Fault-Tolerant Vector-Field Consistency

(extended abstract of the MSc dissertation)

André Santos
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisors: Professors Paulo Ferreira & Luís Veiga

***Abstract*—In recent years there has been an exponential growth of games on mobile devices. Multi-player ad-hoc network games are not easily developed because of the inherent issues of mobile devices and ad-hoc networks, such as limited connectivity, low processing power and short battery time. Vector-Field Consistency is an optimistic consistency model which reduces network usage, by selecting important updates to propagate to replicas. VFC is enforced by Mobihoc, a middleware platform, whose goal is to ease the development of multiplayer distributed games for ad-hoc networks. In this work we extend the VFC model in order to support the entry and departure of nodes from the system, that is, to make a fault-tolerant VFC.**

I. Introduction

Nowadays, mobile devices such as the cell phone are widespread. They come equipped with technologies (e.g., bluetooth and wifi[1]) that ease the spontaneous creation of ad-hoc networks[2]. With the proliferation of this kind of devices, the number of applications specifically developed for this kind of environments has also been increasing. Specifically, mobile game has seen a huge growth. With the advent of the ad-hoc networks, people can even play distributed multiplayer games wherever they are (e.g. public transports, restaurants).

In distributed multiplayer games, there is a very high need for data exchange between network nodes (e.g., player positions need to be updated constantly on every node). In ad-hoc networks, this kind of applications that require constant communication, suffer from two major drawbacks. Firstly, the high latency, the reduced network bandwidth and the small processing power of mobile devices bring overheads that dramatically hinder game playability. Secondly, high rate network accesses and the processing of the game itself, drain this devices batteries rapidly. Furthermore, the sole nature of ad-hoc networks themselves suggest a constant change of the network topology. Nodes can join or leave the system in premeditated ways or, in the latter case, also by failing (e.g., the devices batteries can run out).

Vector-Field Consistency (VFC) [3] is a consistency model for replicated objects. This model ensures that two replicas of the same object will eventually be consistent. Although it can be applied to any kind of distributed application, the main VFC purpose is to support distributed multiplayer games over ad-hoc networks. VFC selectively and dynamically strengthens or weakens replica consistency based on the actual game state. It does this at the same time that manages how the consistency degree changes throughout game execution and how the consistency requirements are specified. The consistency degree of each replica is obtained through locality-awareness techniques. This model considers that throughout the game execution, there are certain 'observation points', called *pivots* (e.g. player's position), around which the consistency is required to be strong and weakens as the distance from the pivot increases. Since the players position can change with time, so do the requirements about the replicated objects consistency. The consistency requirements are dealt with a tri-dimensional vector that specifies the consistency degrees. Each vector dimension bounds the replica divergence in *time* (delay), *sequence* (number of operations) and *value* (magnitude of modifications) constrains. The ability to choose which updates should be propagated translates into a much more efficient usage of the mobile devices resources.

This model, despite being developed for ad-hoc networks, does not support the dynamic entry and exit of nodes in the network. The main goal of this master thesis is to change the VFC model in order to allow both orderly and disorderly entry and exit of nodes in the system. Furthermore, it must be possible to persistently store the current system state, in order to safely re-join the game later, without loosing any data.

In order to reach the proposed goals it is necessary to overcome a few obstacles. The main obstacle is how to keep executing the game (or any other type of application) that uses the VFC model, after the departure of one of the network nodes. Because VFC uses a Client-Server architecture, the solution gets even more difficult to achieve if the node that left was responsible for executing the server. In this specific case, not only must the departure of the node be secured but another node must become the new server. Unfortunately, handling how a node leaves is not the only issue regarding node departures. First, the system must know that a node has actually left, if it has not notified the remaining ones before leaving. Another obstacle that must be faced is how can the system state be stored persistently in order to restart it later to the same point. This problem can occur when a node leaves the network in an orderly fashion, as well as in a disorderly fashion.

In spite of the solution that is developed to overcome this obstacles, it will always bring some additional *overheads* to the amount of data that is transferred between nodes. This is the reason why, assuring that the benefits that are drawn from the VFC model are maintained, is another obstacle that must be overcome. It is very important that the use of a fault tolerant VFC continues to reduce the amount of data that is transferred between nodes, in comparison to the same system without VFC. If communication is kept to a minimum, the mobile devices batteries last longer.

This document is organized as follows. Section II we present some related work. In section III we describe our solution's architecture and its most relevant implementation details. Section IV presents the evaluation results of our solution. Finally, in section V we draw some conclusions.

II. Related Work

A. Replication

In distributed systems, data replication is a key technology to achieve enhanced performance, higher availability and fault tolerance [4]. It consists on the maintenance of copies of data across multiple computers. For instance, a web domain can replicate data over several servers to divide the workload and to be able to answer client requests when a server fails. Data replication should be transparent to the user and a certain degree of data consistency between all the replicas must be achieved.

1) Passive Replication: In the *passive* model of replication for fault tolerance, there is, at all times, only one replica manager answering client's requests. This replica manager is known as the primary replica manager. All other replica managers available in the system only serve as backups.

Each client only issues requests to a single replica manager (the primary one), the remaining replica managers wait for the primary to fail to take over and resume operation.

