INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

# Fault-Tolerant Vector-Field Consistency

**André Donato Salomão dos Santos**

Dissertação para obtenção do Grau de Mestre em
**Engenharia Informática e de Computadores**

**Júri**

| | |
|---|---|
| Presidente: | Professor Doutor Pedro Manuel Moreira Vaz Antunes de Sousa |
| Orientador: | Professor Doutor Paulo Jorge Pires Ferreira |
| Co-Orientador: | Professor Doutor Luís Manuel Antunes Veiga |
| Vogal: | Professor Doutor Manuel João Caneira Monteiro da Fonseca |

**Outubro 2011**

# Acknowledgements

I would like to thank Prof. Paulo Ferreira, my MSc advisor, for the support, orientation and motivation throughout the elaboration of this work. His advice and constructive criticism contributed greatly for the quality of this thesis.

To my MSc colleagues and friends, Mário, André and Carlos and Pedro for their support and fellowship this past three years.

Finally, to my family, for their immense support.

Lisboa, November 25, 2011
André Santos

To everyone I rely on, thank you!

# Resumo

Recentemente tem havido um crescimento exponencial de jogos em dispositivos móveis. Não é fácil desenvolver jogos multi-jogador para redes ad-hoc devido aos problemas inerentes aos dispositivos móveis e ás prṕrias redes ad-hoc, tais como ligação limitada, baixa capacidade de processamento e pouco tempo de vida das baterias. Vector-Field Consistency é um modelo de consistência optimista que reduz a utilização da rede, seleccionando apenas as actualizações importantes a propagar para as réplicas. VFC é executado através de uma plataforma middleware, Mobihoc, cujo objectivo é facilitar o desenvolvimento de jogos distribuídos multi-jogador para redes ad-hoc. Neste trabalho extendemos o modelo VFC de forma a suportar a entrada e saída de nós do sistema, ou seja, criamos um VFC tolerante a faltas.

# 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.

# Palavras Chave
# Keywords


## Palavras Chave

Vector-Field Consistency

Tolerância a faltas

Replicação

Checkpointing

## Keywords

Vector-Field Consistency

Fault-Tolerance

Replication

Checkpointing

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Since the arrival of the first mobile devices, their use has been constantly growing, in such ways that nowadays, everyone has got at least one. The most commonly used device is the cell phone, but PDAs and netbooks are also widely spread. These devices come equipped with technologies that allow the exchange of data with the outside world, such as infra red, bluetooth and WiFi [13]. The widespread of these technologies is the main reason why wireless ad-hoc networks have become so popular. This kind of networks are formed spontaneously between two or more devices communicating via wireless interfaces. An ad-hoc network is a decentralized wireless network that does not rely on any pre-existing network infrastructures, such as routers in wired networks or access points in managed wireless networks [33].

With the proliferation of mobile devices, the number of applications specifically developed for this kind of environments has also been increasing. Furthermore, and mostly because of the deployment of Snake on every Nokia phone, the development of games for mobile devices is known to be a very appealing industry [23]. At present time, the mobile devices game market has got a huge offering. The games available can be as simple as a Solitaire card game, or as complex as an internet oriented rally simulation game. With the advent of the ad-hoc networks, people can even play distributed multiplayer games wherever they are (e.g. public transports, restaurants), despite the unavailability of a structured network or without incurring into any connectivity expenses.

In distributed multiplayer games, there is a very high need for data exchange between network nodes. Player positions, maps or scores need to be updated constantly on every node. In order to keep distributed data consistent, additional communication is required for synchronization operations and update propagation. In ad-hoc networks, this kind of applications that require constant communication between network nodes, 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.

Allied to the already mentioned problems, the sole nature of ad-hoc networks themselves suggest a constant change of the network topology. The number of devices connected to an ad-hoc network can change in an instant. New nodes can join the network or current members can leave. This node departures can be premeditated or not. One mobile device can get out of range from the other network nodes, forcing the connection to drop. Any node's network interface can fail. Furthermore, and particularly when executing a communication and processor intensive application like distributed network games, the

devices batteries can run out.

Vector-Field Consistency (VFC) [36] 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 Vector-Field Consistency model enables the selection between critical object updates, that must be propagated as soon as possible, and the ones that can be queued for later dissemination. This ability translates into a much more efficient usage of the mobile devices resources. Most of the network bandwidth is saved and the high latency of ad-hoc networks goes unnoticed.

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 Vector-Field Consistency 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. This problem is common to all asynchronous networks. Another obstacle that must be faced is how can the system state be stored persistently in order to recover it later. 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. In chapter 2 we describe in detail the main concepts which are related to our work, and review several commercial and academic works that share our own goals. Chapter 3 describes the architecture of our system and chapter 4 presents the details about its implementation. In chapter 5 we describe the evaluation environment and present the results obtained, analysing them at

the same time. Finally, chapter 6 summarizes the work performed, introduces our ideas for future work and draw some conclusions.

# Chapter 2

# Related Work

This chapter overviews the main concepts about the topic at hands. The first three section describe replication, fault-tolerance and checkpointing respectively. After that, we discuss these concepts, with regards to our own work. Finally, a few commercial and academic systems are reviewed, which focus on the same topics as ours.

## 2.1 Replication

In distributed systems, data replication is a key technology to achieve enhanced performance, higher availability and fault tolerance [9]. It consists on the maintenance of copies of data across multiple computers. This technology can bring many benefits to a variety of systems, and for this reason, it is widely used. For instance, the DNS naming service maintains copies of the name-to-address mappings over many servers [27]. If these mappings were not replicated, this service would not be usable by the amount of clients which are constantly accessing it over the world wide web.

Data replication can greatly increase a system's performance through a variety of ways. Copies of web resources can be cached by a browser, and proxy servers, to avoid the latency of further accesses to the same website. Furthermore, a web domain can replicate data over several servers. This way, the workload of the domain can be shared between those servers, and a client can choose the nearest server to connect to.

A 100% available service is the requirement of every user. Unfortunately, it is not possible to achieve, but data replication contributes highly to a close to 100% available service. If some service's data is replicated over two or more failure-independent servers, a client can still use the service even if one of them fails.

When data replication technology is used in a service, there are two common requirements. First, the fact that the service is using replicated data must be transparent. That is, a client should not be aware that there are multiple physical copies of data in the servers. Second, a certain degree of data consistency between all the replicas must be achieved. This consistency degree may vary with each application.

There are several ways of analyzing the correctness of replicated objects. One of such ways is by using the *linearizability* property [21]. A system is said to be *linearizable*, if at any given time of its execution,

| Client 1 | Client 2 |
|---|---|
| setBalance$_B$(x,1) | |
| | getBalance$_A$(y)→0 |
| | getBalance$_A$(x)→0 |
| setBalance$_A$(y,2) | |

Table 2.1: Sequencial consistency example.

the interleaving of the operations issued by all the clients is consistent with the real times at which those operations were requested. If the interleaving of operations is only consistent for each client individually, and not for all the clients, the system is said to be *sequentially consistent*.

For instance, consider a replication system with two replica managers at computers $A$ and $B$, that maintain replicas of two bank accounts $x$ and $y$. Replica managers propagate updates to one another in the background after responding to the client. Table 2.1 presents an example execution of both clients requests. In this example the real-time criterion for linearizability is not satisfied, since getBalance$_A$(x)→0 occurs later than setBalance$_B$(x,1), but *Client 2* does not see the update *Client 1* made to the variable $x$, because computer $B$ is yet to propagate the update to computer $A$. However, the interleaving presented satisfies the criteria for sequential consistency, because each client sees the correct interleaving of the operations he performed.

### 2.1.1 Passive Replication

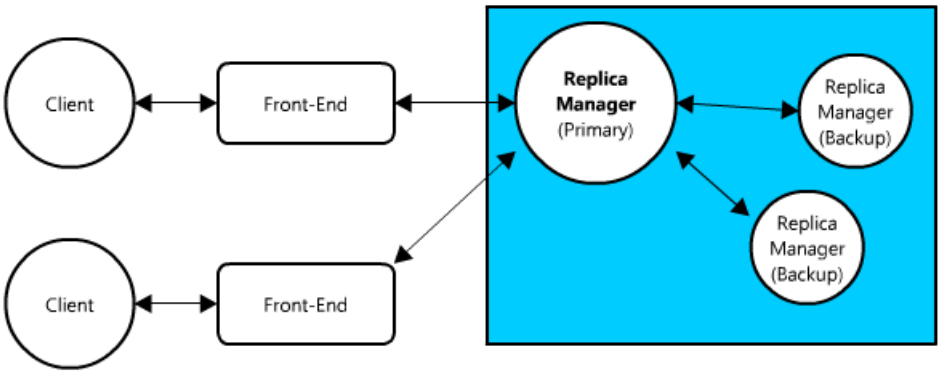


Figure 2.1: Passive replication model.

