Vector-Field Consistency for Ad-Hoc Gaming

Nuno Santos, Luís Veiga, and Paulo Ferreira

INESC-ID/Technical University of Lisbon
Distributed Systems Group
Rua Alves Redol N 9, 1000-029 Lisboa
{nuno.santos,luis.veiga,paulo.ferreira}@inesc-id.pt

Abstract. Developing distributed multiplayer games for ad-hoc networks is challenging. Consistency of the replicated shared state is hard to ensure at a low cost. Current consistency models and middleware systems lack the required adaptability and efficiency when applied to ad-hoc gaming. Hence, developing such robust applications is still a daunting task. We propose i) Vector-Field Consistency (VFC), a new consistency model, and ii) the Mobihoc middleware to ease the programming effort of these games, while ensuring the consistency of replicated objects. VFC 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). Mobihoc adopts VFC and provides game programmers the abstractions to manage game state easily and efficiently. A Mobihoc prototype and a demonstrating game were developed and evaluated. The results obtained are very encouraging.

Keywords: Consistency Management, Replicated Objects, Locality-Awareness, Multiplayer Games.

1 Introduction

The growing utilization of personal appliances such as PDAs and cell phones enables the proliferation of ad-hoc networks. Ad-hoc networks form spontaneously between two or more devices communicating via wireless interfaces. Due to their entertaining nature and motivated by this technological advance, distributed multiplayer games are particularly interesting to deploy in such environments. Once an ad-hoc network is formed, people may play these games irrespective of the place they are (e.g. public transports, restaurants) without the need for a structured network and without incurring into any connectivity expenses.

In distributed multiplayer games there is a need for data sharing between the network nodes (e.g. player positions, maps, scores). Enforcing data consistency requires additional communication for update propagation and synchronization operations. In ad-hoc networks, communication-intensive operations are critical and have a twofold negative impact. Firstly, the high latency, the reduced network bandwidth and the small processing capability of devices brings overheads that dramatically hinder game playability. Secondly, extensive access to the network

R. Cerqueira and R.H. Campbell (Eds.): Middleware 2007, LNCS 4834, pp. 80–100, 2007.

causes devices batteries to consume rapidly. In order to circumvent these negative impacts, game programmers tend to use programming tweaks, low level optimizations and error-prone message-passing approaches to keep the shared data consistent. As a side effect, software becomes harder to manage and less reliable.

Current approaches to optimistic consistency [1] relax the strict consistency model to reduce communication expenses. The common assumption is that applications may allow data inconsistencies up to a certain limit and enable application programmers to specify these limits according to the semantics of applications. The criteria for slacking consistency varies: by divergence between the values of replicas, on a time-basis [2], by applying application based predicates on replica values [3,4], sequential ordering [5], or combining several approaches [4,6]. However, these proposals are inadequate to cope with the dynamics of distributed games: consistency requirements change often and quickly throughout the game execution, namely w.r.t. the players' position in the *virtual world*. The above mentioned systems lack the required adaptability and are inefficient when applied to the ad-hoc scenario. On the other hand, current middleware for multiplayer games embodies the notion of *locality-awareness* (traceable to [7,8]) but offer a very limited consistency model [9], or use it just to drive load-balancing [10] and network traffic between servers [11].

In this paper, we propose a new consistency model for replicated objects called *Vector-Field Consistency* (VFC) and present *Mobihoc*, a middleware adopting VFC to support multiplayer distributed games in ad-hoc networks. A Mobihoc prototype was implemented on the J2ME platform. To demonstrate its feasibility, a distributed version of Pacman was implemented on top of Mobihoc. Both Mobihoc and Pacman were deployed and evaluated in real mobile phones (Nokia 6600) with good performance results.

VFC is an optimistic consistency model allowing bounded divergence of the object replicas. The VFC novelty is the following. 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, and ii) how the consistency requirements are *specified*. The first issue is dealt by employing locality-awareness techniques. It 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. The second issue is handled by providing a 3-dimensional vector for specifying consistency degrees. Each dimension of the vector bounds the replica divergence in *time* (delay), *sequence* (number of operations) and *value* (magnitude of modifications) constraints. Game programmers 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 game programmers: the consistency model based on pivots is intuitive and the parameterization settings allow the game programmer to specify the consistency requirements for a wide range of game scenarios. Second, from the players

viewpoint, VFC allows user experience to proceed within acceptable parameters in the sense that, as far as the players are concerned, the rules of the game are being abided to, and users are provided with all the 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 is mostly useful for those games where the number of updates to propagate is high and the interactivity with the user is demanding. Despite addressing multiplayer games, VFC and Mobihoc can also be used to develop any other cooperative applications based on replicated shared-data.