2) Active Replication: In the active replication model, every replica manager has an equal role. They work as a group and function like a state machine. Active replication was first introduced by Leslie Lamport as state machine replication [5].

As opposed to the passive replication model, in active replication the front-ends send operations to the group of replica managers, and not to a single primary one. For this reason, if a replica manager crashes, it does not affect the overall system performance, because every remaining replica is performing the same operation.

3) Pessimistic Replication: Consists on the synchronous coordination of replicas during accesses and the blockage of other users during an update. With pessimistic replication, when a replicated object is updated, every replica of that object must be synchronously updated before another request is performed on the system.

This model for replication is widely used in commercial systems, but was mainly intended for working over local-area networks, where latencies are small and failures are scarce [6], [7].

4) Optimistic Replication: This kind of replication provides algorithms that allow access to replicated data without *a priori* synchronization, based on the "optimistic" assumption that problems will rarely occur [8]. This approach constitutes the exact opposite of the pessimistic one, where synchronization is the primary concern to avoid possible conflicts. In optimistic replication, updates are propagated on the background and periodic conflicts are solved after they happen. This replication strategy greatly improves availability of systems, and optimistic algorithms can scale to a large number of replicas. All these benefits, however, come at a cost: optimistic replication must deal with issues regarding diverging replicas and conflicts between concurrent operations. It is thus, only applicable for applications which can tolerate occasional conflicts and inconsistent data.

The main objective of every optimistic replication system is to be eventually consistent, that is, at some point in time every replica will be consistent with one another. This goal can be achieved through a variety of ways:

- *Number of writers*: a replication system can be *single-mastered* or *multi-mastered*. A single-master system can only have a single designated writer. All updates originate at the master, and are then propagated to the backups. In a multi-master system, updates can be submitted at multiple sites, and are then propagated on the background to the other replicas.
- *Definition of operations*: a replication system can be *state-transfer* or *operation-transfer*. In the former case when an object is updated it is propagated to the other replicas. In the latter case, when an object is updated, it is the operation that transformed the object that is propagated to the other replicas.
- *Scheduling*: there can be two types of scheduling policies: *syntactic* and *semantic*. Syntactic policies sort operations only with regards to their time, place or author. Semantic policies exploit semantic properties, such as commutativity or idempotency of operation.
- *Handling Conflicts*: can also be of the types *syntactic* and *semantic*. Syntactic policies rely on the timing of the operations for conflict detection, and are not application-specific like the semantic policies.
- *Propagation Strategies and Topologies*: Ad-hoc topologies transfer operations through the network in a epidemic fashion, and out-perform fixed topologies in dynamic scenarios. Regarding the degree of synchrony, systems can be *pull-based* or *push-based*.
- *Consistency Guarantees*: define how much divergence a client application may observe. They can be single-copy consistent, eventual consistent or use the mechanism of session guarantees[9] (*read your rights*, *monotonic reads*, *writes follows reads* or *monotonic writes*).

B. Fault-Tolerance

1) Multicast Communication: Enables a group of processes to receive copies of messages sent to the group with delivery guarantees. These guarantees include agreement on the set of messages that every process in the group should

receive and on the delivery ordering across every group member[10].

Multicast algorithms can have different degrees of reliability and ordering semantics. *Basic multicast* guarantees that every correct process will eventually deliver a message, as long as the multicaster does not crash. *Reliable multicast* guarantees that if a correct process delivers a message, every other correct process eventually delivers the same message. *Ordered multicast* satisfies the same properties as the previous ones and introduces three possible ordering guarantees: *FIFO*, *causal* and *total*.

2) Group Membership: Manages the dynamic membership of groups and multicast communication. It has four main roles in a group communication system: providing an interface for group membership changes, implementing a failure detector, notifying members of group membership changes and performing group address expansion.

3) Failure Detectors: Must be implemented in order to detect when a group member has crashed[11]. It may be *reliable*, over a synchronous system, or *unrealiable*, over an asynchronous one.

4) View-Synchronous Group Communication: Extends the reliable multicast semantics to include the changing of group views. A new view is delivered to the group of processes whenever there is a change on the group membership (e.g. a process joins or leaves) and all members are guaranteed to see the same view contents.

C. Rollback-Recovery

Rollback-recovery (*checkpointing*) is another technology that adds reliability and high availability to distributed systems [12]. It accomplishes this by enabling processes to save recovery information periodically at a stable storage device. After a failure, a process can then restart computation from an intermediate state by accessing the stored information.

A *consistent system state* is one in which, if the state of a process contains the receipt of a message, then the state of the corresponding sender also contains the sending of the same message [13]. Reaching a consistent state is the main goal of every checkpointing protocol when failures happen.

There are two main rollback-recovery approaches: *checkpoint-based* and *log-based*.

1) Checkpoint-Based: Checkpointing protocols require the processes to take periodic checkpoints with varying degrees of coordination. These protocols do not keep track of every interaction with the outside world, so, they do not guarantee that the execution being performed before a failure is exactly regenerated after a rollback.