In the *passive*, or *primary-backup*, model of replication for fault tolerance, there is, at all times, only one replica manager answering client's requests (see Figure 2.1). This replica manager is known as the primary replica manager. All other replica managers available in the system only serve as backups.

In the pure form of the model, the sequence of events when a client requests an operation to be performed is as follows:

1. *Requests:* The front-end issues the request, containing an unique identifier, to the primary replica manager;
2. *Coordination:* The primary replica manager processes each request atomically and in the same order of arrival. If it has already processed a request with the same unique identifier, it simply re-sends the response;

3. *Execution:* The primary executes the request and stores the response;
4. *Agreement:* If the request is an update, the primary sends the updated state, the response and the unique identifier to all the backup replica managers. The backups reply with an acknowledgment;
5. *Response:* The primary responds to the front-end, which hands the response back to the client.

Since each client only issues requests to a single replica manager (the primary one), this system implements linearizability if the primary is correct. On the event of the primary failure, if a single backup replica manager takes over its place, and the new system configuration picks up exactly where the last left off, linearizability is still maintained.

However, if the primary is taken over by two (or more) backups, and if the remaining backups do not agree on the operations performed prior to the primary failure, the system could perform erroneously.

Both these issues can be avoided if the replica managers are organized as a group, and if the primary uses view-synchronous group communication to transfer the updates to the backups (see section 2.2.4). The first issue is easily solved because, when the primary crashes, the communication system installs a new view to the remaining replica managers, which does not include the old primary. After the new view is delivered, the backups can then reach a consensus about which one will be elected primary. The second issue is also easily solved by the ordering property of view-synchrony and the use of stored identifiers to detect repeated requests. When a new view is delivered, either all the backups received a specific update, or none of them did.

### 2.1.2 Active Replication

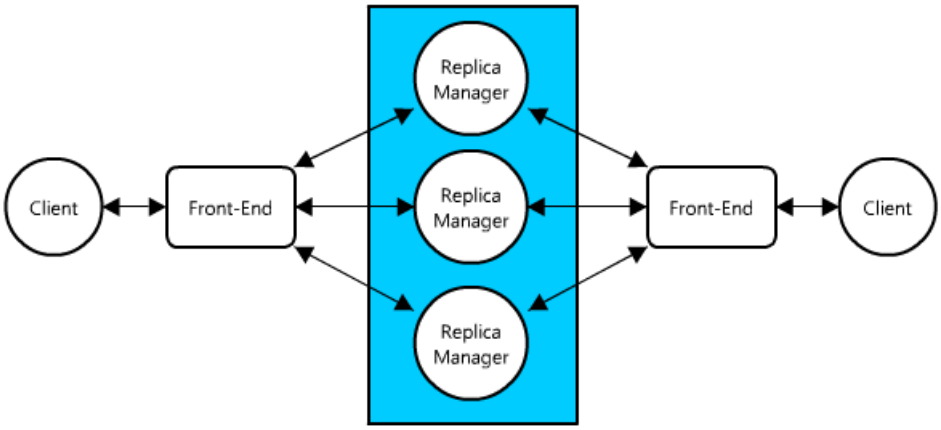


Figure 2.2: Active replication model.

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 in what is probably the most cited article on the theory of distributed systems, as state machine replication [25]. 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. Moreover, this model prevents against byzantine failures in the sense that the front-ends can collect and compare every reply they receive from the replica managers.

Under active replication, the typical sequence of events that happen after a client performs a request is as follows:

1. *Request:* After attaching a unique identifier to the client request, the front-end multicasts it to the group of replica managers, using a totally ordered, reliable multicast primitive. Front-ends can only fail by crashing at the worst, and must wait for a reply before issuing another request;
2. *Coordination:* The group communication system delivers the current request to every correct replica manager in the same order;
3. *Execution:* Every replica manager executes the request. Since every replica manager works as a state machine, and since all requests are totally ordered, every correct replica manager produces the same output for the operation at hand. The response contains the client's unique request identifier;
4. *Agreement:* Because of the multicast delivery semantics, no agreement phase is necessary;
5. *Response:* Finally, each replica manager sends its response to the front-end. The front-end can then send the first response that arrives to the client, depending on the failure assumptions and on the multicast algorithm in use.

Since all correct replica managers process the same operations in the same order, because of the reliability of multicast and total order, this system is able to achieve sequential consistency. Another condition that enables this kind of systems to achieve sequential consistency is the fact that every replica manager is a state machine and, as such, they all produce the same output.

However, this model is unable to achieve linearizability. The reason being that the order of the operations being performed by the replica managers, might not be the same as the real-time order in which the clients made their requests.

### 2.1.3 Pessimistic Replication

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 [30, 10].

Good performance and high availability cannot be expected from pessimistic replication on wide-area networks, such as the Internet, for three key reasons:

- The internet is slow and unreliable. It's communication latency and availability is not expected to improve radically. Moreover, smartphones, mobile computers, and other mobile devices with intermittent network connectivity are widespread. A pessimistic replication algorithm, trying to synchronize with an unavailable internet service on one of these devices, might block indefinitely when connectivity is lost;
- Pessimistic algorithms scale badly in wide area networks. As the number of servers increase, it is also increasingly difficult to maintain every replicated object up to date, because the throughput and availability suffer;

- Finally, some applications simply cannot function under pessimistic replication conditions. Cooperative engineering or software development often requires people to work offline. A more flexible model is needed that allows the relaxation of synchronization procedures and that provides conflict resolving.

### 2.1.4 Optimistic Replication

Optimistic replication algorithms allow the access to replicated data without *a priori* synchronization, based on the "optimistic" assumption that problems will rarely occur [35]. 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. Moreover, optimistic algorithms enable sites and users to become autonomous, for instance, replicas can be added with no change needed to existing sites. Finally, they enable asynchronous collaboration between users, and can provide immediate feedback as they can apply updates tentatively as soon as they are submitted. 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 which are analysed in the remainder of this section:

#### Number of Writers

Like as been shown in section 2.1.1 and section 2.1.2, a replication system can be *single-mastered* or *multi-mastered*. A single-master system as only, and at all times, a single designated writer. All updates originate at the master, and are then propagated to the backups. This kind of systems are simple but have limited availability, especially when the system experiences many updates on a short amount of time. In a multi-master system, updates can be submitted at multiple sites, and are then propagated on the background to the other replicas. This kind of systems have an increased availability, but can only achieve this at the cost of being more complex. Multi-master systems can only remain eventually consistent if they can solve issues regarding operation scheduling and conflict management.

#### Definition of Operations

Regarding the definition of operations, a replication system can be *state-transfer* or *operation-transfer*. The former case is much more simple, because when a replica object is updated all it takes to update all replicas is to send them the updated object. In the latter case, when an object is updated, it is the operation that transformed the object that is propagated to the other replicas, not the object *per se*. An operation-transfer is more complex because an ordered operation update history is required to maintain replica consistency. However, because state-transfer systems require that the complete object which was updated is propagated to other replicas, this kind of systems are more inefficient regarding network traffic. Furthermore, operation-transfer systems also allow for a finer-grained conflict resolution.

### Scheduling

Scheduling policies are an important part of optimistic replication systems. Through them, replicas can maintain a consistent state which can be observed by users. There can be two types of scheduling policies: *syntactic* and *semantic*. On the one hand, syntactic policies are easier to implement, because they sort operations only with regards to three factors: time, place or author. Timestamp-based ordering is the most widely known example. On the other hand, semantic policies can solve many conflicts automatically, because they exploit semantic properties, such as commutativity or idempotency of operation. Semantic scheduling can only be used in operation-transfer systems, since state-transfer systems do not keep track of the operations executed on the replicated objects.