This paper is organized as follows. Section 2 briefly describes the VFC consistency model. Section 3 presents the Mobihoc architecture. Section 4 describes the implementation details of Mobihoc. Section 5 presents and discusses the obtained experimental results. Section 6 surveys the relevant related work and Section 7 draws some conclusions.

2 Consistency Model

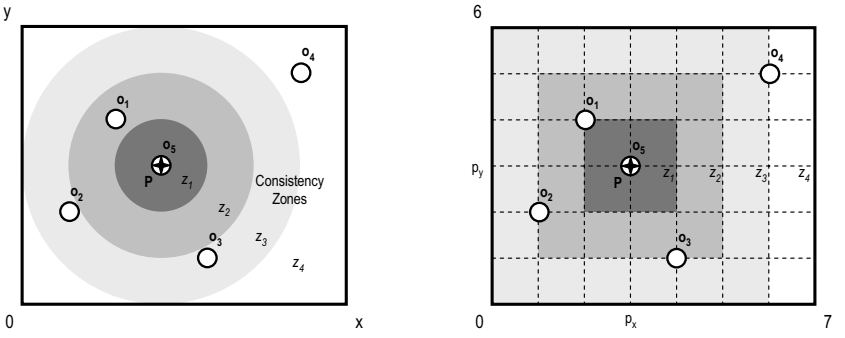
In VFC, objects are positioned within a *virtual world*, an abstraction of an N-dimensional space. Without loss of generality, we consider the virtual world to be 2-dimensional. In many games these abstractions map immediately to the game semantics; for example, in the Pacman game, the virtual world 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.

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

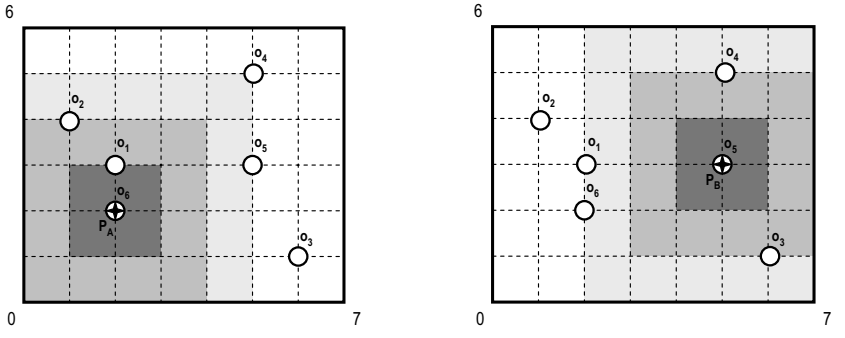
Within a particular view, object consistency depends on their distance to a *pivot* (P). It is characterized by a position in the virtual world and it can move over time. A pivot can be an object (e.g. the Pacman player) or just a function (e.g. an editor cursor). Figure 1.a illustrates a virtual world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



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

function of the 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 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 Pacman game, ghosts consistency weakens as they are 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 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. 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.

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 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 on the object internal state and is implementation dependent (e.g. it may reflect a drift regarding the player score or the player life charge).

[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.

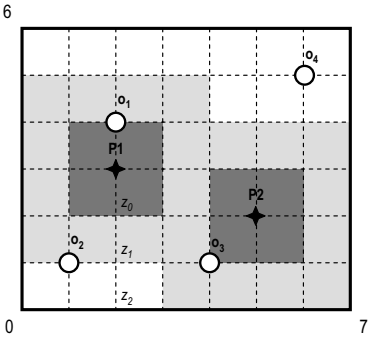


Fig. 3. Multi-pivot generalization

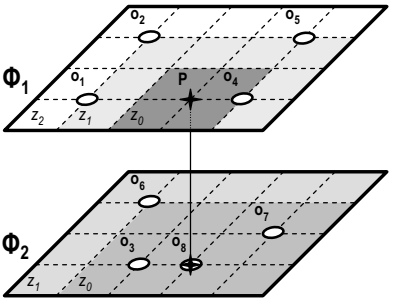


Fig. 4. Multi-zones generalization

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 $\kappa_m \leq \kappa_i \leq \kappa_M$.