With *uncoordinated checkpointing* each process is responsible for choosing the best time to take checkpoints. *Coordinated checkpointing* requires the processes to organize themselves in order to take checkpoints. Finally, with *communication-induced checkpointing* processes are responsible for taking their own checkpoints, and they are also induced to take semi-coordinated checkpointing by special information piggybacked on each message.

2) Log-Based: This type of rollback-recovery is more proper suited for applications that frequently interact with the outside world. These protocols allow for the logging of the determinants of nondeterministic events.

Pessimistic logging protocols log to stable storage the determinant of each nondeterministic event before the event is allowed to actually affect the computation. In *Optimistic logging*, however, there is the assumption that logging will complete before a failure occurs. *Causal logging* requires non-stable determinants to be piggybacked on messages sent to other processes, which can then be recovered after a failure.

D. Mobihoc

Mobihoc [3] is a middleware platform aimed at supporting the design of multiplayer distributed games for ad-hoc networks. It follows a client-server architecture, where one of the participating nodes must act as the server (it may also act as a client at the same time). The server has the role of providing write-lock management to the replicated objects, update propagation and enforcing the VFC model (section I).

With the exception of lock messages, the communication between clients and server is divided in rounds which are initiated periodically, and systematically, by the server. Each round the server propagates the object updates to the clients.

Through VFC, Mobihoc enables the optimistic replication of data, but it does not provide fault-tolerance to the system.

III. Fault-Tolerant Vector-Field Consistency

In our solution we propose a fault-tolerant Client/Server architecture that supports the design of distributed games for ad-hoc networks. We use Vector-Field Consistency (VFC) as our optimistic consistency model to reduce the bandwidth requirements imposed on both the users and the servers of the game. The version of VFC implemented by our system is an extension to the original model, designed to enable its execution when in the presence of dynamic entries and departures of nodes from the system. Furthermore, nodes also have the possibility to save their current game state in order to recover it later. For this reasons, we named our system as ”UbiVFC” , since it is ubiquitous in the sense that even when in the presence of failures, the server can keep being executed on any node.

A. Vector-Field Consistency Model

As was described in section I, VFC is an optimistic consistency model designed to manage replicated data across mobile devices executing a multi-player distributed game. The remainder of this section presents some details about this model.

1) Consistency Zones and Consistency Vectors: A consistency zone is a field around each pivot that resembles an electric or gravitational field. In the same way that a metal object is less attracted to a magnet as the distance between them increases, so do the consistency requirements of an object decrease as its distance to the pivot increases. Thus,

pivots generate consistency zones, concentric ring shaped areas, that enforce the same *consistency degree* to objects contained in the same consistency zone. Despite describing the consistency zones as ring shaped areas, they are actually implemented as concentric squares which improves the performance of the computationally expensive operation of determining the position of an object within a radial surface.

Consistency degrees are 3-dimensional *consistency vectors* $\kappa = [\phi, \theta, \nu]$. κ bounds the maximum divergence between an object in a particular zone and the value of its primary replica. Each dimension is a numerical scalar that defines the maximum divergence of replicated objects regarding the following metrics:

- *Time (θ)*: Specifies the maximum time an object can stay without being refreshed with its primary replica's latest value;
- *Sequence (ϕ)*: Specifies the number of updates an object can get without them being applied to its replicas;
- *Value (ν)*: Specifies the maximum divergence between the contents of the local copy of an object and its primary replica. This metric is application dependent since the objects are defined by the application programmers;

2) VFC Generalization: VFC also introduces two generalizations that allow a broader utilization of the VFC model: *multi-pivot* and *multi-view*.

The multi-pivot generalization enables the existence of more that one pivot on each view. It proves useful when there is the need to update more often two or more positions of a map, such as both the player's avatar and the flag on a "capture-the-flag" first-person shooter game. In a multi-pivot setup, an object's consistency zone is assigned with relation to its closest pivot.

The multi-view generalization enables different sets of objects to be defined with different consistency requirements regarding the same pivot. Using the same example as before, two different views can be used to define the consistency requirements of our player's team and the opposing one.

Despite this generalizations, only multi-pivots are implemented in our system. We use VFC as a single-view model.

3) Consistency Enforcement: The VFC model is enforced by a two-part algorithm, with each part being executed independently. The first part is executed by function *update-received* and the second part is executed by *round-triggered*:

- *update-received*: This function is executed each time a client makes an update to a replicated object. When the update is received, the number of missing updates to that objects is increased by one for every client on the system;
- *round-triggered*: This function is executed periodically by VFC to propagate object updates to clients according to their VFC settings. Each time it is executed it checks which objects are *dirty* and sends them piggybacked on round messages to the clients[1].

[1]An object is considered dirty to a client when it violates the consistency degree associated with the consistency zone it is located in.

B. System Architecture

Figure 1 presents the main components of our solution's system architecture. UbiVFC uses a client-server architecture. The Network Layer is responsible for establishing the connections between clients and server. It provides a generic interface that allows game programmers to implement any type of connection they require for their game. Clients and server possess different UbiVFC layers, in spite of some of the components bearing similar names. Just below the application layer, lies the API which is to be implemented by the game programmers to use the UbiVFC services. On the client side, the *Activity Manager* implements the services that are used by the server. On the server side, the *Notification Services* allow the the server application to acknowledge a series of events that may be important to the game programmers.