### Handling Conflicts

In optimistic replication, conflicts can happen when updating objects replicated in the system. For this reason, there are several ways to handle them. The best approach is to prevent them from happening altogether. In a pessimistic replication system this is easily accomplished by blocking or preventing operations as necessary. Single-master systems accomplish the same goal by only allowing updates to be made at a single replica. Both these types of systems accomplish this, however, by reducing availability. Some systems simply ignore conflicts. If a pending operation is potentially conflicting with another, one of them is simply overwritten.

If a system can detect conflicts, the user experience is improved. Like the scheduling policies, conflict detection is also divided into *syntactic* and *semantic* policies. Where semantic conflict detection systems reduce conflicts by providing application specific policies, such as allowing the reservation of the same room at two different times in a room-booking application, syntactic conflict detection policies rely more on the timing of the operations for conflict detection, and are not application-specific.

### Propagation Strategies and Topologies

At some point in time, every replica must send its local operations to the other replicas present in the replication system, in order to stay consistent with each other. The way this transfer of operations is performed can be classified along two different axes: communication topology and the degree of synchronization.

Fixed topologies such as star or semi-structured can be very efficient but perform poorly in dynamic, failure prone scenarios. At the other end of the spectrum, many optimistic replication systems rely on ad-hoc topologies which transfer operations through the network in a epidemic fashion, and perform exceptionally well in dynamic scenarios.

Regarding the degree of synchrony, systems can be *pull-based* or *push-based*, or a hybrid of these two. In a pull-based system, each replica checks the other replicas periodically for updates. In a push-based system, when a replica has got new updates, it proactively sends them to the other available replicas. If there is a constantly high rate of operations being propagated through the system, the number of conflicts and the degree of replica inconsistency is lower, however the complexity and overhead will be greater.

**Consistency Guarantees**

An optimistic replication system is prone to some divergence between each replica's state. Consistency guarantees define how much divergence a client application may observe. The strictest guarantee is single-copy consistency or *linearizability* which was previously defined at the beginning of section 2.1. At the other end of the spectrum is *eventual consistency*, which only guarantees that the state of replicas will eventually converge. It is a fairly weak concept, as clients can witness arbitrarily stale state, or even incorrect state, but is the guarantee most optimistic replication systems employ.

In between linearizability and eventual consistency policies lie *bounded divergence* policies. Bounded divergence is usually achieved by denying access to replicas that do not meet certain consistency conditions. *Session guarantees* present a mechanism which allows individual applications to have a view of the database that is consistent with their own actions [40]. Four per-session guarantees are presented:

- *Read your writes* guarantees that the contents read from a replica incorporate previous writes by the same user;
- *Monotonic reads* guarantees that successive reads by the same user return increasingly up-to-date contents;
- *Writes follow reads* guarantees that a write operation is accepted only after writes observed by previous reads by the same user are incorporated in the same replica;
- *Monotonic writes* guarantees that a write operation is accepted only after all write operations made by the same user are incorporated in the same replica.

These guarantees are sufficient to solve a number of real-world problems. They are implemented using a *session* object carried by each user. A session records two pieces of information: the *write-set* of past write operations submitted by the user, and the *read-set* of writes that the user has observed through past reads. Each of them can be represented in a compact form using vector clocks [14].

## 2.2 Fault-Tolerance

The previous section 2.1 described data replication strategies that are, among other things, intended to provide systems with fault-tolerance. That is not however its main goal. *Group communication* is a technology which main purpose is to provide replication systems with fault-tolerance [8].

### 2.2.1 Multicast Communication

Group, or *multicast*, communication requires coordination and agreement. With multicast communication a group of processes is able 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. A multicast is different from a *broadcast* in the sense that while a multicast enables a process to issue only one multicast operation to send a message to a group of processes, a broadcast allows a process to issue multiple send operations to all the system processes. The use of a single multicast operation instead of multiple send operations enables the implementation to be efficient and allows it to provide stronger delivery guarantees:

- *Efficiency:* Issuing a single operation to send a message to a group of processes allows for an efficient use of bandwidth. It can arrange to only send a single message over any communication link, by sending the message over a distribution tree. It can also take advantage of network hardware that supports native multicast support. Finally, it can minimize the amount of time necessary to deliver the message to all processes transmitting it together in parallel.
- *Delivery guarantees:* With multicast communication it is possible to attend that a set of messages is delivered in order and with reliability. If the messages were sent independently this would not be possible.

**Multicast Reliability and Ordering Semantics**

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* is a multicast guarantee that satisfies the three following properties [5]:
  - *Integrity:* A correct process $p$ delivers a message $m$ at most once;
  - *Validity:* If a correct process multicasts message $m$ then it will eventually deliver $m$;
  - *Agreement:* If a correct process delivers message $m$, then all other correct processes in *group(m)* will eventually deliver $m$.
- *Ordered multicast* satisfies the same properties as basic and reliable multicast, and introduces an ordering guarantee to the delivered messages:
  - *FIFO ordering:* If a correct process issues *multicast(g,m)* and then *multicast(g,m')*, then every correct process that delivers *m'* will deliver $m$ before *m'*;
  - *Causal ordering:* If *multicast(g,m)* $\rightarrow$ *multicast(g,m')*, where $\rightarrow$ is the happened-before relation induced only by messages sent between the members of $g$, then any correct process that delivers *m'* will deliver $m$ before *m'*;
  - *Total ordering:* If a correct process delivers message $m$ before it delivers *m'*, then any other correct process that delivers *m'* will deliver $m$ before *m'*.

**Agreement**

Multicast delivery is essentially a problem of agreement between processes. *Consensus* is one of such problems. There are other variants of consensus, such as the byzantine generals and interactive consistency problems [26].

Consensus enables a group of processes to reach a common decision, based on their seeded variable values and despite possible failures, such as electing a leader or agreeing on a replicated sensor value. In order to reach consensus, every process begins in the undecided state and proposes a single value, drawn from a seeded set. Every process then communicates with one another, exchanging values, until they all settle for a single one (setting the decision variable), therefore entering the decided state. Finally, the processes propose to proceed or to abort the consensus. The requirements of a consensus algorithm are that the following properties should hold for every execution of it:

- *Termination:* Eventually every correct process sets its decision variable;
- *Agreement:* The decision value of all correct processes is the same;
- *Integrity:* If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value.

The consensus algorithm can not be reliably used by itself on asynchronous distributed systems where a single process failure can occur [15]. The same goes for reliable and totally ordered multicast algorithms. In order to achieve consensus on asynchronous distributed systems, one must use failure detectors [5].

#### Elections

Another specific type of agreement can be found on *election* algorithms [7, 17]. Election algorithms enable a group of processes to select a leader among themselves. These algorithms are very useful in Client-Server scenarios where a single process from a group must act as a server.

In order to reach an agreement many solutions can be used, such as, the process with the highest unique identifier, or the process with the lowest computational load gets elected. A typical election algorithm is initiated by a process that finds that the current leader as crashed. When the election starts, every other process from the group can either be a participant, or a non-participant. Every process possesses an $elected_i$ variable which is set to "$\perp$" when they become participant in the current election. The requirements for any run of the algorithm are as follows:

- *Safety:* A participant process $p_i$ has $elected_i = \perp$ or $elected_i = P$ (where P is chosen as the non-crashed process at the end of the run with the largest identifier, for instance);
- *Liveness:* All processes $p_i$ participate and eventually set $elected_i \neq \perp$, or crash.

### 2.2.2 Group Membership

A vital part of a group communication system is its *group membership service*. This service manages the dynamic membership of groups and multicast communication. A group membership service has four main roles in a group communication system:

- *Providing an interface for group membership changes:* A group membership service possesses operations that enable the creation and elimination of process groups, and the addition and removal of processes to or from a group. In most systems a process can belong to various groups at the same time;
- *Implementing a failure detector:* This service also incorporates a failure detector (see section 2.2.3). Failure detectors monitor each group process in order to know if they become unreachable. If the failure detector suspects a process from crashing, the group membership service excludes it from the group;
- *Notifying members of group membership changes:* The service notifies each group's members when a process joins or leaves the membership;

- *Performing group address expansion:* When a group receives a multicast message, the message does not include the list of the members of the group, so the group membership service handles the delivery of the message to every process group that currently belongs to the membership.

