (e.g. Bluetooth) that imposes a single node of the network to relay all messages between any two nodes.

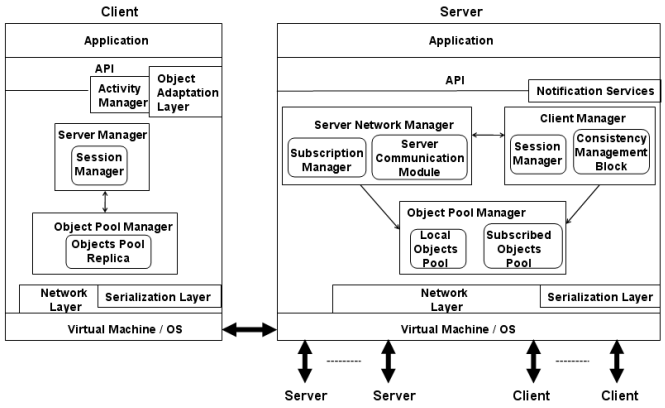


Fig. 5 VFC generic meta-architecture.

We now describe in more detail the VFC aspects that are common to any architecture: first, reading/writing objects (Section 3.2) and, secondly, how VFC is enforced (Section 3.3). Then, we focus on the architectural aspects that result from having the virtual world partitioned among several servers, each handling a region (Section 3.4). Note that in the following two sections (addressing common VFC aspects), when referring to a server we mean a single server in an ad-hoc architecture and, when in a large-scale partitioned architecture, a server means one of the several servers, each one handling a specific region of the virtual world. The communication that exists among such servers to ensure VFC in multiple regions of the virtual world (i.e. the subscription protocol) is addressed in section 3.4.

3.2 Read & Write Objects

The shared data is a collection of objects. Each client node maintains local replicas of all objects in the Object Pool container. The server maintains a primary copy of the object pool while the clients keep replicas of such objects. From the architectural viewpoint, there is no restriction whatsoever w.r.t. the representation of data (e.g. object graphs, tuples, relations). Also, the Object Adaptation Layer maps the application data representation to the architectural specific internal data representation.

Clients read and write objects through a VFC API. Read operations are performed on the local replicas without locking requirements (clients may read stale data). Write operations need to acquire locks in order to prevent the loss of updates. The server manages locks centrally; clients exchange messages with the server to acquire and release them. Object updates are sent to the server when clients release locks. The server propagates the new object versions to the other nodes according to the VFC specification.

With the exception of lock messages (for obtaining and releasing locks), nodes operate periodically w.r.t. the interactions between them. The server, periodically, sends a message to all clients defining a *round*. This has a twofold implication. In each round, the server sends round messages to the clients; updates are piggybacked on the round messages and merged at client object pools at reception time. On the other hand, it enables the execution of synchronized application handler functions (*activities*) at the client side. Whenever a round message is received the Activity Manager executes client activities. This feature may be used by many games based on turns. For example, activities may be used to update players locations, scores or other game state information. Since updates are received and merged before executing activities, the game programmers know that local replicas are stable when their activities execute (albeit possibly stale within the bounds defined by corresponding consistency κ vectors).

3.3 Enforcement of the VFC Model

The Consistency Management Block (CMB) at the server side enforces the VFC model. The CMB coordinates the propagation of updates to clients according to the VFC consistency parameters specified by each client. There are two phases: the *setup phase* and the *active phase*. During the *setup phase*, clients register the objects to be shared and send their consistency parameters (VFC ϕ settings) to the server; the CMB aggregates all the clients ϕ settings. The *active phase* is when clients may access the registered objects. In this phase, the server processes: 1) write requests (sent asynchronously by the clients piggybacked in lock release messages), and 2) round events (triggered periodically). The CMB is involved in handling both these events. It provides two functions that are called by the Session Manager (SM): CMB-Update-Received and CMB-Round-Triggered. As both functions are called, the CMB accumulates and computes the required information to build the clients' consistency views according to the previously specified ϕ settings. When called by the SM, the CMB-Round-Triggered function returns the updates to be sent to each client, which the SM piggybacks in the round messages.

Besides supporting VFC, the CMB module offers a generic interface allowing the support of different consistency models. The remainder of this section describes the internals of CMB that enforce VFC: update sending obeys the ϕ settings specified by clients. For each step we describe the algorithms underlying CMB-Update-Received and CMB-Round-Triggered functions.

Consistency Management Block In order to guarantee that all updates received since the last round event are taken into account when sending updates to clients in the next round, the CMB keeps track of which objects became dirty meanwhile in a bidimensional array D: each element corresponds to an object of the object pool regarding a particular client's view. In each particular client's view, an object is considered dirty if, according to the VFC requirement, its data should be propagated to the client.

```
CMB-Update-Received(o, u_o)
1  for c ← 1 to #_C
2  do if D[c, o] = 1
3        then continue
4     κ ← K[c, o]
5     S_σ[c, o] ← S_σ[c, o] + 1
6     if S_σ[c, o] ≥ κ_σ or
7        |ν(u_o) − S_ν[c, o]| ≥ κ_ν
8        then D[c, o] ← 1
9  Enqueue(U, ⟨o, u_o⟩)
```

a. CMB update handler.

```
CMB-Round-Triggered(t, M)
1  Merge(O, U)
2  for c ← 1 to #_V
3  do u ← New-Vector()
4     for o ← 1 to #_O
5     do κ ← K[c, o]
6        t_δ ← t − S_θ[c, o]
7        if D[c, o] = 1 or t_δ ≥ κ_θ
8           then Add(u, O[o])
9                D[c, o] ← 0
10               S_θ[c, o], S_σ[c, o], S_ν[c, o] ← t, 0, ν(O[o])
11       ⟨Z, C, P⟩ ← Φ(c, o)
12       z_closer ← −
13       for p ← 0 to #_P
14       do ⟨p_x, p_y⟩ ← ⟨P[p].x, P[p].y⟩
15          ⟨o_x, o_y⟩ ← ⟨O[o].x, O[o].y⟩
16          z ← Max(|p_x − o_x|, |p_y − o_y|)
17          z_closer ← Min(z_closer, z)
18       K[c, o] ← C[Z[z_closer]]
19    Piggyback(M[c], u)
```

b. CMB round handler.

Fig. 6 Pseudo-code of CMB.

Figure 6 presents the pseudo-code of the algorithms implementing this semantics. Whenever the server receives an update from a client, CMB-Update-Received is invoked and, if appropriate regarding the corresponding κ vector, the object is marked as dirty in D's correspoding entry and putting the update in the queue of pending updates U. At each round event, CMB-Round-Triggered is executed: it merges the pending updates in the object pool and sends all pending updates piggybacked in round messages to each client after testing the respective D dirty flags. Each D flag is then cleared meaning that the new versions were sent to the respective client.

The following data structures are also required: Z, C, P and K. Z, C and P refer to the data structures related to the clients ϕ settings (see Section 2.2). K is a matrix storing per object κ vectors of each view, that are valid during a time slot. κ vectors are computed per object, per view, according to clients ϕ settings. A κ vector is a consistency array that specifies when and which updates must be propagated to a client. Each κ consistency vector expresses three orthogonal dimensions: time, sequence and value. Each dimension is evaluated independently and auxiliary data structures (S arrays) are kept for each dimension. Each dimension of each κ consistency vector is evaluated as follows:

- *Time* – S_θ keeps the time of the last sent update. Whenever this time exceeds the one specified by κ_θ, the update is sent (see Figure 6.b lines 6-7) and the CMB internal state (D and S arrays) is reset. The time is approximated to a multiple of the round period.
- *Sequence* – S_σ is simply a counter of the number of updates that where received by the server since the last update was sent. There is a counter per object. When an update is received, this counter is incremented. When the counter exceeds the value κ_σ, the object is set to dirty in D in order to send the update in the next round (see Figure 6.a lines 5-8).
- *Value* – This qualitative dimension implies querying the object state to test when the difference to the last propagated version exceeds κ_ν. This query is evaluated by a function ν, provided by the game programmer and dependent of the game semantics. S_ν keeps the query result of the last propagated version and do the test of Figure 6.a line 7 whenever an update is received.