The main UbiVFC components are the **Consistency Management Block** (CMB), the **Session Manager** (SM), the **Object Pool** (OP), the **Membership Service** (MS), the **Failure Detector** (FD) and the **Checkpoint Recovery System** (CRS). Only the first three components of this list were also present in the original VFC design. The CMB and the MS are only available at the server, as opposed to the CRS, which only exists at the client. The remaining components are present on both the clients and server, presenting however, different characteristics.

The remainder of this section presents some details about the UbiVFC components on both client and server.

1) Session Manager: The SM is responsible for executing the protocol that enables the communication between clients and server. The *Server Session Manager* (SSM) implements the following services that process requests from clients: *subscribe*, *publish*, *enable*, *write*, *disable*, *leave*, *system info* and *pingpong*. This component manages all the interactions between itself and the other components available at the server. Moreover, the SSM is also responsible for triggering the rounds that are required for the execution of VFC. Each time a round is triggered, a new round update is broadcast to the clients. On the other hand, the *Client Session Manager* (CSM) does not only implement the services required to process requests from the server, but also contains the necessary functions that the application uses to communicate with the server. The services implemented by the CSM are the following: *round*, *enable*, *disable*, *new server* and *connect to new server*. Like the server's Session Manager, the client's one also manages the interactions between itself and the rest of the components in the client.

2) Consistency Management Block: This component is executed exclusively on the server and is responsible for the enforcement of the VFC model. Both functions mentioned in section III-A3 are in fact executed by CMB. This component provides a generic interface allowing UbiVFC to support different consistency models depending on the consistency requirements of the game programmers.

In order to enforce the VFC model, CMB aggregates the VFC consistency parameters specified by each client

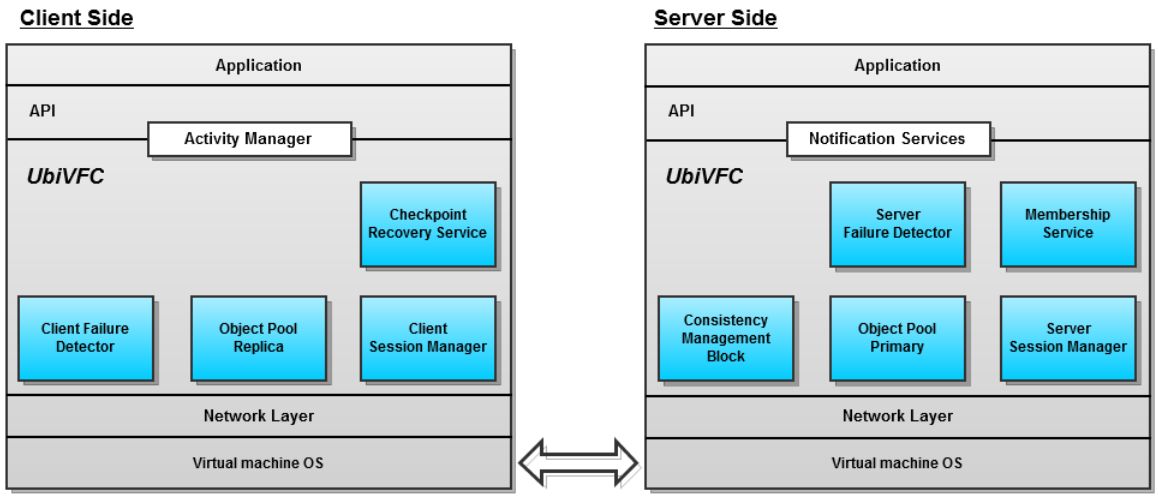

Figure 1. UbiVFC architecture.

(section III-A1).

Each time a client makes an update to a replicated object, the Server Session Manager dispatches it to the CMB where it is stored and the number of missing updates to that particular object are incremented. When a new round is triggered by the Session Manager, the CMB uses the VFC settings of each client to know if that object is to be propagated to said client. After making this check for each client and object on the system, it sends the updates to the Session Manager, which propagates them to the clients.

3) Object Pool: The OP is the repository of all game objects in the system. The server has the *Object Pool Primary* (OPP) that keeps all the most up-to-date objects, while clients have a mere replica.

The OPP is updated each time a client updates an object, and the server session manager is notified of that event through the *write* service. New objects can also be added to the Object Pool when a new client joins the system.

The *Object Pool Replica* (OPR) in the clients is updated each time a new round is triggered and the server's CMB finds new updates to propagate to that client.

4) Membership Service: The MS provides UbiVFC with a means to manage the game clients. It allows the addition and removal of clients from the system and chooses which clients can be the next server.

The clients elected to be the next server are known as *backup clients* and receive every new object update that the server receives, with no regards to their VFC settings. This ordered list of clients is subject to change at any given time. It can change when a new client joins the system, or when an existing one leaves. It can also change when a client sends the server updated information about its mobile device's remaining battery life (with the *System Info* service).

Receiving constant updates of clients' remaining battery life, the Membership Service is able to estimate the ones that are going to endure the execution of the game the longer, and elects them as backup clients.