A group membership service can exclude a *suspected* process from a group, even though it may not have crashed. On these situations the group effectiveness is reduced, but if the process has actually crashed, the group no longer needs to wait for messages from this process. In the case of a closed group, the suspected process must rejoin the group (usually with a new identifier) in order to resume exchanging messages with the other members.

Some group membership services can handle network partitions, allowing each sub-group to continue to operate separately. When the network is no longer partitioned, the groups are merged again. On the other hand, some group membership services only allow one sub-group to survive a partition. Usually the one that has more members. All the other processes are ordered to suspend operations.

### 2.2.3 Failure Detectors

A group communication system can not be fault-tolerant if it is unable to detect when a group member has crashed. For this reason, a group membership service must implement a *failure detector* [5].

There can be two types of failure detectors: *reliable* or *unreliable*. A failure detector can only be reliable if the underlying system is synchronous (and few pratical systems are). This type of failure detector is always accurate in detecting a process's failure. After checking on a process's state it may produce one of two values:

- *Unsuspected:* Which only means that some sort of evidence has been recently received that suggests that the process has not failed. The process may, however, have crashed since then;
- *Failed:* A result of *failed* means that the process has crashed. Since it is a synchronous system, there is an absolute bound on message transmission times and for this reason a failure detector knows for certain that a process has failed.

A failure detector is unreliable if the underlying system is asynchronous. This type of failure detectors are never certain of a process's failure, they can only have hints. As such, an unreliable failure detector can produce one of two values:

- *Unsuspected:* As with reliable failure detectors, can only be a hint;
- *Suspected:* A result of *suspected* means that the failure detector has some kind of indication that the process may have failed. Since it is working over an asynchronous system, there is no actual bound to message transmission times, but an estimate is usually used as hint. So, an unreliable failure detector may suspect that a process has failed because it did not send a message in time, but it may just happen to be on the other side of a network partition.

There are eight different classes of failure detectors. The $\diamond$S class is the weakest one able to solve consensus, even though it requires a majority of correct processes to do so. To be useful, a failure

detector class has to satisfy some properties, and those have to be as weak as possible while still allowing the problem of interest to be solved. The class ◇S includes all the failure detectors that satisfy the following properties:

- *Strong Completeness*: Eventually, every correct process that crashes is permanently suspected by every correct process;
- *Eventual Weak Accuracy*: There is a time after which some correct process is never suspected by the correct processes.

Every implementation of this class of failure detectors assume that the underlying system is eventually stable. If this stability assumption is satisfied during a long enough period, the sets of suspected processes satisfy the properties defining ◇S.

### 2.2.4 View-Synchronous Group Communication

View-synchronous group communication was first introduced in the ISIS system [4], then known as "virtually synchronous" communication paradigm. It extends the reliable multicast semantics described in section 2.2.1 to include the changing of group views.

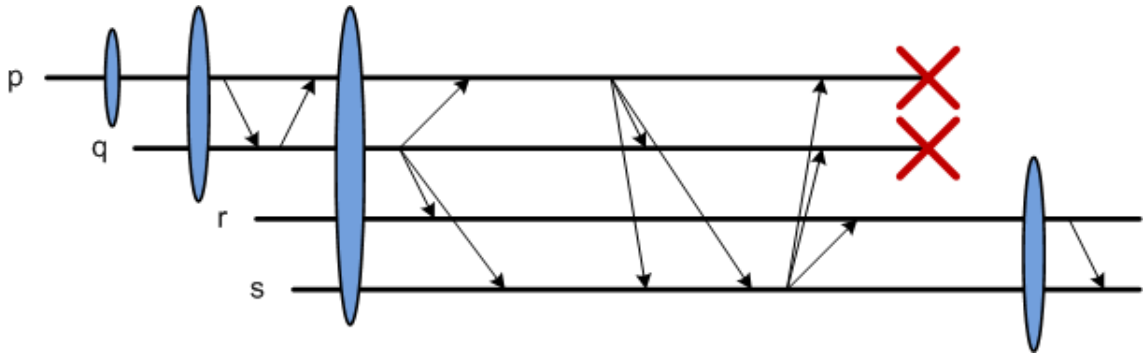


Figure 2.3: View-synchronous group communication.

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). All members are guaranteed to see the same view contents. When a new member joins a group, it will often need to learn the current state of the group (i.e. the current values of data replicated within it). This is supported through a *state transfer*: when installing a new view that adds one or more members to a group, the system performs an upcall in some existing member to request a state checkpoint for the group. This checkpoint is then sent to the joining member or members, which initialize their group replica from it. As with ordered reliable multicast, every process sees the same events in the same order, and hence can maintain a consistent perspective on the data managed by the group. Figure 2.3 illustrates an ordinary view-synchronous group communication run of four processes. First, process $p$ creates a process group, which is subsequently joined by process $q$, and then by $r$ and $s$. Eventually $p$ and $q$ are suspected of having crashed, and the group adjusts itself to drop them.

The guarantees provided by view-synchronous group communication are as follows:

- *Agreement:* Correct processes deliver the same sequence of views (starting from the view in which they join the group), and the same set of messages in any given view;

- *Integrity:* If a correct process delivers message $m$, then it will never deliver $m$ again;
- *Validity:* Correct processes always deliver the messages that they send. If the system fails to deliver a message to any process $q$, then it notifies the surviving processes by delivering a new view with $q$ excluded, immediately after the view in which any of them delivered the message.

## 2.3 Rollback-Recovery

Rollback-recovery, or *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, which can survive all tolerated failures. After a failure, a process can then restart computation from an intermediate state by accessing the stored information. This way, the amount of lost computation is greatly reduced. The recovery information can include the states of the participating processes (*checkpoints*), logs of interaction with input and output devices, events that occur to each process and even messages exchanged among processes.

Rollback-recovery is difficult to achieve in message passing systems because messages create interprocess dependencies during failure-free operation. When a failure occurs, these dependencies may force a process that did not fail, to revert back to an older state, creating what is called *rollback propagation*. For instance, if the sender of message $m$ fails and then rolls back to a state previous to the sending of $m$, the receiver of $m$ must rollback aswell. If it does not rollback, the system will reach an *inconsistent* state where a process received a message that was not sent by any other. In extreme conditions, processes may rollback to their initial state, loosing all the work performed before the failure. This situation is known as the *domino effect*.

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 [6]. Reaching a consistent state is the main goal of every rollback-recovery protocol when failures happen. A message passing system often interacts with the *outside world*. Most of this interactions are important so that the outside world perceives a consistent behaviour of the system. For this reason, rollback-recovery protocols must provide special treatment for these interactions, such as saving them on stable storage. Stable storage is an abstraction that is commonly used to describe a storage device which contains checkpoints, event logs, and other recovery-related information, and that tolerates failures. It can be implemented through several ways, such as, the volatile memory of another process, a local disk in each host, or even a persistent medium (e.g. a replicated file system) outside the host. Information can not be indefinitely saved in the stable storage without it getting full. So, *garbage collection* algorithms are required to delete recovery information which becomes useless.

There are two main rollback-recovery approaches: *checkpoint-based* and *log-based*. The former relies only on checkpoints to achieve fault-tolerance. The latter combines checkpointing with logging of nondeterministic events.

### 2.3.1 Checkpoint-Based Rollback-Recovery

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. For this reason,

they do not guarantee that the execution being performed before a failure is exactly regenerated after a rollback.

#### Uncoordinated Checkpointing

In the uncoordinated checkpointing protocol, each process is responsible for choosing the best time to take checkpoints. Since none of the checkpoints are coordinated, during a recovery it is necessary to find a consistent global checkpoint. To accomplish this, two approaches exist: *rollback-dependency graph* [3] and *checkpoint graph* [42]. Both these approaches attempt to compute the *recovery line*[1] of the system on a similar fashion. These approaches also form the basis for performing garbage collection, since every checkpoint prior to the recovery line is unnecessary. This protocol suffers from the possibility of the *domino effect*. That is, if no recovery line can be found, every process will rollback to their initial state.

#### Coordinated Checkpointing

In coordinated checkpointing, the processes organize themselves in order to take checkpoints. One process sends a request message to the others, and they all take checkpoints before continuing with the normal operation. This coordination can, however, exist without blocking the normal operation of each process, with a protocol known as *distributed snapshot* [6]. When a coordinated checkpoint is taken, a rollback point is sure to exist where the system is at a consistent state. With this protocol, recovery is simplified and the systems are not susceptible to the domino effect. Furthermore, since every process agrees on a checkpoint, there is no need for garbage collection and only one checkpoint is required to be maintained.