Calculating κ vectors is straightforward (see Figure 6.b, lines 11-18). Function $\Phi(c, o) \rightarrow \langle Z, C, P\rangle$ retrieves the ϕ settings referring to o for each client view s: Z, C and P. The algorithm proceeds as follows: 1) determines in which consistency zone z_{closer} the object is, and 2) resolves and stores in K the object consistency degree κ. Regarding the first step, since the object may be positioned in more than one consistency zone, each one belonging to a pivot, it is necessary to know which of these consistency zones imposes strongest consistency requirements. This is found by detecting which pivot is closer to the object, hence the z variable to evaluate the distance to a pivot and z_{closer} to keep the shortest one. Finding the distance from object o to a pivot $P = \langle p_x, p_y\rangle$ implies discovering in which P centered square of side l the object $\langle o_x, o_y\rangle$ is positioned such that $z = l/2 = Max(|p_x - o_x|, |p_y - o_y|)$. Since consistency zones are delimited by squares centered in P, it is enough to compare z with half the length of the squares that bound a certain consistency zone (e.g. s_1 for the inner square and s_2 for the outer square). Thus, the object is ensured to be in a determined consistency zone if $s_1 < z \leq s_2$. The operation that provides the number of the consistency zone based on z_{closer} is $Z[z_{closer}]$ in line 18. After determining which is the consistency zone of the closest pivot, determining which is the corresponding consistency degree is simply done by consulting the C table.

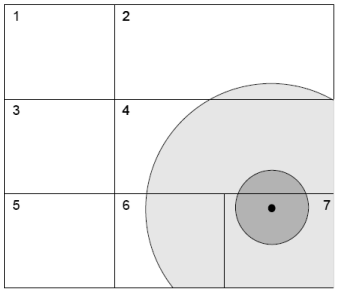


Fig. 7 Partitions with different dimension.

3.4 Inter-Server Subscription Protocol

As previously mentioned, each partition of the virtual world is assigned to one server of a distributed server network. Players are assigned to one of the servers based on the position of their avatars in the virtual world.

Because the *area-of-interest* (AOI) of a player may cross partitions handled by a different server (including non-adjacent partitions, as depicted by Figure 7), servers may need to share information with each other in order to enforce the consistency model and, thus, allow inter-partition interaction. To do so while minimizing server synchronization requirements, we designed a subscription protocol in which each server only knows about the non-adjacent servers it actually needs to be aware of (i.e., those whose partition may be crossed by one of its players' AOI). Furthermore, the protocol also ensures that an update from a player is received only by the servers whose objects may be affected by it. As a result, the protocol requires having only a *partial view* of the network (each server knows only a subset of the complete server set), which favors the scalability of the system.

In the partitioned architecture of VFC, servers are organized in a peer-to-peer Content Addressable Network (CAN [25]). Like in a CAN, each peer (in this case, a server) is responsible for one partition of the virtual space and, consequently, for the objects (i.e., avatars, other game items) located on it. Each server has only a *partial view* of the network, i.e, it knows only a subset of the total number of servers of the network. The *partial view* always contains the servers responsible for its adjacent partitions and it may also contain a dynamic number of other servers, added to the *partial view* as a result of the subscription protocol.

The partitioning of the virtual world is defined as servers join the network. Like in a CAN, a joining server chooses one of the existing servers and splits its partition in half, becoming responsible for one of the halves. Choosing with which server to split depends on the actual goal of the join: if the server is joining to relieve the load on one of the servers it has to know about that server beforehand (e.g., from a load monitor) and splits with it; if the goal is to improve the overall performance of the network, the system simply follows the CAN node join protocol, i.e, the server chooses a random position of the virtual world and joins that server.

3.4.1 Inter-partition Interaction

Two players may potentially interact when one is within the other's AOI. Thus, inter-partition interaction occurs when two interacting players are located on different partitions, adjacent or not. This means that the *view* of a given player may require information from multiple servers. Inter-partition interaction is supported by a player subscription protocol that gathers the information each server needs to know in order to enforce the consistency model. The protocol is divided in three parts (performed at independent times) and runs as follows.

Partition Subscription: When a server joins the network it immediately identifies and connects to those servers whose partition its objects' AOI may cross. To do so, it inspects the objects located on its newly defined partition to find the one with the largest AOI radius R. It then publishes to the network the dimensions of the area *partition outset* defined by adding R to each side of the partition. Publication consists in broadcasting the publication message to the server's neighbors: the server sends the message to its direct neighbors who, in turn, forward it to their neighbors and so forth, until every server whose partition is crossed (or contained) by the *partition outset* receives it; forwarding stops as soon as it reaches a server whose partition is not crossed (nor contained) by the *outset*. As a result of this publication, every server whose region is crossed by the *partition outset* informs the publishing server of its existence and is added to its *partial view*.

Object Subscription: The main task of the *Subscription Manager* (see Figure 5) consists in subscribing its own player objects (pivot objects) to servers that may contain information required by its players. It does so by executing an object subscription protocol when relevant changes occur that runs as follows:

1. The protocol starts by checking if any of its player's AOI crosses the partition of any server on its *partial view*. If it finds one such object, it adds the mapping "server ↔ player object" to a *subscription queue* to be processed in the next step.
2. Then, the *subscription queue* of step 1 is used to publish the list of objects to the corresponding servers, determined in that step. Publication is performed by directly sending, to each server, the list of objects whose AOI crosses its partition and, thus, require information about objects only known by it.
3. After the subscription process is finished, the mappings "object ↔ subscribed server" are stored in a *subscription table*.

Server Synchronization: The second part of the protocol consists in synchronizing objects between the servers according to the subscription results. Synchronization consists in forwarding/propagating the contents of each object's primary replica to the subscribing nodes. It is performed, conservatively, every time a player submits an update, although with an optimistic approach: when an update is received the server consults its *subscription table*

and retrieves from it the entry of the subscription table corresponding to the object to update; then, if necessary, it forwards the received update to the servers on that list.

As a result of these steps (i) the servers become aware of those players located in other partitions that may need information about their objects and (ii) each server knows to which other server updates to a given object have to be forwarded. Thus, each server has all the information it needs in order to enforce the consistency model and, as a result, allow consistent inter-partition interaction.

3.4.2 Ensuring VFC Across Partitions

As a result of the subscription protocol, each server has two disjoint sets of players: a set of *owned players* comprising the players that are located at the server's partition; and a set of *subscribed players* composed of the players that are located at a partitions handled by a different server, but whose AOI crosses the server's partition.

Obviously, a player must receive updates from every game object that is within his AOI, even those located at different partitions. This is achieved by having each server updating both its *owned* and *subscribed* players with the information about the objects located on its partition. This means that each server enforces VFC only for the slice of a player's AOI that crosses its partitions. As a result, the responsibility of enforcing VFC for a given player is distributed between the servers whose partitions are intersected by the player's AOI. Considering Figure 7, consistency for the player p represented by the solid (black) circle would be enforced complementarily by the four servers: S_1 would be responsible for updating p with the information about its four *owned objects*, S_2 would update it considering only its two *owned objects*, and so forth.

The enforcement of the consistency model requires servers to keep information about the state of each players' local copy of an object regarding the object's latest state. For this purpose, each server stores a *consistency table* that keeps, for each *owned* and *subscribed* player p and for each object o that is within the server's partition (i.e., for each *owned object*), the data related to the three metrics of consistency of the model:

- The time elapsed since the last time o was refreshed at player p;
- The number of updates to o that were not sent to p, since the last time it was refreshed with o;
- The *value* that object o had the last time it was sent to p.