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.

The multi-zones generalization allows different sets of objects to be characterized differently w.r.t. their consistency requirements. For example, in Pacman, 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

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.

Fig. 5. Table describing the ϕ parameters of VFC

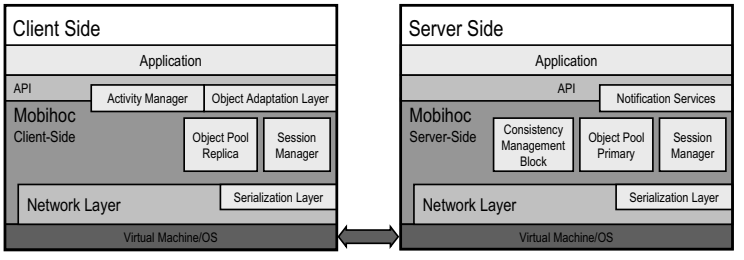


Fig. 6. Mobihoc architecture

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 setting is described by $\phi_i = [O_i, Z, C, V]$, where $O_i \subseteq O$. Figure 5 presents a table summarizing 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

Mobihoc is a middleware platform aimed at supporting the design of multiplayer distributed games for ad-hoc networks. Mobihoc enforces VFC by managing the game state between the network nodes and provides programmers with the adequate means to parameterize VFC according to game semantics.

Mobihoc follows a client-server architecture (see Figure 6).[2] Upon the establishment of the ad-hoc network, one of the nodes becomes the server. Naturally, the server device may also act as a client allowing all nodes to participate in the

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

game. The server has 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 peer. Communication is performed between clients and the server on a star like topology using the services of components Network Layer and Serialization Layer.

The remainder on this section presents, firstly, the mechanisms for reading/writing objects (Section 3.1) and, secondly, the mechanisms for VFC enforcement (Section 3.2), and exposes other relevant architecture components.

3.1 Read and Write Objects

The shared data is a collection of objects. Each 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 Mobihoc internal data representation.

Mobihoc allows clients to read and write objects through its 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 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.

3.2 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*

```
CMB-Update-Received(o, u_o)
1  D[o] ← 1
2  Enqueue(U, ⟨o, u_o⟩)
```

a. CMB update handler.

```
CMB-Round-Triggered(t, M)
1  Merge(O, U)
2  u ← New-Vector()
3  for o ← 1 to #_O
4  do if D[o] = 1
5        then Add(u, O[o])
6             D[o] ← 0
7  for c ← 1 to #_C
8  do Piggyback(M[c], u)
```

b. CMB round handler.

Fig. 7. Pseudo-code of CMB Version 1

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.

In spite of implementing VFC, the CMB module offers a generic interface allowing Mobihoc to support different consistency models. The remainder of this section describes the internals of CMB that enforce VFC. The description of the CMB algorithm is performed gradually as three versions are progressively presented for a better understanding: 1) the CMB sends every client all updates performed since the last round event, 2) the CMB supports consistency degrees (κ vectors), 3) the CMB provides full VFC support, i.e. 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.

Version 1. In order to guarantee that all updates received since the last round event are sent to all clients in the next round, the CMB keeps track of which objects became dirty (i.e. were written) meanwhile in array D. Only the dirty objects are propagated to clients. Figure 7 presents the pseudo-code of the algorithms implementing this semantics. D has an entry per object of the object pool. Whenever the server receives an update, CMB-Update-Received is invoked setting the object as dirty in D 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 clients after testing the D dirty flags. D is then cleared meaning that the new versions were sent to all clients.