#### Communication-Induced Checkpointing

Communication-induced checkpointing is an hybrid between the previous two checkpointing protocols. Processes can take two kinds of checkpoints: *local* and *forced*. Local checkpoints are taken independently, while forced checkpoints are taken in a semi-coordinated fashion, to insure that the recovery line is making progress, and there is no chance of the domino effect happening. Forced checkpoints do not require any special coordination messages to be exchanged, because specific protocol information is piggybacked on each message. This special information is used to determine if past communication and checkpoint patterns can lead to the creation of useless checkpoints. A forced checkpoint is then taken, in order to break these patterns. This intuition has been formalized in a theory based on the notions of Z-path [29], which is a generalization of Lamport's happened-before relation [25], and Z-cycle. A Z-path is a special sequence of messages that connects two checkpoints and allows the application to know that no consistent snapshot can be formed with them. A Z-cycle is a Z-path that begins and ends with the same checkpoint. So, if the processes detect that a Z-cycle is about to be formed, they take a checkpoint to create a consistent rollback point.

[1] A recovery line is a dependency connection between checkpoints from each process which reflect a consistent system state.

### 2.3.2 Log-Based Rollback-Recovery

Log-based 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[2] to stable storage, in addition to the creation of checkpoints as in checkpoint-based rollback-recovery. Since nondeterministic events are being logged, it is possible that upon a failure, a process $p$ that has not failed depends on the execution of a nondeterministic event which cannot be recovered from stable storage or from volatile memory. In this situation, $p$ becomes what is known as an *orphan*. Log-based rollback-recovery relies on the *piecewise deterministic* ($PWD$) assumption [38], which postulates that all nondeterministic events that a process executes can be identified and that the information necessary to replay each event during recovery can be logged in the event's *determinant* [1, 2].

#### Pessimistic Logging

Pessimistic logging protocols log to stable storage the determinant of each nondeterministic event before the event is allowed to actually affect the computation. They do this everytime, because they assume that a failure can occur after any nondeterministic event, hence they are "pessimistic". Upon a failure, processes can resend the determinants logged after the latest checkpoint, allowing the system to recreate the pre-failure execution. Since every determinant is logged, it is impossible for a process to become an orphan.

#### Optimistic Logging

In optimistic logging, there is the assumption that logging will complete before a failure occurs. Thus, and in contrast to pessimistic logging, the application is not required to block before sending a message until the determinant is logged. Because determinants are asynchronously logged, there is the possibility of loosing some when a process fails. For this reason, optimistic logging protocols are prone to the creation of orphan processes. When an orphan process exists, it must rollback to a previous state where it does not depend on any lost determinant. This state is achievable because causal dependencies are tracked during failure-free executions.

#### Causal Logging

In causal logging, every process piggybacks the non-stable determinants on the messages sent to other processes. Upon the receipt of such message, the receiving process adds the piggybacked determinant to its volatile log. When a failure occurs, the determinants not logged to stable storage can then be recovered by the other processes in the system, preventing the creation of orphan processes. However, to be able to recover the determinants lost upon a failure, the recovery protocol of causal logging needs to be more complex.

[2]The receipt of a message and internal events constitute nondeterministic events. Sending a message, however, is not a nondeterministic event.

## 2.4 Discussion

This section presents a discussion about the main topics covered in our related work. We clarify their importance in the choices we make for our proposed solution.

### 2.4.1 Replication

There are many ways to introduce data replication in a system. However, consistency requirements, availability, fault-tolerance and performance are factors that can greatly influence the way a system implements data replication.

The passive replication model , provides systems with strict consistency guarantees, high availability and fault-tolerance (section 2.1.1). However, these advantages may come at the expense of performance, since at any given time there is only one replica manager attending clients demands. Active replication on the other hand, allows the system to answer requests of the clients using many replica managers (section 2.1.2). This can greatly improve the system's performance because there is not a single *bottleneck point*. However, in order to reach an agreement in the presence of byzantine failures, performance might be degraded until the replica managers reach a consensus.

Pessimistic replication is widespread in commercial systems, providing them with strict consistency guarantees, great data availability, fault-tolerance and high performance. However, this solutions can only be deployed in local-area networks and synchronous systems, since they require high network bandwidth with low latency in order to be able to maintain their high consistency guarantees. Optimistic replication also provides the systems with fault-tolerance, data availability and high performance at the cost of looser consistency guarantees. Many applications, however, have loose consistency requirements, or can cope with them, so the ability to deploy this systems in large-area networks, like the Internet, is a "selling point" for optimistic replication.

For this reasons, and considering the environment where our solution will be used (spontaneous ad-hoc networks), we believe that our system should implement an optimistic replication model. Furthermore, since the original VFC design was developed with bluetooth technology in mind, a single server must be used, as such, we also believe a passive replication model proves to be more adequate.

### 2.4.2 Fault-Tolerance

There are several ways to ensure delivery guarantees of messages to a group. Multicast communication allows a group of processes to receive copies of messages sent to the group with varying delivery guarantees (section 2.2.1). These guarantees can assure to the reliability of the service as well as to the order in which the messages are delivered. Multicast communication alone, however, is unreliable on asynchronous distributed systems where a single process failure can occur.

To enable the reliability of multicast communication over asynchronous systems (prone to failures), group membership services are usually implemented (section 2.2.2). This services provides group communication systems with an interface to add and remove processes from a group, a failure detector to monitor the availability of group members, a way to notify members of group membership changes and a guarantee to deliver a message to every process in the group.

A failure detector can be reliable if the underlying system is synchronous, or unreliable if it is asynchronous (section 2.2.3). When it is reliable, it can know for certain when a member of the group has failed. On the contrary, an unreliable failure detector can only assume that a group member has failed, since there is no bound to message transmission times.

View-synchronous group communication is implemented by group membership services to ensure that all correct members of a group see the same view contents (section 2.2.4). Each time there is a change on the group membership, a new view is delivered to every member.

Since VFC already works in a round-basis (see section 2.5.4), it functions almost like a view-synchronous group communication system. In order to tolerate fault-tolerance however, it is necessary to implement the remaining group membership services: a failure detector and an interface for group membership changes.

### 2.4.3 Rollback-Recovery

There are two kinds of rollback-recovery approaches that are not mutually exclusive: checkpoint-based and log-based.

Checkpoint-based rollback-recovery protocols do not keep track of every interaction with the outside world, so they cannot guarantee that the state observed before a failure is exactly restored after a rollback (section 2.3.1). Uncoordinated checkpointing makes each process responsible for the timing of their checkpoints. Since they are uncoordinated, they can lead to the *domino effect*. Coordinated checkpointing requires the synchronization between group members to create checkpoints consistent with each other. This protocol provides a consistent state for every process to rollback to, however, it requires a great deal of coordination to create each checkpoint. Communication-induced checkpointing is an hybrid between the previous two protocols. It does not suffer from the possibility of the *domino effect*, and the coordinated checkpoints do not require any special coordination messages to be exchanged. However, it does require specific protocol information to be piggybacked on each message, which must then be analyzed to force coordinated checkpoints.

Log-based rollback recovery protocols are more suited for applications that frequently interact with the outside world (see section 2.3.2). Pessimistic logging logs to stable storage the determinant of each nondeterministic event before the event actually affects the computation. This protocol ensures the re-creation of a pre-failure execution at the expense of processing power. Optimistic logging is the opposite of the previous protocol, that is, it assumes that the logging of information will happen before a failure occurs. This can can leave the process in a state that is not consistent, however, it does not need great processing power to execute. Finally, causal logging piggybacks non-stable determinants in messages so that it can recover them from other processes after a failure. However, the recovery process of this protocol is much more complex than the previous two.

Rollback-recovery in mobile devices and ad-hoc networks bring some significant issues that do not exist in other systems and networks. Mobile devices have energy constraints, intermittent communications and low performance processors. For these reasons, checkpointing protocols which allow maximum autonomy to participating processes, require low overhead in resources and can function with the minimum possible number of messages exchanges, are favored. Therefore, uncoordinated and communication-induced checkpointing are more viable options than coordinated checkpointing for this environment. Furthermore, pessimistic logging and causal logging also tend to be more appropriate because they provide a higher

degree of autonomy during recovery.