With this information, VFC is able to enforce consistency on its players by executing the following tasks:

1) Update Processing: To update players according to their VFC specification, servers monitor the updates to their (*owned* and *subscribed*) objects. For that purpose, every time a server receives an update it executes one of the following actions, depending on the origin of the update:

- If the update is received from a player, then it concerns an *owned object*. Hence, the server updates its *consistency table* by updating, for each player (*owned* and *subscribed*), the entry corresponding to the updated object. This includes updating, for each player, the time elapsed since the object was refreshed and the number of lost updates of the object.
- If the update is received from a server as a result of server synchronization, then it concerns a *subscribed object* corresponding to a player p owned by a different server. As such, the information received is only necessary (as far as VFC is concerned) to update the object's position, so that, when enforcing the consistency model to p, VFC can correctly identify which objects are within p's AOI.

2) Client Updating: VFC propagates updates to clients periodically, according to their consistency specification and the information gathered in the *update processing* step. The process of updating clients is as follows:

1. First, the server identifies, for each player (owned and subscribed) p, which of the objects *owned* by the server (i.e, those that are located on its partition) are within p's AOI. Then, for each object o previously identified, it checks in which *consistency zone* of p's AOI object o is located. Finally, it verifies if o is in violation of the *consistency degree* associated with that consistency zone. If so, that object is queued and, after verifying the remaining objects, the server sends it to the player p.
2. After verifying consistency for every player (owned and subscribed), the server sends them the objects identified in the previous step.

As a result of these steps VFC achieves a distributed and decentralized consistency management algorithm in which the consistency of a single player is enforced not by a single server but by the complementary work of a group of servers. Having the load and the responsibility of enforcing consistency partitioned between the servers of the network improves the flexibility of the system and promotes scalability.

In this distributed setting with partitioning, serializability is ensured at each server. For each client, causality regarding its updates is ensured since FIFO order is enforced when sending updates to the servers. Asynchronism among servers is necessary for scalability, performance, playability. Thus, servers neither globally relay nor acknowledge each round (i.e. there is no distributed consensus). It is worthy to note that such synchronized rounds could be enforced within clusters, not in wide area. Therefore, updates performed by a given client to several objects owned by different servers, sent in the same round, may not arrive in the same exact round to each of the other clients (nonetheless, updates to individual objects are always propagated in total order). In this sense, inter-partition causality is temporarily relaxed, although recovered in the next round, as updates arrive. This allows

preservation of intent, a reasonable measure of causality assurance: the user sees a wider portion of the game world, observes interaction outcomes correctly, temporarily with lower detail or some lag/delay.

3.4.3 Player Transfer

We take advantage of the fact that the subscription protocol previously described needs information from objects located on remote servers to transfer a player's data before the player actually moves to a new partition. When a server S_i sends a subscription message directly to another server S_j it piggybacks the player's data on that message. When S_j receives the message it gets all the information it needs about the player from the piggybacked data.

The actual transfer of a player from his current designated server S_i to his new designated server S_d occurs only when the player's avatar moves to that server's partition. The transfer is triggered by S_i after it receives an update from the player that positions its avatar on an adjacent region. After finding out (by analyzing the entry of the *subscription table* corresponding to the object) which server S_d is responsible for that region, S_i issues the transfer request to it. Because the player's data has previously been transferred from S_i to S_d, the latter needs no additional information. As such, the transfer request is, in fact, a simple one-way asynchronous transfer notification.

4 Implementation

VFC has been implemented mainly as middleware prototypes for the two targeted architectures: ad-hoc (VFC-hoc) and large-scale networks (VFCLS). In this section we describe the essential implementation details, in addition to the detailed architectures explained earlier, regarding the main aspects of VFC core implementation, and the VFC-hoc and VFCLS prototypes. Additionally, we have implemented a multiplayer ad-hoc game using VFC and a simulating engine to compare VFCLS with competing approaches. As a more sophisticated proof-of-concept of application of the VFC model, we are currently extending an open-source distributed *first-person shooter* multiplayer game, in order to make use of VFC. We provide more details of this ongoing work in the next section.

The relevant core of VFC implementation consists of the Consistency Management Block (CMB) and the Session Manager components. The CMB internals implement the algorithms presented in Figure 6, regarding both the functionality and the data structures. The Session Managers of both the client and server sides execute the protocol that provides the VFC services to the game programmers. Each implements its own state machine (see Figure 8). Shaded circles represent the states; arrows between the states represent state transitions. State transitions are triggered by events. Each arrow description has two parts separated by a slash: the left side is the event name, the right side is the outgoing message sent to the remote peer. Straight arrows represent incoming messages, dashed ones represent API requests or internal events.

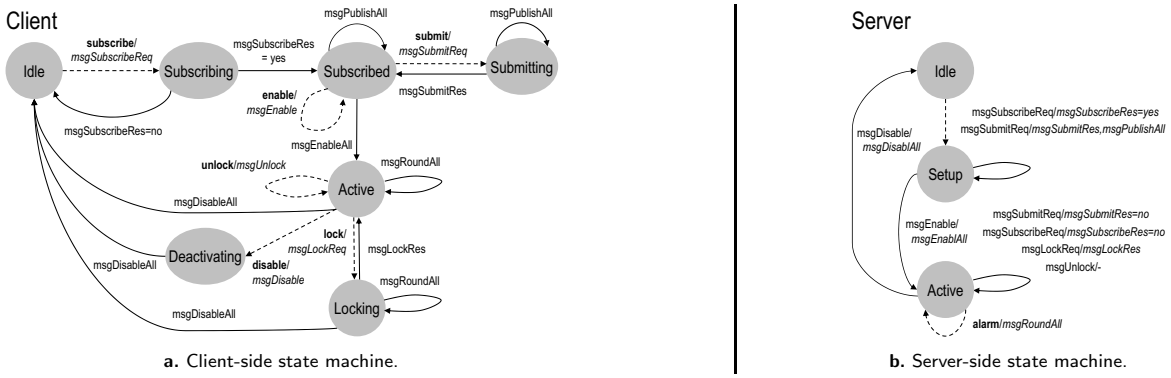
Session Managers coordinate in order to enforce the two phases already presented in Section 3: the *setup* and the *active* phases. Broadly speaking, first, the server declares its intention to accept client connections and enters the SETUP state. Then, clients connect to the server and subscribe to its services. Clients may now submit to the server the objects to be shared, which the server forwards to every client. When the server receives an *enable* request, it switches to the ACTIVE state and the system enters the *active* phase. While in this state, the server sends periodic round messages and handles lock and release requests. Updates are received by the server piggybacked with the release messages. The system leaves this phase when clients send the server a *disable* request causing the server to switch to the IDLE state.

As far as VFC is concerned, the virtual world is a bounded area populated with DataUnits. A DataUnit (DU) is an object that represents a shared game entity (like an avatar or a food object). Each DU carries a unique integer session identifier *duId* and the coordinates of the DU in the virtual world. Users are represented in the system by class UserAgent (UA). Like DUs, UserAgent objects also have a unique integer session identifier (*uaId*), along with an also unique nickname and reference/address, invoked to propagate notifications and updates. Servers also store a list containing the mapping between UserAgents and its corresponding DataUnits.

VFC-hoc internals: A prototype of VFC-hoc was implemented in Java and deployed on J2ME MIDP 2.0 CLDC 1.0 compliant devices (Nokia 6600 phones), according to the meta-architecture described in Figure 5. We use Bluetooth to support communication between the devices. The Network Layer uses JSR 82, the J2ME Bluetooth API, for discovery of nearby devices and services, management of active connections and sending/receiving data. Note that internally, the Network Layer is multithreaded in order to prevent blocking and increase parallelism. All messages exchanged between devices are implemented as Java objects.