Version 2. This version considers that, instead of sending all updates to every client, there is a consistency vector κ, common to all clients. κ specifies when and which updates must be propagated to clients. Figure 8 presents the pseudo-code

```
CMB-Update-Received(o, u_o)
1  S_σ[o] ← S_σ[o] + 1
2  if S_σ[o] ≥ κ_σ or
3     |ν(u_o) − S_ν[o]| ≥ κ_ν
4     then D[o] ← 1
5  Enqueue(U, ⟨o, u_o⟩)
```

a. CMB update handler.

```
CMB-Round-Triggered(t, M)
1  Merge(O, U)
2  u ← New-Vector()
3  for o ← 1 to #_O
4  do t_δ ← t − S_θ[o]
5     if D[o] = 1 or t_δ ≥ κ_θ
6        then Add(u, O[o])
7             D[o] ← 0
8             S_θ[o], S_σ[o], S_ν[o] ← t, 0, ν(O[o])
9  for c ← 1 to #_C
10 do Piggyback(M[c], u)
```

b. CMB round handler.

Fig. 8. Pseudo-code of CMB Version 2

of the algorithms that support consistency vectors. The 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. Without loss of generality, we assume there is a single and fixed κ vector for all clients, thus all clients receive the same updates obeying κ. Each dimension 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 8.b lines 4-5) 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 8.a lines 1-4).
- *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 8.a line 3 whenever an update is received.

Version 3. In order to fully support VFC, it is required to maintain per client consistency views. This imposes two extensions w.r.t. the CMB Version 2: 1) D and S become bidimensional matrices where the additional dimension regards individual client views, and 2) κ vectors are computed per object, per view, according to clients ϕ settings. To this extent, additional data structures are required: K, Z, C and P. K is a bidimensional matrix storing per object κ vectors of each view, that are valid during a time slot. Z, C and P refer to the data structures related to the clients ϕ settings (see Section 2.3).

Calculating κ vectors is straightforward (see Figure 9, 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

```
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)
```

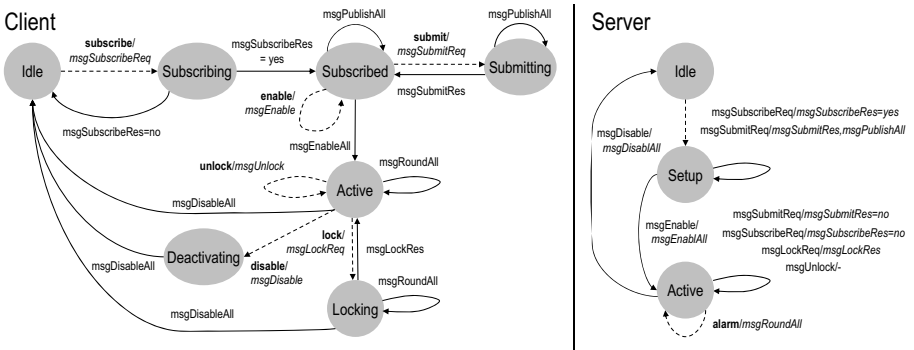
b. CMB round handler.

Fig. 9. Pseudo-code of CMB Version 3

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.

4 Implementation

A prototype of Mobihoc was implemented on J2ME. Mobihoc can be deployed on J2ME MIDP 2.0 CLDC 1.0 compliant devices. The prototype design follows the architecture of Figure 6. In this section, we specify the most relevant implementation details of the internal components (Section 4.1) and provide a brief insight on how the game programmers specify the VFC ϕ settings (Section 4.2).



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

Fig. 10. Client and server Session Manager state machines

4.1 Implementation Internals

We adopted Bluetooth to support communication between the network nodes. The Network Layer (see Figure 6) uses JSR 82, the J2ME Bluetooth API, for discovery of nearby devices and services, management of active connections and sending/receiving data. Internally, the Network Layer is multithreaded in order to prevent blocking and increase parallelism. All messages exchanged between peers are implemented as Java objects.

Due to the lack of binary object serialization support in J2ME, a Serialization Layer was implemented in order to (un)marshal objects (see Figure 6). 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 compiler was developed that transparently extends the application source code accordingly. Naturally, since it is not possible to access the already compiled class code, there are several limitations concerning the objects that can be serialized. The fields of the serializable objects are required to be: i) Java primitive types or, ii) serializable objects or, iii) arrays of primitive types/serializable objects or, iv) Vector and Hashtable objects of the Java API. Message objects exchanged via the Network Layer observe these restrictions.