5) Failure Detector: A FD is implemented on both the client and the server. The *Server Failure Detector* (SFD) sends a *ping* request to any client that fails to communicate with the server after a pre-defined number of rounds. If that client does not acknowledge the request after another pre-defined number of rounds, the SFD then notifies the SSM of the assumption that that client has failed.

The *Client Failure Detector* (CFD) only needs to periodically check the server for availability. Like the SFD, it only sends a Ping request after not receiving any message for a pre-defined number of rounds. When it detects a server failure it initiates the protocol to join a backup server, or to assume the server duties itself.

6) Checkpoint Recovery System: This component is responsible for periodically creating checkpoints of the current client system state, so that the client can re-join the game session later on, from the same state where it left-off.

Each time a new round message is received, the client's CRS checks for updates to its client's owned objects. Then, the objects that were updated that round are stored on their storage devices.

When a client re-joins a running game session, if it has recovery objects stored on its device, the client has the choice to recover the game session, or create a new one. Recovery objects are only available when the previous game has crashed, or the client has explicitly requested to save the game session when it previously left.

C. UbiVFC Protocols

This section presents the protocols that were implemented to accomplish the goals we set out to get. They are what distinguish the original VFC from our UbiVFC.

1) Client Subscribes: When a user wishes to join a game, if its client has recovery data available, the user is asked if he wants to recover the game state, or create a new one.

Either way, the protocol that is executed is presented on the remainder of this section. However, if the user wants to recover the previously saved game state, it waits until the end of subscribe protocol to automatically execute the publish protocol with the recovered data.

In order to join the game, the client must send a *subscribe* request to the server. The SSM acknowledges the client if the subscription was successful and then proceeds to take one of two actions that depend on what state the game is at: if the game is not yet started, the SSM adds the client to the MS, which then proceeds to register it in the system; if the game is already active, the SSM adds the client to a subscription waiting queue where it stays until a *publish* request is received from that same client.

2) Client Publishes: When a client sends a *publish* request to the server, the SSM sends back a response of acknowledgment if the request was completed successfully. In that case, the server can do one of two things, depending on the state it is in:

1) If the game has not yet started, the SSM hands the objects piggybacked on the *publish message* to the OPP for storage, and notifies the CMB of the new arrivals. It then propagates the new objects to the remaining clients, and the rest available in the OPP to the new client;
2) If the game is already running, the SSM adds the client to a *publishing waiting queue*. When a new round is triggered, the SSM checks the queue for entries, and proceeds to execute the protocol described on the previous point. Then, the SSM sends an enable request to the client so that it can initialize the game.

3) Client Leaves: When a user wants to leave the game, it is presented with an option to save the current game state so that it can be resumed later on. Either way, it must send a *leave* request to the server in order to do so in an orderly fashion. If the game has not yet started, the SSM simply asks the MS for the removal of that client's personal data and notifies the remaining clients of the departure. If the game is already underway, the SSM puts the leaving client's request on a *leaving waiting list*. When a new round is triggered, the server processes each client on that waiting list, first notifying the remaining clients of the departure, and then by asking the MS to remove that client's information and objects from their respective structures.

4) Client Fails: The Server Failure Detector periodically checks if a client has failed by sending it *ping* messages, when they fail to communicate for a certain amount of time. Therefore, on the event of a client failure, the server has a certain protocol that it must follow to allow the game to keep running.

Each round trigger, the SFD checks if there was a client that has not communicated with the server for a certain amount of time. So, if λ has elapsed without the server getting any message from a certain client, it sends a *ping* message to that client. If another λ time has gone by without receiving a *pong* response, or any other type of message, the SFD notifies the SSM of this event. The SSM then broadcasts the departure information to every other client, and notifies the MS and the CMB to remove any data that belonged to the failed client.

5) Server Leaves: When a server wants to leave the game, the server role must be assumed by a client previously chosen as backup (section III-B4). Each round, the SSM sends the backup client all the updates received that round, and some optional backup data[2]. The protocol that is followed when the server wants to leave the game is as follows:

1) The server sends a *new server* request to the backup client;
2) The backup client notifies the leaving server that it is starting the new server;
3) When the server receives the acknowledgment from the backup client, it notifies the remaining clients with a *connect to new server* request;
4) When the remaining clients acknowledge the departing server, it can safely shutdown;
5) The new server waits for the connections of the remaining clients, and when all clients have re-joined, the new server resumes the game.

In order for this protocol to work on a real-life environment, a set of backup measures were implemented:

- The new game server waits a certain amount of time for the re-joining of the clients. If any clients fail to re-join the game, the server automatically enables it, removing the failed clients from the game.
- The clients that are connecting to the new server have a certain amount of attempts to do so, before they assume that the new server is not available.
- If the backup fails to start the server and if there is another backup among the clients attempting to re-join the game, that backup assumes the server role itself, and the remaining clients attempt to connect to the next backup elected by the previous server. The clients iterate over all the backups on the game until they find one that is available. If no backup is available, the client quits the game.

6) Server Fails: The CFD works in a similar fashion to its server counterpart, in order to detect server failures. It waits λ time until it sends a *ping* request to the server, if no message has been received during that time. If after another λ time, the client has received no response from the server, it initiates the game resuming protocol, which shares some resemblance with the one explained on the previous section III-C5.