Game state state can be represented as Java object graphs with individual objects registered in Objects Pool. Due to the lack of binary object serialization support in J2ME, a Serialization Layer was implemented in order to (un)marshal objects (see Figure 5). It requires objects to implement a specific interface allowing the middleware to read and write the object fields. The game programmer does not have to implement this code; a code enhancer was developed to transparently extend the application source code accordingly.

VFCLS prototype internals: The VFCLS prototype was developed in Java with Sun J2SE 6.0 development kit (JDK) and runtime environment. VFCLS was developed using only the standard Java libraries provided by JDK.



a. Client-side state machine. **b.** Server-side state machine.

Fig. 8 Client and server Session Manager state machines.

We used Java Remote Method Invocation (RMI) to support communication between the nodes of the system.

Integration with Applications and Programming Languages: For a consistency model to be widely used, it should be seamlessly integrated with popular programming languages, such as Java and C#. We describe how programmers can programmatically specify VFC ϕ settings. Pivots are registered by name and objects are associated with them using the overloaded methods defined by VFC `setPivot(String, Object)`, `setPivot(String, Object[])`. Sets of objects are selected by applying VFC declarative tags to object classes in source code, represented as Java *annotations* (`@VFCPlane{}`, `@VFCZone{}`) or .NET *attributes* (`[VFCPlane()]`, `[VFCZone()]`) with parameters stating zone *ranges* and κ-tuple components (e.g. `@VFCZone{int range, float time, int seq, float vDiff}`).

Java support for annotations is limited. In J2SE, it disallows multiple applications of the same annotation (even with different parameters) to the same class. Therefore, we make use of composite annotations (e.g. `@VFCPlane{}` that encapsulates the parameters of multiple `@VFCZone{}` annotations). In J2ME, there is no support for annotations whatsoever. Therefore, they are parsed as source code comments and classes extended to bear annotation parameters as private static fields.

Games must register objects in order for VFC to manage them. The VFC middleware can make use of reflection to allow registration of object graphs. At the client-side the game client must register the players' avatar with VFC. Likewise, at the server-side, the game server application may (depending on the game's design) register objects corresponding to non-player computer controlled characters.

To allow inspection of objects by VFC, classes must implement the `IVFCConsistency` interface that describes three methods: `getPosition` for objects to provide their current coordinates in the virtual world, `getValue` to provide their internal data to be propagated, and `vDiff` to provide an application-dependent measure (in percentage) of difference w.r.t. contents of another object.

After registration, objects can also be updated according to game logics. When an object is locally updated by a client, the game client (either explicitly or via enhanced source code) informs VFC via a UserUpdate notification. As a result, the update can be sent to the client's designated server.

If desired, game clients and server applications can be informed when a state update message is received from a server. For this purpose, when the application starts, they must register themselves as *update listeners* using a *RegisterStateUpdateListener* function provided by VFC.

VFC also provides functions for applications to query the local object pool. This allows, for example, game servers to perform validation and anti-cheating periodically, instead of every time an update is received.

5 Evaluation

In this section, we present the main results relative to VFC evaluation, regarding its suitability and flexibility for gaming experience, and its performance resorting to micro-benchmarks. The evaluation of VFC takes into account the two architecture instantiations proposed: ad-hoc, and large-scale networks.

5.1 VFC-hoc: VFC for Ad-hoc and Mobile Networks

VFC-hoc was deployed on Nokia 6600 phones connected via Bluetooth. To evaluate VFC-hoc qualitatively, we implemented a distributed multiplayer version of the popular Pac-Man game, illustrated in Figure 9. It has a maze divided into a matrix of 8×8 rooms; each room is assigned a 2-coordinate position. Players have access to the whole maze; yet, during the game, each player's device only shows the room where its avatar is in at that instant, stating the room coordinates $(0, 0)$ at the center of the screen. If two players' avatars are in the same room, they can see each other. If they are in adjacent rooms a periodic beep warns of opponent's proximity.

The quantitative evaluation studies the impact of VFC enforcement on overall VFC-hoc performance. Our micro-benchmarks focus on the most costly operation – the CMB-ROUND-TRIGGERED function, described in the CMB algorithm (see Figure 6), and on network delays. This function

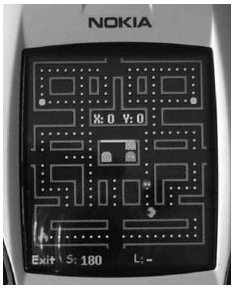


Fig. 9 Game view in one phone.

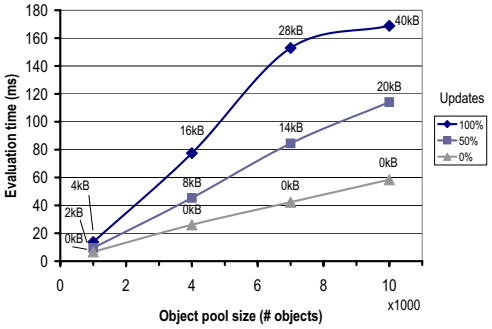


Fig. 10 Evaluation of CMB round handler.

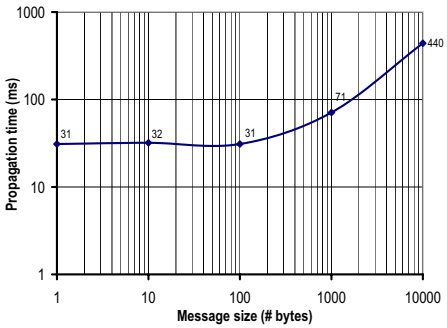


Fig. 11 Evaluation of message propagation delays.

not only performs intensive computations but it is also executed periodically, once per round. From the algorithm, we can see that, disregarding the cost of the merging operation in line 1, the overall cost is proportional to the number of clients. Thus, we evaluated the algorithm cost for a single client.

We continuously ran this micro-benchmark on phones and measured the execution time of the CMB-ROUND-TRIGGERED function (at the server side) by varying two factors: i) the number of objects in the pool (between 1000 and 10000 objects), and ii) the percentage of updates piggybacked in the round messages to the client (0%, 50% and 100% simulated update percentages).[3] Additionally, experiments were conducted with the following fixed conditions: i) the simulated ϕ settings included 1 pivot and small C and Z (arrays with 3 positions), ii) object payload was 4 bytes (e.g. 2 small integers for space coordinates). Figure 10 presents the performance measurements. Each result is annotated with the corresponding volume of data to be sent to the client.

In order to measure the cost of wireless communication, a second micro-benchmark was executed on the phones to measure network propagation time using Bluetooth. The size of the messages varied from 1 to 10000 bytes. Figure 11 presents the obtained propagation times which allows us to establish a comparison w.r.t. the VFC evaluation result.

Results show that VFC performance is influenced by the number of updates sent to clients, and therefore influenced by the VFC parameterization: weak consistency requirements cause less updates to be sent, increasing efficiency. Also, for a reasonable number of objects, the computation time is less than the corresponding transmission time in the network. Hence, the VFC computation costs can be masqueraded if they are performed in parallel with the transmission of the updates to clients and there is still time to attend game logic and rendering on the clients. Further, since the propagation time is nearly stable for messages below 200 bytes, the CMB may be enhanced to adapt the number of updates in order to increase efficiency.

[3] Updates are piggybacked in the round message if the test of line 7 of Figure 6.b is true. The micro-benchmark simulated this setting according to the update percentage provided as input.

Variation	Description
Aura1	Aura with radius of 40 units
Aura2	Aura with radius of 80 units
Aura3	Aura with radius of 120 units
VFC1	VFC with three zones with radius [40, 80, 120] and respective K vectors [3,0,0], [10,10, 0] and [50,10,500]

Table 2 Description of the different parameter variations

5.2 VFCLS: VFC for Large-Scale Networks

To evaluate VFCLS we developed a simulation infrastructure to simulate and compare different types of architectures (Centralized and Replicated C/S), as well as different usages of locality-awareness (Interest Management models, in particular auras), with VFCLS. Clients are simulated by a game skeleton where automatic clients move their objects - small circles - in straight lines along the game map, periodically changing the direction of their trajectory. The size of each object is 200 bytes, 50 `ints` or `floats` for game information. The tests were performed on two Intel Core2 Quad processors with 8.0 GB of RAM running Ubuntu Linux, connected by a Gigabit LAN.