Game programmers are invited to share data as Java object graphs. The current implementation of the Object Adaptation Layer maps directly the objects of the graph into objects individually managed and stored in the object pools. Further optimizations may assemble clusters of application level objects to be managed as single units. Notice that, since these objects require to be (de)serialized in order to be exchanged between clients and server, game programmers must follow the constraints imposed by the Serialization Layer.

The Mobihoc core consists of the CMB and the Session Manager components. The CMB internals implement the algorithms presented in Figure 9, regarding both the functionality and the data structures. The Session Managers of both the client and server sides execute the protocol that provides the Mobihoc services to the game programmers. Each implements its own state machine (see Figure 10).

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.

Due to space constraints it is not possible to fully explain the details of the Session Manager state machines. Briefly, 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 into 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.

4.2 Integration with Programming Languages

For a consistency model to be widely used, it should be seamlessly integrated with popular programming languages, such as Java and C#. In this section, we describe how programmers can programmatically specify VFC ϕ settings.

Pivots are registered by name and objects are associated with them using the Mobihoc overloaded methods `setPivot(String, Object)` and `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 sequence, float valueDiff}`).

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. In .Net (including .Net CF) there is support for multiple application of attributes to classes which eases programmers' lives (e.g. `[VFCZone(range, time, sequence, valueDiff])` applied as `[VFCZone(10,0.5,5,0.2)]`, `[VFC Zone(20,1.5,15,0.6)]` and `[VFCZone(30,4.5,25,0.9)]`).

To allow inspection of objects by Mobihoc, 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 `valueDiff` to provide an application-dependent measure (in percentage) of difference w.r.t. contents of another object.

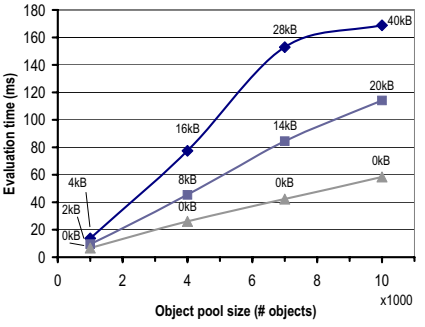


Fig. 11. Evaluation of CMB Version 3 round handler

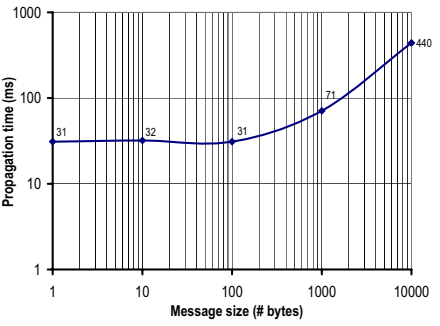


Fig. 12. Evaluation of message propagation delays

5 Evaluation

The Mobihoc prototype was evaluated in a twofold perspective: quantitative (Section 5.1) and qualitative (Section 5.2). The former consisted on a Mobihoc performance study; several mini-benchmarks were implemented for this purpose. The latter evaluated the effectiveness of Mobihoc in developing distributed games. We implemented a distributed Pacman game for this purpose. The code was deployed and executed in Nokia 6600 phones.

5.1 Quantitative Evaluation

The main objective was to study the impact of VFC enforcement on the overall Mobihoc performance. VFC is implemented in Mobihoc according to the CMB Version 3 algorithm (see Figure 9). Due to space limitations, we focus our attention on the most costly operation – the CMB-Round-Triggered function. This function not only performs intensive computations but it is also executed periodically, once per round. Observing the algorithm of Figure 9, it is straightforward to see that, disregarding the cost of the merging operation in line 1, the overall cost is proportional to the number of clients. Thus, we implemented a micro-benchmark in order to evaluate the algorithm cost for a single client.

Several experiments were conducted by running this micro-benchmark on Nokia 6600 phones and measuring 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

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

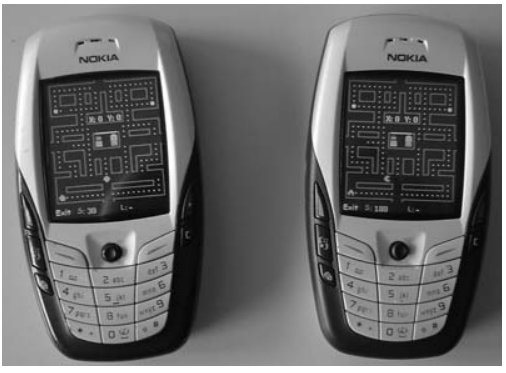


Fig. 13. Two phones enroled in a distributed Pacman game

Fig. 14. Game view in one phone

(e.g. 2 small integers for space coordinates). Figure 11 presents the performance measurements. Each result is annotated with the corresponding volume of data to be sent to the client.