1) The clients detect that the server has failed;
2) The first backup client assumes the server duties;
3) The remaining clients connect to the new server;
4) The new server enables the UbiVFC server, resuming the game where it left-off when the previous server failed.

[2]This optional backup data is only sent when there is a change in membership or a client sent some updated battery information

Like was presented in the previous section, the same backup measures can be taken if another failure is detected when this protocol is being processed.

D. Supporting Data Structures

Table I presents a summary of the most important data structures implemented in our solution. They are clearly divided by their provenience, that is, if they were adapted from the original VFC implementation, or if they were implemented just for UbiVFC. The last table column presents the corresponding data structure type.

As the table shows, much more data structures were created just for UbiVFC than adapted from the original VFC. However, many of these are empty on most situations. For instance, every structure in the Client Session Manager except *backupUsers* is only available at backup clients. Moreover, the *clientsLeaving*, *clientsSubscribing* and *clientsPublishing* hash tables at the Server Session Manager only have mappings when clients are leaving, subscribing or publishing, respectively. Furthermore, the *clientsRemainingToResume* list in the same component, only has strings when a server is assuming another server's role. Finally, the *ServerSyncData* object is only used to transfer backup data.

IV. Evaluation

In order to test our Fault-Tolerant Vector-Field Consistency system, we decided to develop a multi-player version of the Snake[14] game that did not require user intervention to be played, and named it **Snakes VFC**. Each player starts with a *snake* that randomly wanders the map searching for *apples*. On the contrary of the original version, our snakes do not grow or start to move faster, as apples are found. However, with each found apple, the player adds a point to his *score*.

Each snake only moves one time per round. The snake and score are *pivot* objects owned by the clients while the apples are owned by whatever player is the server at the moment. Each player uses default VFC settings. This settings are constituted by two consistency vectors that are defined for two possible consistency zones. The first consistency zone is the one that is comprised inside a 10 tile[3] radius of the pivot. The second zone is the area beyond the 10 tile radius. For the first zone we have a consistency vector κ = [2,2,2], while for the second zone we have a consistency vector of κ = [.,10,5][4].

A. Quantitative Evaluation

1) Amount of Messages Exchanged: The first tests that were performed on our solution are related to the amount of messages that are exchanged between clients and server. We executed the game for 100 rounds and measured how many messages were sent by the clients to the server, and how many messages were sent by the server to the clients using three different consistency models: *non-VFC*, *VFC* and *UbiVFC*. We ran this tests with 2, 3 and 4 clients, separately.

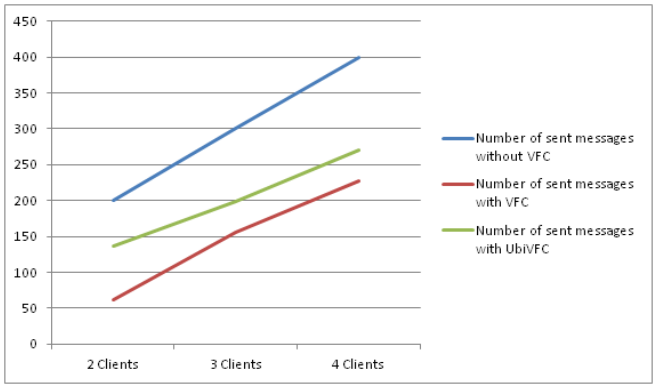

Figure 2. Comparison between the number of messages sent from all three models.

Since the amount of messages received by the server is roughly the same using the three consistency models, in figure 2 we compare the number of messages sent. As we can see, the number of messages sent without VFC (represented by the blue line) constantly increases as the number of clients also increase. The other two lines (green representing UbiVFC and red representing VFC) also constantly increase, but they start with much lower values and have a much lower slope. The UbiVFC line seems to get closer to the VFC one as the number of clients increases.

The number of messages sent by the server using our UbiVFC solution is higher than the original VFC implementation. However, it still is much lower than not using VFC or UbiVFC. For the additional functionality that UbiVFC brings, wee feel that this increase in messages sent is low enough to still present a real viable solution.

2) Time Elapsed: The second tests that were performed on UbiVFC are related to the amount of time the protocols described in section III-C take to be executed. We compared the results to the time it takes to perform other critical protocols and algorithms in our system. As in the first battery of tests, we also ran this with 2, 3 and 4 clients, separately.

Figure 3 presents a comparison between the average times UbiVFC takes to execute different protocols and algorithms when there are three clients in the game.

UbiVFC takes on average **0.01** seconds to compute the updates it must propagate to one client. Since the backup client always gets every update available, UbiVFC does not need to compute the updates to send to it, so no time is spent. To actually send an update to a client, our solution usually takes around **0.1** seconds. This algorithm and protocol are based on the original VFC implementation.

When there is a change in membership, and a new client is selected as backup, a *ServerSyncData* object must be sent to that client with all the backup data required. On average, it takes **0.408** seconds to send the backup data and the round updates to the backup client. If a client joins the game, but the backup client stays the same, only some partial backup

[3]A tile is an image that represents an object in the game map. For instance, an apple in our game is a tile.