Locality-Awareness: To compare the VFC consistency model with aura based IM regarding the bandwidth requirements imposed on players, we employed the parameter variations described in Table 2, and illustrated in Figure 12. The results in Figure 13 show the measured bandwidth in a context with variable number of players (50, 100 and 500) in a 1000x1000 virtual world. As expected, the performance of auras decreases as their radius increases, as the number of objects inside the AOI is higher. In VFC1, on the other hand, varying the radius of consistency zones does not necessarily mean that the bandwidth spent will increase. Since VFC has other configurable parameters, it is possible to increase the range covered by VFC zones while maintaining, or even reducing, bandwidth. This way, it is possible to enlarge player's visibility with little or no impact on bandwidth, although at the cost of fidelity (that decreases progressively or gracefully, nonetheless).

Although VFC1 only performs better than Aura3, to fully understand the meaning of these results, we have to

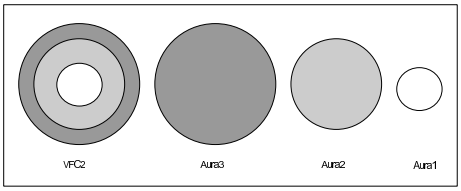


Fig. 12 AOI size and variations

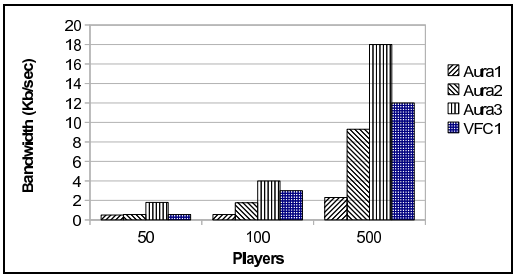


Fig. 13 Client-side usage requirements: auras versus VFC

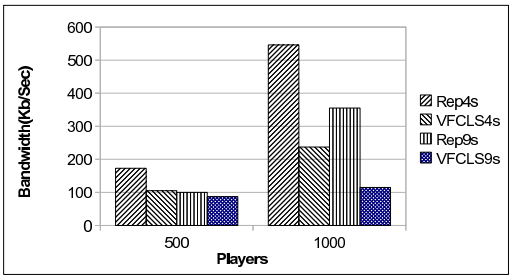


Fig. 14 Execution times of function round-trigger for different architectures: VFCLS and replicated architecture highlight

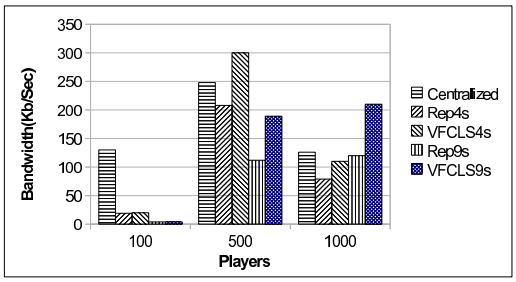


Fig. 15 Per-server bandwidth usage of different architectures

analyze them in light of Figure 12. Aura1 corresponds to the inner zone of VFC1. Hence, it is natural that VFC1 uses more bandwidth than Aura1. However, VFC1 corresponds only to an example VFC consistency vector defined by us. A different consistency vector could perform better than VFC1 and possibly yield bandwidth results similar to Aura1. Finally, we can see that the resulting bandwidth of Aura3 is higher than VFC1. More importantly, these results happen despite the fact that Aura3 and VFC1 cover the same exact area of the virtual world, a more relevant comparison. This is due to the high flexibility that VFC exhibits.

Architecture: We compared VFCLS with the centralized architecture (such as that of VFC-hoc) and a replicated architecture, by varying both the number of clients and the number of servers for each of the two distributed architectures (VFCLS and the replicated one), described in Table 3. We simulated clients with the following VFC specification: 3 consistency zones [120, 200, 500] and respective consistency vectors κ = [(3,0,0), (10,5,100), (50,10,500)]. The virtual world consisted of a 5000 x 5000 map. To analyze performance throughput we measured the execution time of function *CMB-Round-Triggered* in VFCLS. This also provides the impact on game's playability, as the more often a system is able to update its clients, the more interactive the game is. Figure 14 shows results considering 500 and 1000 players. It shows that VFCLS outperforms the replicated architecture both with 500 and 1000 players. With 500 players, VFCLS4s only outperforms the equivalent Rep4s, while Rep9s achieves better results, but still worse than VFCLS9s. With 1000 players, both VFCLS4s and VFCLS9s outperform the two replicated architectures, with each reducing by more than half the execution times of their replicated equivalent (Rep4s and Rep9s, respectively). Moreover, both can provide a highly interactive experience to its users, as the execution times are low. We can also see the difference between execution times of VFCLS9s with 500 and 1000 players is not meaningful, indicating good potential for scalability in VFCLS.

Server-to-client Bandwidth: Figure 15 shows results regarding bandwidth. At first glance, it looks like VFCLS not always saves bandwidth (500 players, VFCLS4s tops; with 1000 players, it is VFCLS9s). However, to fully understand this, we need to also take into account the results of the previous performance analysis. Since VFCLS is able to issue an higher number of round messages per second, it performs consistency enforcement more often than the other (slower) architectures. For instance, Rep4s can only perform consistency enforcement about once per second in the 1000 players context, while VFCLS9s does it almost five times more (considering that rounds are issued every 100 milliseconds, as was the case of our testing). Therefore, VFCLS is able to send messages more often to its players, which results in the higher bandwidth requirements. However, this is a false drawback, since it means that VFCLS can provide a highly interactive experience that the replicated architecture cannot.

Inter-server Communication: Table 4 shows the average number of messages exchanged between servers in a game context with 100 players. The results were obtained by replaying the traces obtained during a 5 minute simulation. We can see that VFCLS, due to its partitioning

Name	Description
VFC	Single server centralized architecture
Rep4s	Replicated architecture with four servers.
Rep9s	Replicated architecture with nine servers.
VFCLS4s	Four servers VFCLS
VFCLS9s	Nine servers VFCLS

Table 3 Description of the evaluated architectures

approach and the subscription protocol, limits the propagation of player update messages between servers. In particular, considering the 9 server configurations, VFCLS9s exchanges more than six times less messages than Rep9s. Even with a small number of servers, VFCLS4s reduces the number of synchronization messages to less than half, regarding Rep4s. Furthermore, the network overhead caused by transfer and subscription messages is not meaningful, which favors scalability. Even more relevant is that the difference between VFCLS4s and VFCLS9s, regarding those two types of messages, is relatively low. This indicates that the addition of servers has a small impact on synchronization, which also strongly favors scalability. Regarding server synchronization, in Rep4s and Rep9s, a server exchanges synchronization messages with every other server (3 and 8 servers, respectively). VFCLS, on the other, limits the number of synchronizing servers to a small set. Not only this reduces the network load, it also means that if a conflicting update occurs, the number of servers (and, as a result, players) affected by it is limited.

Arch.	Inter-server Messages	Average Servers in Synch	Subs.	Transf.
Rep4s	285747	3	—	—
VFCLS4s	104671	1.12	213	115
Rep9s	230967	8	—	—
VFCLS9s	37373	1.27	301	167

Table 4 Server communication in VFCLS.

VFC-enabled *First-Person Shooter* Game: We are currently extending an open-source *first-person shooter* distributed multiplayer game to make use of VFC: "*Cube 2: Sauerbraten*".[4] In Figure 16, we illustrate how a client perceives the game world, depicting 3 consistency zones and, according to player's orientation, its field-of-view.