In order to better perceive the real impact into the overall system performance, we also measured the cost of wireless communication. For this purpose, we implemented a second micro-benchmark, deployed it and executed it on two Nokia 6600 phones to measure network propagation time using Bluetooth. The size of the messages varied from 1 to 10000 bytes. Figure 12 presents the obtained propagation times which allows us to establish a comparison w.r.t. the VFC evaluation result.

Results show that as the number of updates (sent to clients) grows, the VFC overheads increases. Thus, we infer that performance is influenced by the VFC parameterization: weak consistency requirements cause less updates to be sent, increasing efficiency. Also, considering 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.

5.2 Qualitative Evaluation

To evaluate Mobihoc qualitatively, we implemented a distributed multiplayer version of the popular Pacman game. Our version of the game considers 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. If two players' avatars are in the same room, they can see each other. Figure 13 is a snapshot of two devices enroled in a Pacman game match. It captures a moment

were both avatars are in the same room. Figure 14 displays a magnified screen of one of these devices showing the details of the game in that room, particularly the avatars and the room coordinates $(0, 0)$ at the center of the screen.

The implementation of this game explores Mobihoc and VFC features for sharing the game state as follows. The maze is mapped to a bi-dimensional 8×8 virtual world. The game state referring to rooms, players and ghosts is implemented as objects with a position in the virtual world. Rooms are assigned a fixed coordinate regarding its overall location in the maze. Both players and ghosts, regardless of having fine grained positions within each room relevant for the game semantics, w.r.t. consistency, are also assigned a position referring to the room where they are at each moment. For each player, we consider that there is only one pivot assigned to its avatar. Also, we defined $Z = [0, 1]$ characterizing three consistency zones. The first zone affects the objects in the same room as the avatar; here, consistency is required to be strong. The second refers to the avatar adjacent rooms; it is a weaker consistency zone and it is relevant mainly when the avatar leaves the current room. The weakest consistency zone is beyond the adjacent rooms. We defined three consistency degrees based on the sequence dimension (σ).

This game, while being very simple and using few VFC features, demonstrates the usability of VFC and Mobihoc. Our experience, from the application programmer viewpoint, is that the model is intuitive, it is simple to describe consistency requirements and to programmatically use Mobihoc employing VFC as a consistency model. We believe it is straightforward to describe consistency requirements for more demanding game scenarios.

6 Related Work

In this section, we discuss relevant work related to ours. Since we are addressing consistency enforcement for multiplayer games in ad-hoc networks, we focus on: i) other work regarding optimistic consistency (see [1] for a thorough survey) in the presence of replicated data, ii) game development for ad-hoc and mobile networks using resource constrained devices such as PDAs and mobile phones, iii) other techniques leveraging *locality-awareness* (in games) to improve middleware performance and scalability, and iv) middleware support for game development and deployment.

Optimistic Concurrency and Divergence Bounding. Optimistic consistency techniques are mostly used in loosely-coupled scenarios (e.g. mobile computing). We find they are also suitable to multiplayer games in ad-hoc networks, as they may be employed to circumvent known issues associated with low bandwidth and high latency.

Real-time guarantees [2] allow an object replica to remain stale and still be used (i.e., without being refreshed) for a specified maximum time, before the replica must be made consistent. *Order bounding* [5] is used to limit the number of uncommitted updates that may be applied to a replica. This allows given

transactions to proceed faster because they can ignore the effects of a bounded number of transactions preceding them.

Numeric bounding is introduced in TACT [4,6], a multi-dimensional consistency model that proposes its combination with order bounding. Numeric bounding is based on the notion of defining maximum quotas for allowable updates to each replica (e.g. $10 for a number of replicas of a $100 bank balance). Once the quota has been completely used by a replica (e.g. to withdraw money from the account), the replica can no longer be updated until it is made consistent w.r.t. operations performed on the other replicas. Although TACT proposes a multi-dimensional model for consistency enforcement and limiting replica divergence, it does not embody any notion of locality-awareness. 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 also related with *escrow techniques* [3] on data updates which are employed by mobile databases during disconnection periods, such as *reservations* in Mobisnap [12].