[4]The "." means that this vector imposes no constrains regarding the number of rounds that have passed since this replica was last updated

	Adapted	New	Type
Fundamental	DataUnit		*DataUnit*
	UserAgent		UserAgent
Object Pool	DataPool		*DataPool*
CMB	timesUpdated		*Map¡UserAgent, List(int)¿*
	lastUpdates		*Map¡UserAgent, List(DU)¿*
	updateQueue		*UpdateQueue*
Server Session Manager	phi		*Map¡UserAgent, Phi¿*
	duAssociation		*Map¡int, UserAgent¿*
		clientsLeaving	*Map¡int, ClientLeaving¿*
		clientsSubscribing	*Map¡int, UASIPair¿*
		clientsPublishing	*Map¡int, Array¡DU¿*
		resuming	*boolean*
		clientsRemainingToRes	*List(String)*
Client Session Manager		registeredUsers	*Map¡int, UserAgent¿*
		duAssociation	*Map¡int, UserAgent¿*
		phi	*Map¡UserAgent, Phi¿*
		usersURLs	*Map¡String, String¿*
		connectionType	*int*
		cliIdCount	*int*
		ObjectIdCount	*int*
		backupUsers	*Array¡String¿*
Membership Service	registeredUsers		*Map¡int, UserAgent¿*
		usersURLs	*Map¡String, String¿*
		clientsOrdered	*List(BatteryRecords)*
		backupUsers	*List(UserAgent)*
		previousBackupUsers	*List(UserAgent)*
Client Failure Detector		lastNewsFromServer	*long*
		pingSent	*boolean*
Server Failure Detector		lastNewsFromClient	*Map¡int, ClientNews¿*
Checkpoint Rec Sys		ownedObjects	*Array¡int¿*
		objectsToBackup	*List(int)*
Other		ServerSyncData	*ServerSyncData*

Table I
SUMMARY OF SUPPORTING DATA STRUCTURES IMPLEMENTED IN UBIVFC.

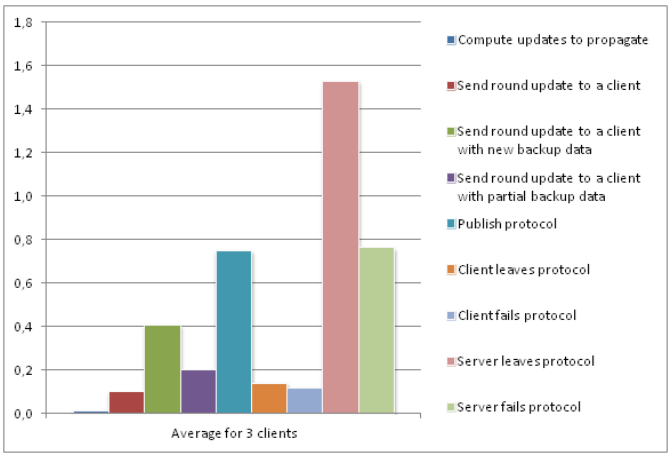

Figure 3. Comparison between the time it takes to execute each protocol or algorithm in seconds.

data must be sent to that client, which on average takes **0.2** seconds to finish (including the round update data).

The *client publishes* protocol usually takes **0.751** seconds to complete, the *client leaves* protocol takes on average **0.140** seconds, the *client fails* **0.120** seconds, the *server leaves* protocol takes around **1.531** seconds and finally the *server fails* protocol takes on average **0.763** seconds to complete.

Examining the figure, we can plainly see that the three protocols that require more time to complete are *publish*, *server leaves* and *server fails*. The last two take this much time because they are complex protocols that require the termination and creation of new connections in order to resume the game on another server. The *publish* protocol, however, requires this much time because of the amount of objects that must be diffused among the clients. Both the *compute updates to propagate* algorithm and the *send round update to a client* protocol need much less time to complete than this three protocols. However, this last two processes are performed every round and as many times as there are clients, which does not happen with the three former protocols. The *publish* protocol is only executed when a client joins the game when it is already running, *server leaves* is executed when a server leaves in an orderly fashion and *server fails* is only executed when a server leaves in a disorderly way.

B. Qualitative Evaluation

We cannot end the evaluation of our solution without considering it from the eyes of the user. A user only wants to play the game without problems, and does not care if a server has failed, or how much time it takes for a round of updates to be propagated to every client. For this reason we tested our solution not only on android emulators[15] but on real devices as well. The execution of the game on real

devices and on a real wifi network is a much smoother and faster experience. This comes with no surprise since android emulators (as well as other mobile devices' emulators) are known to be "resource hogs".

When a user leaves or joins the game, all other players are notified of that event by a *popup* message informing the players of membership changes, without being obtrusive.

When a client crashes, the remaining users of the game barely notice that event, since in one round the player's snake is there, and the next it is not. However, when a client fails by stopping communicating with the server, until the SFD acknowledges this event, the failed client's snake lingers in the game. The time elapsed between a client's failure and the SFD finding it can, however, be tweaked by the game programmers. If the time it takes for a FD to assume that a client as failed is lower, the game experience is better for the user, however, it is also easier to detect false-positives.