In VFC, however, every host is already coordinated by means of rounds defined by the server (see section 2.5.4). So, the implementation of coordinated checkpointing can be easier than it usually is in a mobile environment.

## 2.5 Commercial & Academic Systems

This section presents four commercial and academic distributed systems that provide data replication for high availability.

### 2.5.1 Gossip

The gossip architecture [24] was developed as a framework with the objective of providing highly available services by replicating data close to where clients need it. A gossip service provides two types of operation: *queries* and *updates*. Clients perform these operations on the gossip system replica managers through a front-end. Front-ends are able to choose the replica manager that is more suited at the moment to execute each client request. Replica managers exchange gossip messages in the background that propagate updates to replicated data.

Each front-end and replica manager, maintain a vector timestamp [14] that keeps track of the latest data updates it has seen on replica managers (each vector entry contains the updates observed on a replica manager). This way, front-ends make sure that clients never observe data that is older than they have already seen.

When a client performs a query operation, the front-end finds a replica manager that contains replicated data sufficiently up-to-date and returns the response to the client, merging the timestamp value of both vectors. If no replica manager can be found that is sufficiently up-to-date, one replica manager must wait for the missing updates, which should eventually arrive in gossip messages, or it can actively request the missing updates from the replica managers that have them.

When a client performs an update operation, the front-end submits that update to one ore more replica managers. The updates can be applied with three different ordering guarantees:

- *Causal:* When a client requests two updates, the second update is applied last in every replica manager. They both can be applied, however, at different times in different replicas;
- *Forced:* An update that is totally as well as causally ordered;
- *Immediate:* Immediate updates are applied in a consistent order relative to any other update at all replica managers.

Replica managers use their vector timestamps to find which replicas are missing updates they have, and propagate them in the background. The frequency at which the gossip messages are propagated in the background is application specific.

The main goal of the gossip architecture is to provide a replicated service, which can still work despite network partitions. The messages are propagated in the background in a optimistic replication fashion, in a similar way to how VFC ensures consistency. It does not, however, use locality-awareness techniques to select which objects should be propagated, like VFC does.

The gossip architecture achieves fault-tolerance by assuring that at any moment at least one site has a replicated object. If a replica manager crashes, the object that a client has just updated is ultimately still present at the client's machine.

### 2.5.2 Bayou

The Bayou system [41, 32] enables data replication for high availability through weaker guarantees than sequential consistency. Bayou replica managers allow clients to keep working despite variable connectivity, by exchanging updates in pairs. Furthermore, the bayou system can handle conflicts that may occur when two replica managers merge their updates, in a semantic way (see section 2.1.4).

Clients are able to query and update the replicated objects in the Bayou system. Moreover, in order to resolve conflicts, Bayou may undo and reapply updates. Until every conflict is resolved by Bayou, each update is considered *tentative*. Under usual implementations of the Bayou system, a replica manager is designated primary, and this primary replica manager has the responsibility of resolving the conflicts and *committing* the updates to the database. In order to detect conflicts, each Bayou update contains a *dependency check* and a *merge procedure*. Replica managers call the dependency check procedure before committing an update, which checks if that update may conflict with any other update already committed. If the dependency check indicates a conflict, then Bayou invokes the merge procedure operation. The merge procedure has some application specific guidelines which allow the system to resolve the conflict, and finally commit the update.

Each implementation of the Bayou system, requires application specific conflict detection, that is, semantic conflict detection. For this reason, it is very complex for the application programmer to create dependency checks and merge procedures, for a possibly high number of conflicts that need to be detected and resolved. Bayou also brings extra complexity to the user, since the system may alter updates that were performed by the user. These issues do not occur with VFC, since what is propagated between replicas is the object itself, and not operations that update the object. That is because VFC is a state-transfer system (see section 2.1.4).

### 2.5.3 Totem

The Totem system [28] introduces an extended virtual synchrony model which extends the model of virtual synchrony to systems in which processes can fail and recover and in which the network can partition and reemerge. This model ensures that the agreed and safe-delivery guarantees are honored within every configuration, even if faulty processors are repaired, or if a partitioned network reemerges. This model can be guaranteed only if messages are born-ordered, meaning that the relative order of any two messages is determined directly from the messages, as broadcast by their sources. This order of messages is guaranteed by the Totem single-ring protocol, which consists on a logical token-passing ring overlay network (see Figure 2.4). The token circulates around the ring as a point-to-point message, with a token retransmission mechanism to guard against token loss. Only the processor holding the token

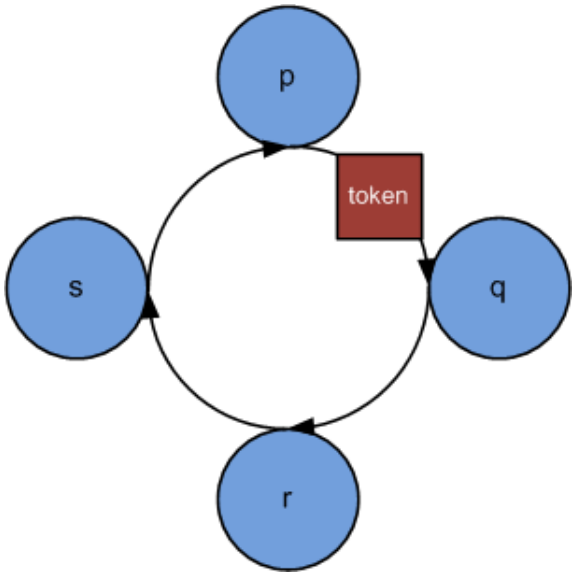


Figure 2.4: The token of the Totem single-ring protocol being passed from one processor to the next.

can broadcast a message. This token provides total ordering of messages, rapid detection of faults, and effective flow control. Since it provides a reliable totally ordered multicast, the consistency of replicated data can be maintained.

In order to operate in large-scale systems, the Totem system provides a multiple-ring protocol, which provides essentially the same service as the single-ring protocol, and with the same properties. The difference, as the name implies, is that the former protocol enables the existence of multiple rings, connected through a ”gateway” process. This multiple-ring protocol uses Lamport timestamps [25] to deliver messages in the same order across multiple rings.

Other implementations of view-synchronous group communication have been developed since the ISIS system [4], such as, Horus [34], Transis [11], Moshe [22], and more recently TransMAN [37] which is a group communication system for mobile adhoc networks.

As with most view-synchronous group communication systems, Totem implements a membership service and failure detectors to provide fault-tolerance. These systems do not provide selective optimistic replication, the way VFC does. Especially Totem, organizing hosts on a ring makes it extremely difficult and slow to propagate updates to specific hosts.

### 2.5.4 Mobihoc

Mobihoc [36] is a middleware platform aimed at supporting the design of multiplayer distributed games for ad-hoc networks and enforces the Vector-Field Consistency model (see chapter 1).

Mobihoc 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 (see Figure 2.5). The Mobihoc core consists of the Consistency Management Block (CMB) and the Session Manager (SM) components. CMB enforces the VFC model, which coordinates the propagation of updates to clients according to the VFC settings they have specified. SM is responsible for sending periodic round messages

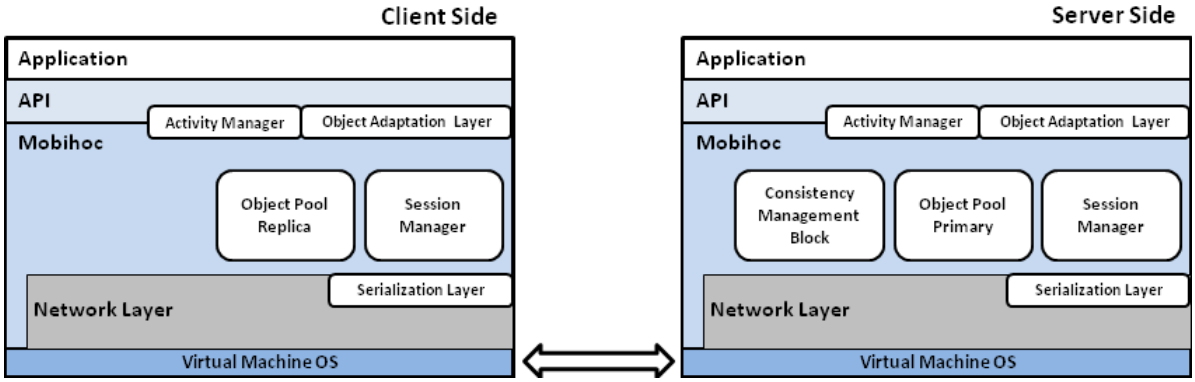


Figure 2.5: Mobihoc client-server architecture.

to clients, and handling lock and release requests.