In VFC, besides introducing support for locality-awareness in existing optimistic consistency techniques, we are also able to extend them. We leverage the fact that in the ad-hoc networks we address, there is a central node in charge of routing that is able to monitor all object updates. Therefore, we are able to further 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).

Game Development for Ad-hoc and Mobile Networks. The work in [13] compares the two dominant platforms for ad-hoc gaming (Java J2ME, and .Net Compact Framework) w.r.t. portability and performance of native code invocation, numerical and graphic code. It also studies the performance of several communication strategies (namely packet forwarding). Though providing insights on the environments we are addressing, it assumes a strict consistency model, with a centralized game server.

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. Although offering control at some level over replica divergence, this work does not explore locality-awareness. Thus, the divergence of all objects of a given type (e.g. representing players) is bound by global parameters irrespective 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.

Locality Awareness in Large-Scale Multiplayer Middleware. The notions of locality-awareness can be traced back to *interest-management* [8], used to filter routing massive volumes of data in large-scale distributed simulations. Locality-awareness is employed in [10] to perform load balancing on massive multiplayer games. The authors propose a transparent mechanism to partition vast virtual worlds into a cluster of dedicated servers to ensure scalability. Based on their locations, players are redirected to servers in charge of the corresponding partition. As this approach is vulnerable to hot-spots in the game (e.g. crowding, player flocking), it employs heuristics when to reduce server load (by splitting highly populated partitions) and leverage idle resources (coalescing empty partitions in the same server).

The work described in [14] proposes the use of peer-to-peer (P2P) network topologies, such as Pastry [15], to handle massive multiplayer games, in a scalable and cost-effective way, due to the increased flexibility provided by self-organization, while obviating the need for dedicated servers. This also enables game creation and enrollment to be performed in a ad-hoc manner, instead of handled exclusively by central servers. These properties can be leveraged with locality-awareness 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.

Communication between nodes is handled by multicast using Scribe [16]. Object state is kept consistent by employing a coordinator-based approach, analogous to the tokens employed in Mobihoc. The effects of varying population density, growth, message aggregation, and network dynamics are also studied. 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.

Matrix [9] proposes the use of locality-awareness by perceiving a multiplayer game as a *decomposable system* [7] where there is 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. Matrix requires programmers to explicitly tag individual packets carrying updates with their corresponding spatial coordinates where they took place in the game. With this information, the middleware checks the game visibility radius (a global parameter) and decides whether and where to forward the packet. While providing a very interesting approach based on localized consistency, Matrix also 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.

The work in [17] also explores locality-awareness but w.r.t. actual physical location of players that must wear tags. It describes a number of experiments trying to determine how game accuracy and feed-back detail (e.g. graphics, sound) may be balanced against the communication latency observed.

Other Large-Scale Multiplayer Middleware. Regarding online games in wide-area networks, the work in [18] proposes to re-use server infrastructure to deploy several MMOG[4] side-by-side. It describes a service platform that can host a number of games on-demand, leveraging existing grid technology. The work described in [19] proposes a methodology to reduce human-resources costs in MMOG development. It makes use of message-oriented middleware, arguing that games typically operate in an event-driven manner. Protocol and message-handling code for clients and servers is automatically generated from message descriptors written in XML. Game and virtual-world logic is managed via *adapters*, which can be plugged-in asynchronously, on-the-fly within an entire running MMOG application. These works, while being relevant cases of employment of middleware to support online multiplayer games, are not targeted to the kinds of constrained devices and ad-hoc networks we are addressing.

7 Conclusions

In this paper we present a novel consistency model to manage replicated data (VFC) and a middleware platform (Mobihoc) adopting VFC to support multiplayer distributed games in ad-hoc 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*. Thus, objects inside an area of interest must be kept strongly consistent, while the values (or updates to it) of objects outside are simply discarded. VFC combines and extends more sophisticated consistency models (such as TACT), with the notions of *locality-awareness* in a unified model. VFC and Mobihoc provide intuitive, simple and flexible abstractions such that application programmers are able to easily express their consistency requirements according to application semantics. Moreover, VFC and Mobihoc are widely applicable, not being restricted to the development of distributed games for ad-hoc networks.