As it happens when a client crashes, when a server does, the remaining clients automatically adjust to that situation, electing a new one and only allowing the users to barely notice that event. On the other hand, when a server fails by stopping communicating with the clients, the users will take notice. Since it is the server that triggers the rounds, if the clients do not receive a round update, their game pauses. The CFDs wait the number of rounds they are programmed to wait, until they assume that the server has crashed. As it happens with the SFD, the amount of time the client's must wait to take over a failed server can be changed by game programmers.

Finally, when a game crashes, the possibility of recovering the game session is very important for the user. For this reason, we made it really simple to recover a crashed game. When a user tries to connect to a game server, if UbiVFC detects that there is recovery information on the mobile device's storage, the user is presented with a *dialog* that asks if he wants to recover the game session. If he does, his game session is recovered in the same state that it was before the crash, if he does not, a new game state is launched.

V. Conclusions

In recent years, we have witnessed a proliferation of mobile devices, which lead to an increased supply of applications designed specifically for this kind of environments. Many of this applications are multi-player games that require that two or more devices are connected to each other. In this type of games there is a great need for data exchange between nodes and since mobile devices have small processing power, and the ad-hoc networks they form are subject to high latency and reduced bandwidth, game playability may be hindered, and the devices batteries drain faster.

Vector-Field Consistency (VFC) is a consistency model that reduces network usage by selecting critical updates to propagate to replicas (see section III-A). It is enforced by Mobihoc, which is a middleware platform for multi-player distributed games in ad-hoc networks (see section II-D). Even though this model was developed for this kind of networks, it does no support the spontaneous entries and departures (orderly or disorderly) that are so common in this environments. In this work we proposed to extend VFC in order to support the dynamic entry and departures of nodes from the system.

In this document we have discussed how current approaches, both commercial and academic, try to enable constant membership changes and ensure availability when in face of node failures. We focused on replication, fault-tolerance and rollback-recovery because we consider this three areas to be the most relevant to the achievement of our goals.

We proposed UbiVFC, a fault-tolerant VFC that enables the execution of the VFC model in the presence of node failures. Our solution allows clients to join and leave the game when they want and to recover their game session data if they want to re-join a game they were playing. Furthermore, it also allows the game server to leave or crash at any moment, ensuring that the game keeps being executed on another host. To create this solution we used replication, fault-tolerance and checkpointing techniques such as: passive and optimistic replication, membership services, failure detectors and coordinated checkpointing.

To test our system we developed a multi-player distributed game called Snakes VFC. This game is based on the old Snake game, much popular among late-nineties Nokia phones owners. We used this game to compare the amount of messages that are exchanged between nodes using VFC, not using any consistency model and using our own UbiVFC. We also compared the time it takes to execute the protocols that allow UbiVFC to achieve its goals, in the presence of a different number of users. Our results show that even though the amount of messages that are exchanged between nodes is bigger using UbiVFC than VFC, it still is less than two thirds the amount of messages not using a consistency model. Furthermore, the time it takes to execute critical system protocols continues to be small.

Acknowledgments

This work was partially supported by the Mercury project and by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds.

References

[1] P. F. Ferro, E., "Bluetooth and wi-fi wireless protocols: a survey and a comparison." *IEEE Wireless Communications*, vol. 12, no. 1, pp. 12–26, 2005.

[2] R. Rajaraman, "Topology control and routing in ad hoc networks: a survey." *ACM SIGACT News*, vol. 33, no. 2, 2002.

[3] V. L. F. P. Santos, N., "Vector-field consistency for ad-hoc gaming." *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, 2007.

[4] D. J. Coulouris, G., "Distributed systems: concepts and design." 2005.

[5] L. Lamport, "Time, clocks, and the ordering of events in a distributed system." *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[6] Oracle, *Oracle7 Server Distributed Systems Manual.*, 1996, vol. 2.

[7] D. Dietterich, "Dec data distributor: for data replication and data warehousing." *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, p. 468, 1994.

[8] S. M. Saito, Y., "Optimistic replication." *ACM Computing Surveys (CSUR)*, vol. 37, no. 1, pp. 42–81, 2005.

[9] D. A. P. K. S. M. T. M. W. B. Terry, D., "Session guarantees for weakly consistent replicated data." *Proceedings of the third international conference on on Parallel and distributed information systems*, pp. 140–150, 1994.

[10] K. I. V. R. Chockler, G., "Group communication specifications: a comprehensive study." *ACM Computing Surveys (CSUR)*, vol. 33, no. 4, pp. 427–469, 2001.

[11] T. S. Chandra, T., "Unreliable failure detectors for reliable distributed systems." *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

[12] A. L. W. Y. J. D. Elnozahy, E. N., "A survey of rollback-recovery protocols in message-passing systems." *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375–408, 2002.

[13] L. L. Chandy, K., "Distributed snapshots: determining global states of distributed systems." *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.

[14] E. Koivisto, "Mobile games 2010." *Proceedings of the 2006 international conference on Game research and development*, pp. 1–2, 2006.

[15] Google, "Android development tools," October 2011. [Online]. Available: http://developer.android.com/sdk/eclipse-adt.html