Currently, this game employs only one server. For the purpose of evaluating VFC performance w.r.t. bandwidth usage (both inbound and outbound) by the server, we tested Cube2-VFC with 3 Intel Quad-Core machines with 8 GB RAM, connected via 1 Gigabit LAN. One executes the server and two others execute 48 simulated players (i.e., *bots* employing the game's very own artificial intelligence engine), 24 in each. During 10-minute runs, *bots* move around the game world in search of weapons and energy, while looking for enemies and shooting each other on sight.

Figures 17 and 18 evaluate bandwidth usage by employing VFC when compared with the game's original implementation. We measured bandwidth usage rate necessary to propagate: i) objects (i.e., player's object contents due to player's movements with consequent modification in position), ii) so-called *events* (shooting, triggering of sound effects, in-game messaging, etc.), and iii) total bandwidth usage (sum of the previous two, albeit dominated by object propagation).

[4] http://sauerbraten.org/

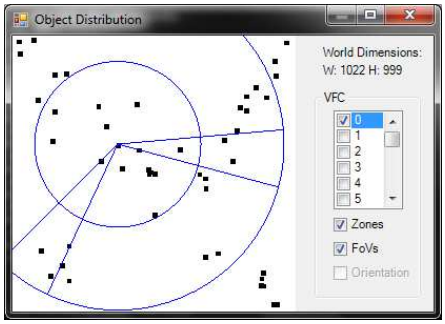


Fig. 16 Sky view of game world observed by client with object distribution across consistency zones (with indication of player's field-of-view).

On one hand, in Figure 17, we observe that inbound bandwidth rate, at the server, is similar with both approaches, i.e., introduction of VFC makes mostly no improvement nor degradation. This makes sense since the server must still receive all updates (movements) and events from all clients. On the other hand, in Figure 18, we can observe that VFC offers savings in total bandwidth usage of roughly 50%, an encouraging result. In particular, VFC cuts to half the bandwidth usage necessary to propagate objects, because it only propagates objects when this is required in order to meet the requirements expressed in κ vectors (VFC objects vs. native objects). Regarding events (VFC events vs. native events), there are also savings because the server is now applying VFC consistency requirements and thus filtering events (namely shooting and triggered sound effects) that are taking place too far from the player or, at least, propagating them less often (e.g., the player perceives farther shootings as delayed, and with fewer shots but still, he/she knows some other farther player is over there and shooting). Thus, it requires sending fewer messages to each of the clients.

In summary, although there is still a centralized server, VFC clearly introduces relevant bandwidth savings that improve game scalability w.r.t. the number of supported players, while ensuring the game experience's playability.

6 Related Work

In this section, we discuss and compare related work addressing relevant aspects in common with our work. VFC is a consistency model protocol that unifies optimistic consistency (divergence bounding) with locality-awareness techniques (traditionally employed in multiplayer games, but also applicable to cooperative editing). The meta-architecture proposed has been instantiated in two architectures for different scenarios (ad-hoc and large scale networks). The implemented prototypes aim at providing middleware support to ease the adoption of VFC in the development and deployment of multiplayer games. Therefore, due to space limitations, we focus mainly on the following: i) work on optimistic consistency in distributed systems with replicated data, ii) techniques specifically employed to improve

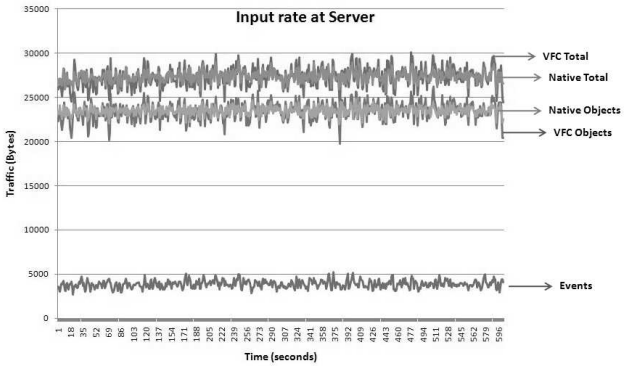


Fig. 17 Input bandwidth usage rate (B/sec.) at the server.

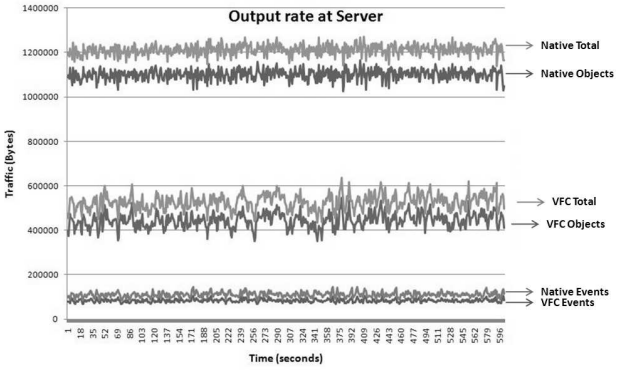


Fig. 18 Output bandwidth usage rate (B/sec.) at the server.

the performance and scalability of multiplayer games, resorting to middleware leveraging *locality-awareness*, iii) other middleware and network architectures for large-scale multiplayer gaming.

Optimistic Consistency with Replicated Data: Optimistic consistency techniques [27] (and divergence bounding in particular) are traditionally used in loosely-coupled scenarios, such as cooperative work, mobile computing, replicated databases. They are also suited to multiplayer games, as they may be employed to reduce bandwidth usage and masquerade latency.

Real-time guarantees [1] allow an object replica to remain in use while stale (i.e., without being refreshed) for a specified maximum time, before the replica must be made consistent. *Order bounding* [14] limits the number of uncommitted updates that may be applied to a replica. Transactions proceed faster because they can ignore the effects of a bounded number of other transactions preceding them.

Numeric bounding, as described in TACT [30,31], is a multi-dimensional consistency model proposing its combination with order bounding. Divergence bounds are applied to *conits* (i.e., physical or logical units of consistency) defined by the programmer. Numeric bounding defines maximum quotas (number or weight) for allowable updates to each replica (e.g. $100 for a number of replicas of a $1000 bank balance). A given replica cannot be further updated until it is made consistent w.r.t. operations performed on the other replicas, when its quota has been exhausted (e.g. by money withdrawals). While TACT proposes a multi-dimensional model for consistency enforcement and limiting replica divergence, it does not take locality-awareness into account. There is no notion of spacial relation neither among individual data objects nor among users. The middleware is oblivious to them. State is simply represented as individual database records or shared/replicated variables in servers. Therefore, it cannot be used in game scenarios where the consistency degree required for an object varies with player position and corresponding *sensing* and *acting* ranges. Numeric bounding is related with *escrow techniques* [15] on updates to data performed by mobile databases during disconnection periods, such as *reservations* in Mobisnap [24].

In VFC, we clearly extend optimistic consistency techniques by introducing support for locality-awareness. Furthermore, we can leverage the fact that, in our meta-architecture, each part of the game scenario is under the control of an assigned server. By monitoring all object updates, we extend escrow and numeric bounding techniques, allowing application programmers to define limits on the *value divergence* resulting from updates performed by other nodes (instead of simply limiting their own updates in a conservative manner).

Locality Awareness in Multiplayer Games and Simulation Environments: The notions of locality-awareness can be traced back to *interest-management* (or IM [20]), used to filter routing massive volumes of data in large-scale distributed simulations. It is motivated by the observation that players (or simulated entities) are not equally interested in (or affected by) every object (e.g., other players, entities, items). Instead, they are more concerned about objects located near them (e.g., their AOI) and, as the proximity to objects decreases, so does the player's interest in them. This observation is typically materialized by two different approaches and their variations: region based, and user based (e.g., auras, orientation, line-of-sight). Regions are contiguous partitions (either static or dynamic) of the game scenario defined by servers [29,3]. Players in the same region receive updates from each other, but not from others in different regions, which is not the case with VFC.