Each node contains an *Object Pool* where local object replicas are kept. Read operations are made without restrictions, but to perform write operations, a node must acquire a lock from the server. After the release of the lock, object updates are propagated to the server, and then propagated to each node according to their VFC settings.

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. In each round, the server propagates the object updates to the clients and enables the execution of *activities* on them.

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

## 2.6 Summary

In this section we discussed the most important research topics related to our work. We started by analyzing different notions of replication, fault-tolerance and checkpointing and then discussed the most relevant techniques used in these areas. Finally we reviewed some commercial and academic systems that focus on the same topics as ours.

# Chapter 3

# Architecture

In this chapter we describe in detail the architecture of our system. We start with an overview of the overall system architecture and its main characteristics. Then we describe the Vector-Field Consistency model in detail, highlighting the changes we made to the original VFC design. We then proceed to explain the details of our architecture, describing its most important components. Finally, we present our fault-tolerant version of the Vector-Field Consistency model.

## 3.1 System Overview

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.

To accomplish this, we propose the implementation of a *Membership Service*, which executes on the server, and manages the membership changes throughout the system (see section 2.2.2). This management is facilitated by the round based operation of the UbiVFC's server, that already behaves in a similar fashion to a view-synchronous communication group (see section 2.2.4).

More than allowing clients to join and leave the VFC system freely, the group membership service is also responsible of notifying the remaining clients present in the system of this events. When the game is being setup, these notifications can occur at anytime, but when it is active they are only diffused on a round to round basis.

Since a client can fail, a *Failure Detector* is implemented on the server that monitors the liveness of each client. If a client spends a certain amount of time without communicating with the server, the server's failure detector sends a ping request to that client. If that client fails to acknowledge this request within a certain amount of time, the failure detector notifies the group membership service which removes the presumably failed client from the system and notifies the remaining clients of this departure.

A rollback-recovery system is implemented on each client that allows them to recover a saved state when re-joining a server. This system stores a copy of the client's owned objects on their storage devices. This checkpoint is performed each round, and only uses the objects that were updated that round. When a client that has previously failed, or that willingly saved its game state, attempts to re-join the game, the user can either recover its stored game data, or start a new session.

However, the failure of clients is not the only problem that can affect the VFC system. A server failure is a possibility that must be prevented in a way that the game can keep being executed without severely affecting the user experience. In order to achieve this goal, we propose to send to specific clients all the newly updated game objects with no regards to their personal VFC settings each round. With this data, and other structures important to the execution of the server, this *backup clients* are able to assume the server duties when the actual server leaves the system. Backup clients are chosen by the Membership Service from the top of a list that keeps every client ordered. The top clients are the ones that this service estimates will endure more time on the system.

When a server leaves orderly, it notifies the top backup client to assume the server duties, and the remaining clients to connect to the new server. If a server crashes, however, the clients must acknowledge this event by themselves. To accomplish this, a failure detector is implemented on each client that pings the server whenever it fails to communicate for a certain amount of time. The top backup client knows that it must assume the server duties, and the other clients know that they must connect to the new server, because every client keeps a backup clients list, that is updated by the group membership service. This list can only suffer changes when a client joins or leaves the system, or when there is an update to a client's remaining battery expectancy.

## 3.2 The Vector-Field Consistency Model

Vector-Field Consistency is an optimistic consistency model designed to manage replicated data across mobile devices executing a multi-player ditributed game. The main objective of this consistency model is to reduce bandwidth requirements of the system it is being executed on, and therefore, reducing the impact on the device's battery. VFC accomplishes this goal with the definition of *consistency zones* around *pivot* objects (e.g, a player's avatar on a first-person shooter game). Each consistency zone possesses different consistency requirements that decrease as the distance to the pivot increases. In the following sections we describe the Vector-Field Consistency model in detail.

### 3.2.1 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.

In figure 3.1 we can see pivot (P) generating four consistency zones (Z1, Z2, Z3 and Z4) which contain several game objects (O1, O2, O3 and O4) that are subject to the consistency degrees of the zones they

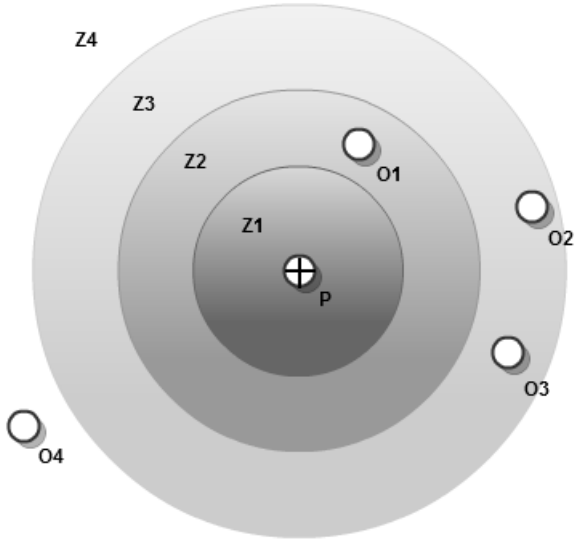


Figure 3.1: Conceptual consistency zones.

are within. In VFC, consistency zones $Z_i$ are defined as follows: if i = 1, then $Z_1$ is the inner circle surrounding the pivot (in figure 3.1 it corresponds to zone $Z_1$); if 1 < i < n, then $Z_i$ is the area enclosed between $Z_i$ and $Z_{i-1}$ (zones $Z_2$ and $Z_3$ in firgure 3.1); if i = n, then $Z_i$ is the zone beyond the last circunference surrounding the pivot (zone $Z_4$ in figure 3.1).

Consistency degrees are 3-dimensional *consistency vectors* $\kappa = [\phi, \theta, \nu]$. $\kappa$ 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 ($\theta$)*: Specifies the maximum time (in seconds in the original implementation of VFC, in rounds in our system) an object can stay without being refreshed with its primary replica's latest value. With this metric VFC guarantees that an object within a consistency zone specified by vector $\kappa$ is, at most, $\kappa_\theta$ rounds in an inconsistent state;
- *Sequence ($\phi$)*: Specifies the number of updates an object can get without them being applied to its replicas. With this metric VFC guarantees that an object within a consistency zone specified by vector $\kappa$ is, at most, $\kappa_\phi$ updates behind the primary replica;
- *Value ($\nu$)*: 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 (e.g., they can represent the player's score), and it defines the maximum percentage difference between an object and its replica. VFC guarantees that an object within a consistency zone specified by vector $\kappa$ is, at most, $\kappa_\nu$ percent divergent from its primary replica.

#### 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.

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

## 3.3 System Architecture

Like VFC, UbiVFC also follows a client-server architecture. Clients are organized on a star-like topology with the server at the center, as shown in figure 3.2. At any given moment, every client is only connected to the server, and the server is connected to each client. A mobile device hosting the server can also be executing a client.

This topology was first chosen for the original VFC design, which was implemented over bluetooth connections. The reason for this choice was the limitations of the bluetooth technology, that imposes a single node of the network to relay all messages between any two nodes. For our own design we decided to keep this topology for two reasons:

- *Connections constraints*: a client can only have a connection open at any moment, which is used to communicate with the server. With this design decision, clients are not required to be expecting connections, or to start connections of their own, saving some important mobile resources, as battery life and bandwidth;
- *Simplicity*: the original VFC design was developed with this architecture in mind, so, in order to keep it simple, we decided to use an architecture with proven results.