Regarding future work, we envisage to perform thorough empirical studies to compare the performance of VFC and Mobihoc with other game consistency protocols and frameworks. Also, we aim to employ our solution to different types of real games in order to i) analyze the benefit of our solution in terms of efficiency/playability, and ii) to explore the flexibility of VFC in parameterizing consistency requirements for different game scenarios. Although the partitioning of the game space into zones depends on the application semantics, we envisage to develop the mechanisms to help game programmers better deciding how to partition the game space into zones. Additionally, we intend to study how this approach scales across either number of objects or number of nodes, and possibly to redesign Mobihoc to environments other than ad-hoc networks.

[4] Massive Multiplayer Online Games.

Acknowledgments. The authors wish to acknowledge the students José Lopes and Tiago Bernardo for their implementation work in the Pacman game.

References

1. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. 37(1), 42–81 (2005)
2. Alonso, R., Barbara, D., Garcia-Molina, H.: Data caching issues in an information retrieval system. ACM Transactions on Database Systems (TODS) 15(3), 359–384 (1990)
3. Krishnakumar, N., Jain, R.: Escrow techniques for mobile sales and inventory applications. Wireless Networks 3(3), 235–246 (1997)
4. Yu, H., Vahdat, A.: Design and evaluation of a conflit-based continuous consistency model for replicated services. ACM Transactions on Computer Systems (TOCS) 20(3), 239–282 (2002)
5. Krishnakumar, N., Bernstein, A.: Bounded ignorance: a technique for increasing concurrency in a replicated system. ACM Transactions on Database Systems (TODS) 19(4), 586–625 (1994)
6. Yu, H., Vahdat, A.: The costs and limits of availability for replicated services. ACM Transactions on Computer Systems (TOCS) 24(1), 70–113 (2006)
7. Simon, H.A.: The architecture of complexity. Proceedings of the American Philosophical Society 106, 467–482 (1962)
8. Morse, K., et al.: Interest Management in Large-scale Distributed Simulations. In: Information and Computer Science, University of California, Irvine (1996)
9. Balan, R., Ebling, M., Castro, P., Misra, A.: Matrix: Adaptive middleware for distributed multiplayer games. In: Alonso, G. (ed.) Middleware 2005. LNCS, vol. 3790, Springer, Heidelberg (2005)
10. Chen, J., Wu, B., Delap, M., Knutsson, B., Lu, H., Amza, C.: Locality aware dynamic load management for massively multiplayer games. In: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 289–300. ACM Press, New York (2005)
11. Griwodz, C.: State replication for multiplayer games. In: Griwodz, C. (ed.) Proceedings of the 1st workshop on Network and system support for games, pp. 29–35 (2002)
12. Preguiça, N., Martins, J.L., Cunha, M., Domingos, H.: 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)
13. Janecek, A., Hlavacs, H.: Programming interactive real-time games over WLAN for pocket PCs with J2ME and.NET CF. In: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games, pp. 1–8. ACM Press, New York (2005)
14. Knutsson, B., Lu, H., Xu, W., Hopkins, B.: Peer-to-peer support for massively multiplayer games. In: IEEE Infocom, IEEE Computer Society Press, Los Alamitos (2004)
15. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)

16. Castro, M., Druschel, P., Kermarrec, A., Rowstron, A.: Scribe: a large-scale and decentralised application-level multicast infrastructure. IEEE Journal on Selected Areas Commun(JSAC) (Special Issue on Network for Support Multicast Commun.) 20(8), 100–110 (2002)
17. Mansley, K., Scott, D., Tse, A., Madhavapeddy, A.: Feedback, latency, accuracy: exploring tradeoffs in location-aware gaming. In: Proceedings of ACM SIGCOMM 2004 workshops on NetGames 2004: Network and system support for games, pp. 93–97. ACM Press, New York (2004)
18. Saha, D., Sahu, S., Shaikh, A.: A service platform for on-line games. In: Proceedings of the 2nd workshop on Network and system support for games, pp. 180–184 (2003)
19. Hsiao, T., Yuan, S.: Practical middleware for massively multiplayer online games. IEEE Internet Computing 9(5), 47–54 (2005)