An aura [19] is a concentric consistency zone defined around a player's avatar. When the auras of two avatars intersect they can see each other; outside of its aura, a player sees nothing. The work in [21] (specific to peer-to-peer - P2P - gaming) reduces auras to a field-of-view, using obstacles in the virtual world to further reduce the scope of the aura. The work in RING [10] is particularly targeted to environments with high level of occlusion. Updates are forwarded only to entities in line of sight, regardless of distance. A^3 [4] reduces updates further by combining auras with player orientation in a two-level consistency approach: a 180 degrees field-of-view hides all updates behind the user but a small radius aura ensures that in case of fast rotations, close players and objects appear without delay.

Auras have a traditional problem with objects whose distance to the player oscillates around the aura radius: they keep on appearing and disappearing abruptly, which does not happen with VFC. Leveraging knowledge about user orientation, aiming, and line-of-sight is specially relevant in some classes of games (*first-person-shooters*) but not applicable to e.g., *real-time strategy* games. Moreover, they require additional information, and possibly intervention, regarding game logic and scenario geometry.

Locality-awareness was proposed in [8] to perform load balancing on massive multiplayer games, with an adaptable mechanism to partition vast virtual worlds into regions handled by a cluster of dedicated servers to ensure scalability. Based on their locations, players are redirected to servers in charge of the corresponding partition. To handle hot-spots (e.g. crowding, player flocking), heuristics determine when to reduce server load (by splitting highly populated partitions) and leverage idle resources (coalescing empty partitions in the same server). VFC can extend such mechanisms to allow interactions between users handled by different servers.

Matrix [3] proposes the use of locality-awareness by perceiving a multiplayer game as a *decomposable system* [28] with stronger interaction within each given *subsystem* (e.g. a room, a game level) than among different *subsystems* (e.g. across rooms). Based on this premise, a *radius* or *zone of visibility* can be identified for each event in the game, outside of which, the corresponding updates need not be propagated (e.g. a shot in another room). Thus, the system enforces *pockets* of *locally-consistent* state. While already offering an approach based on localized consistency, Matrix still adheres to an overly limitative approach of *all-or-nothing* consistency, with no method of stating maximum replica divergence. Furthermore, it makes use of a global consistency radius instead of multiple and dynamic zones of consistency with different divergence bounds, as we propose in VFC.

The work in Donnybrook [6] employs a more aggressive version of locality awareness by restricting interest management to a limited number of entities (i.e., players) regardless of their spatial distribution and density. It is specifically tailored to reduce traffic of *first-person shooters* in P2P scenarios. It leverages the notion that (human) players can only keep up with a limited number of simultaneous opponents (a interest set of five); therefore, they only receive updates from five users regardless of player density. This set is recalculated at every frame w.r.t. proximity and aim of other players, and interaction recency with a fine-tuned weighting. It is very effective in a relevant game (Quake III) but this (even more) strict all-or-nothing approach targets only this class of games in P2P architectures (interest set may not apply in others). It also requires extensive information from and intervention to game logic.

The work in [17] describes how the usage of the MMASS (*Multilayered Multi-Agent Situated System Approach*) model can help to design, simulate, manage, and deploy large-scale collaborative environments where people (or in fact agents) may move around a specific scenery and interact. Agents' decisional information and interactions are enriched with contextual awareness (in particular location-awareness) targeting a ubiquitous computing scenario where external sensors may also be employed. Although in a different context, we observe parallelism between the MMASS and VFC approaches, while highlighting their differences.

In the MMASS model, the spatial structure (therefore, agents' location) is represented resorting to general adjacency or undirected connectivity multi-layered graphs. Vertexes hold information that may exist at interconnected multi-layers, possibly corresponding to different types or levels-of-detail of data. VFC makes usage of a cartesian N-dimensional space that is more suited to gaming, and requires less modifications on game code since it needs no additional information regarding, for instance, specific geometrical information of the game world, and/or modifications to game logic. In MMASS, however, graphs are more suited to effectively quickly decide if two agents may or may not interact, albeit this requires additional specific code to maintain adjacency information as agents roam around.

Regarding event (in MMASS) and update (in VFC) propagation, both are inspired by the fundamental notion of *field*, although, embodied in very different ways and with different goals. In MASS events comprise information regarding interaction among agents and are propagated across graph edges, subject to possible composition, where custom agent-specific *diffusion* functions rule event amplification or attenuation, i.e., if an event should be propagated or not to adjacent agents. This equates to the diffusion of a field, with a very important example being the presence field (for proximity perception) that is emitted by agents on the move. Therefore, the presence of an agent is perceived by the agents where its presence field is propagated before attenuation threshold is reached.

In VFC, field intensity (i.e., the promptness of the propagation of an update to a client) must be ruled by distance-based attenuation (as with the gravity and electrical field), with specific field attenuation for each pivot and type of object. The important difference is that in the MMASS model, field range and attenuation result from customized functions of each agent, advantageous for rich behavior simulation. On the contrary, in VFC, field range and attenuation, while customized, are fully described resorting to declarative data (consistency zones and vectors) to avoid significant and extensive intrusion (or modification) to game logic code (possibly none with the help of reflection).

Large-Scale Architectures for Multiplayer Games: Large-scale architectures for multiplayer games exist in two main flavors: distributed client-server (either replicated or partitioned), and peer-to-peer (P2P), with network traffic reduction being essential to both.

In distributed client-server systems, the game management is a responsibility of dedicated servers; clients simply play the game. According to the approach to load balancing, these systems can be classified as *partitioned* or

replicated. In a replicated system [16,9] each server holds a copy of the complete virtual world, but is only in charge of managing a subset of the players. The goal is to foster responsiveness by assigning players to servers geographically closer to them. However, since all servers hold the whole game state, the entrance of one player has a direct impact on the performance of every server. To enforce IM, servers compare all of their players with every object of the system. As the system grows the overhead of representing and processing every object becomes a bottleneck and the performance of the system decays. Hence, the scalability of these systems is limited.

Partitioned systems (such as VFC in large-scale networks) achieve load balance by dividing the virtual world into *partitions*, assigning each one to a different server in the network [3,7,2]. They provide mechanisms to make it unnoticeable to the player, giving him the illusion of being in a single large contiguous virtual world. Servers must synchronize in order to support interaction between players located on different partitions and movement of players across partitions. In [2], the authors propose a partitioned architecture that uses locks to support these two critical situations: when a player moves to another partition or interacts with a player located on a remote partition, his responsible server locks that partition, preventing other players from performing any actions and, consequently, avoiding conflicts. However, if the servers are connected through relatively high latency networks (like the Internet) locks reduce the system's concurrency, leading to a strong cut in performance. VFC does not suffer from this limitation, as it does not require distributed locking of partitions. The authors of [7] and of Matrix [3] propose solutions that do not rely on locks. However, the former uses a static partitioning strategy disallowing servers joining the network, preventing adaptation to load peaks. The latter does allow the addition of servers, but its operation is based on a single coordinator server that performs critical tasks related to consistency management and state repartition. Compared with VFC, that employs a decentralized approach, and allows servers to join and leave, they lack on flexibility and scalability.

P2P support for multiplayer games is an active research topic [12,5,18]. In these systems, clients exchange updates directly, instead of doing so through a server. They enable game creation and enrollment to be performed in a ad-hoc manner, instead of handled exclusively by central servers. P2P systems put the burden of managing the virtual world on peer applications executed on the players' computers, which are considerably resource constrained when compared with dedicated servers. As the number of players of a game increases so does network traffic and the amount of data each peer receives and processes. As a result, the performance of peers may degrade to reduce computational capability and network bandwidth to forward messages to other peers, instead of only one assigned server in VFC. Furthermore, this architecture may not be acceptable to developers of commercial games who want to provide the game as registered, controlled, or paid service. The work described in [13] proposes the use of peer-to-peer (P2P) network topologies, such as Pastry [26], to handle massive multiplayer games. Locality-awareness may be leveraged in order to dynamically organize nodes in groups, reflecting common areas of interest within the virtual world. Therefore, updates to objects are only propagated to other nodes within the same group, which encloses an isle of consistency within the virtual world. Nonetheless, programmers must explicitly pre-define the static partitioning of the virtual world, defining areas of interest. Consistency is therefore strictly enforced within each one and ignored outside altogether, unlike in VFC.

The work in [11] is focused on traffic selection according to its *urgency* (immediate forwarding) and *relevancy* (reliable delivery) to maintain scalability in wide-area scenarios in multiplayer games. Game developers must define statically, for each entity (e.g. class of objects), levels of urgency and relevance. The middleware generates code that assigns network resources dynamically during the game based on the provided requirements. Another popular approach to reduce network traffic is *dead reckoning* [22] that consists in predicting player's movement until the next network packet arrives with updated position and velocity. Although offering control at some level over replica divergence, these works do not explore locality-awareness. Thus, the divergence of all objects of a given type (e.g. representing players) is driven by global parameters regardless of their relative spatial position w.r.t. each player. This *one-size-fits-all* approach is inflexible and may waste bandwidth w.r.t. a more fine-grained and adaptive approach embodied in our proposal.

7 Conclusions

In this paper we present a novel consistency model and meta-architecture to manage replicated data (VFC). In addition, we present middleware prototypes (VFC-hoc, VF-CLS) adopting VFC to support multiplayer distributed games both in ad-hoc and large-scale networks.

While some of previous works embody the notions of *consistency radius*, *locality of interest*, or isles of *localized consistency*, they adopt a rather *all-or-nothing approach*. must be kept strongly consistent, while the values of (or updates to) objects outside that area are simply discarded. VFC combines and extends more sophisticated consistency models (e.g., TACT), with the notion of *locality-awareness* in a unified model. VFC offers an intuitive, simple and flexible abstraction such that application programmers can easily express their consistency requirements according to application semantics.

Regarding future work, we envisage to perform thorough empirical studies using real games to compare the performance of VFC with other game consistency protocols and frameworks, regarding namely: i) the benefit of our solution in terms of efficiency/playability, and ii) the

flexibility of VFC in parameterizing consistency requirements for different game scenarios.

We are currently addressing the adaptation of VFC to other non-competitive settings such as cooperative work; for example, cooperative document editing, or replicated wikis, where position coordinates refer to sentences, paragraphs, sections, etc. Another avenue in progress is embedding VFC in a software development environment plug-in (in particular, for Eclipse) for cooperative (team-based) application development, where position coordinates refer to project entities such as namespaces, modules, classes, methods and fields.

References

1. R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems (TODS)*, 15(3):359–384, 1990.
2. M. Assiotis and V. Tzanov. A distributed architecture for mmorpg. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 4, New York, NY, USA, 2006. ACM.
3. R. Balan, M. Ebling, P. Castro, and A. Misra. Matrix: Adaptive middleware for distributed multiplayer games. *ACM/IFIP Middleware Conference*, 2005.
4. C. E. Bezerra, F. R. Cecin, and C. F. R. Geyer. A3: A novel interest management algorithm for distributed simulations of mmogs. In *DS-RT '08: Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, pages 35–42, Washington, DC, USA, 2008. IEEE Computer Society.
5. A. Bharambe, J. Pang, and S. Seshan. Colyseus: a distributed architecture for online multiplayer games. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 12–12, Berkeley, CA, USA, 2006. USENIX Association.
6. A. R. Bharambe, J. R. Douceur, J. R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang. Donnybrook: enabling large-scale, high-speed, peer-to-peer games. In V. Bahl, D. Wetherall, S. Savage, and I. Stoica, editors, *SIGCOMM*, pages 389–400. ACM, 2008.
7. W. Cai, P. Xavier, S. J. Turner, and B.-S. Lee. A scalable architecture for supporting interactive games on the internet. In *PADS '02: Proceedings of the sixteenth workshop on Parallel and distributed simulation*, pages 60–67, Washington, DC, USA, 2002. IEEE Computer Society.
8. J. Chen, B. Wu, M. Delap, B. Knutsson, H. Lu, and C. Amza. Locality aware dynamic load management for massively multiplayer games. *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 289–300, 2005.
9. E. Cronin, A. R. Kurc, B. Filstrup, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. *Multimedia Tools Appl.*, 23(1):7–30, 2004.
10. T. A. Funkhouser. Ring: a client-server system for multi-user virtual environments. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 85–ff., New York, NY, USA, 1995. ACM.
11. C. Griwodz. State replication for multiplayer games. *Proceedings of the 1st workshop on Network and system support for games*, pages 29–35, 2002.
12. T. Hampel, T. Bopp, and R. Hinn. A peer-to-peer architecture for massive multiplayer online games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 48, New York, NY, USA, 2006. ACM.
13. B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. *IEEE Infocom*, 2004.
14. N. Krishnakumar and A. Bernstein. Bounded ignorance: a technique for increasing concurrency in a replicated system. *ACM Transactions on Database Systems (TODS)*, 19(4):586–625, 1994.
15. N. Krishnakumar and R. Jain. Escrow techniques for mobile sales and inventory applications. *Wireless Networks*, 3(3):235–246, 1997.
16. M. Kwok and J. W. Wong. Scalability analysis of the hierarchical architecture for distributed virtual environments. *IEEE Trans. Parallel Distrib. Syst.*, 19(3):408–417, 2008.
17. M. P. Locatelli and G. Vizzari. Awareness in collaborative ubiquitous environments: The multilayered multi-agent situated system approach. *ACM Transactions on Autonomous and Adaptive Agents (TAAS)*, 2(4):13:1–13:21, 2007.
18. H. Lu. Peer-to-peer support for massively multiplayer games. In *INFOCOM*, 2004.
19. G. Morgan, F. Lu, and K. Storey. Interest management middleware for networked games. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 57–64, New York, NY, USA, 2005. ACM.
20. K. Morse et al. *Interest Management in Large-scale Distributed Simulations*. Information and Computer Science, University of California, Irvine, 1996.
21. J. Pang. Scaling peer-to-peer games in low-bandwidth environments. In *In Proc. 6th Intl. Workshop on Peer-to-Peer Systems IPTPS*, 2007.
22. L. Pantel and L. C. Wolf. On the suitability of dead reckoning schemes for games. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 79–84, New York, NY, USA, 2002. ACM.
23. D. Pittman and C. GauthierDickey. A measurement study of virtual populations in massively multiplayer online games. In *NetGames '07: Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 25–30, New York, NY, USA, 2007. ACM.
24. N. Preguiça, J. L. Martins, M. Cunha, and H. Domingos. Reservations for conflict avoidance in a mobile database system. In *Proc. of the 1st Usenix Int'l Conference on Mobile Systems, Applications and Services (Mobisys)*, 2003.
25. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.
26. A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001, Proceedings*, pages 329–350, 2001.
27. Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.
28. H. A. Simon. The architecture of complexity. *Proceedings of the American Philosophical Society*, 106:467–482, 1962.
29. S. Xiang-bin, W. Yue, L. Qiang, D. Ling, and L. Fang. An interest management mechanism based on n-tree. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2008. SNPD '08. Ninth ACIS International Conference on*, pages 917–922, Aug. 2008.
30. H. Yu and A. Vahdat. Design and evaluation of a conflit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems (TOCS)*, 20(3):239–282, 2002.
31. H. Yu and A. Vahdat. The costs and limits of availability for replicated services. *ACM Transactions on Computer Systems (TOCS)*, 24(1):70–113, 2006.

Aknowledgements: The authors would like to thank José Lopes, Tiago Bernardo, and Bruno Loureiro for their implementation work during several stages of VFC-related projects, as well as research project PTDC/EIA/66589/2006 funded by FCT.