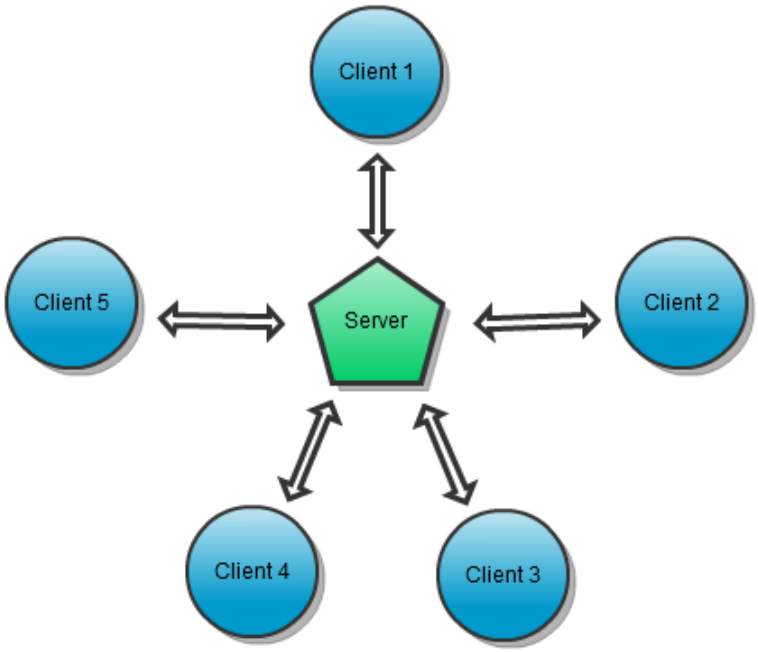


Figure 3.2: UbiVFC's star topology.

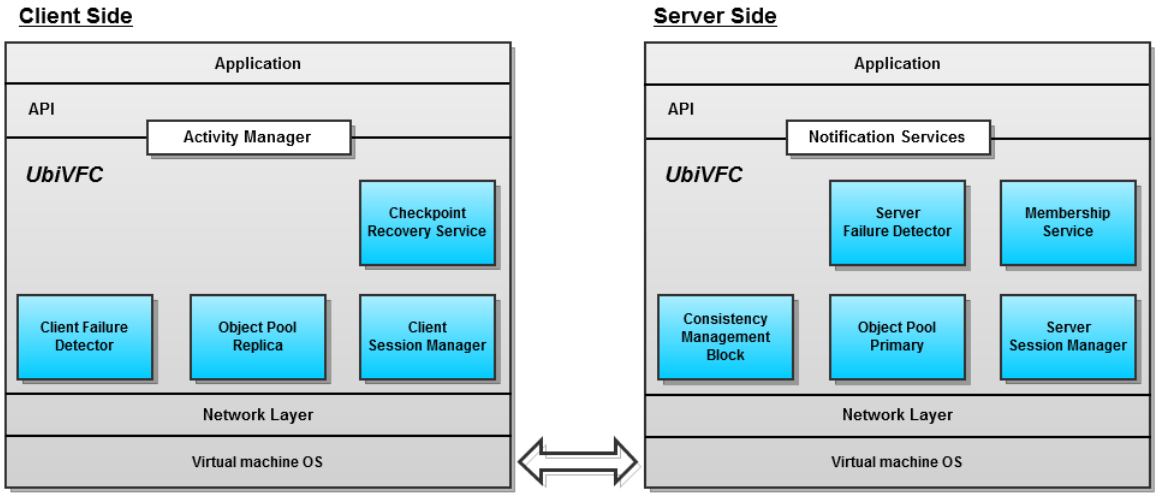


Figure 3.3: UbiVFC architecture.

Figure 3.3 presents the main components of our solution's system 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, if the ones already present do not suit their needs.

On top of the UbiVFC 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.

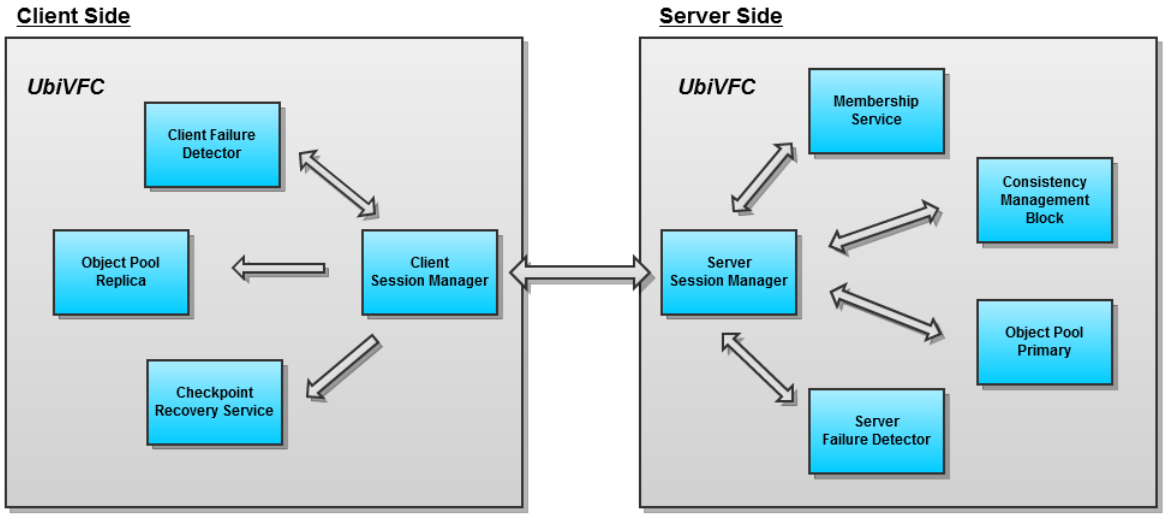


Figure 3.4: UbiVFC components interactions.

The UbiVFC layer contains the components that are responsible for the enforcement of the VFC model and for making our solution fault-tolerant. The client side UbiVFC layer is different from the server's, even though some of their components share similar names. Figure 3.4 presents the way the UbiVFC components interact with each other. On either side, the *Session Manager* is responsible for coordinating every interaction between components. Every component, with the exception of the Session Managers, do not have knowledge of each other. On the server side, the *Membership Service*, the *Consistency Management Block*, the *Object Pool Primary* and the *Server Failure Detector* do not work by themselves. They only process the *Server Session Manager* requests, and hand back the results. On the client side, the *Client Failure Detector* only needs some initial information from the *Client Session Manager* and then executes independently from it. When it detects a failure, however, it hands that information to the Session Manager. The *Object Pool Replica* and the *Checkpoint Recovery System* receive some information periodically. Finally, both Session Managers are also responsible for executing the protocol that enables the communication between client and server.

UbiVFC extends on the original VFC design (section 2.5.4), so some of its main building blocks are also present on our design. There are, however, some differences which we will explain over the next sections.

### 3.3.1 UbiVFC Components

The main UbiVFC components are the Consistency Management Block, the Session Manager, the Object Pool, the Membership Service, the Failure Detector and the Checkpoint Recovery System. Only the first

three components of this list were also present in the original VFC design. As we can see in figure 3.3, the Consistency Management Block and the Membership Service are only available at the server, as opposed to the Checkpoint Recovery System, which only exists at the client. The remaining components are present on both the clients and server, presenting however, different characteristics.

#### Session Manager

The Session Manager is responsible for managing the communication between client and server. Each session manager executes the protocol that provides the UbiVFC services to the game programmers.

The Server Session Manager 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. It provides eight services that are used by the Client Session Manager to process requests. The services provided are described as follows:

- *Subscribe*: this service is responsible for registering a new client into the membership service.
- *Publish*: this service processes the clients owned objects to the game, ultimately adding them to the object pool.
- *Enable*: this service enables the game. A round trigger system is initialized that periodically notifies the consistency management block of new rounds, therefore processing the VFC model.
- *Write*: this service processes updates made by clients to their replicated objects. The updates are merged with the server's object pool and the consistency management block is notified of this event.
- *Disable*: terminates the execution of the current game.
- *Leave*: used by clients to notify the server of its intent to leave the game.
- *System Info*: this service is used by clients to notify the server of a drop on their device's remaining battery life. This information is used by the Membership Service to organize a list that orders the clients over their estimated remaining time on the game.
- *PingPong*: used by the Client Failure Detector to check if the server is still available.

The **Client Session Manager** not only provides the server with the necessary services that are required to process the VFC model, but also provides the game with methods to call the server's services. The services provided by this session manager are described as follows:

- *Round*: this service is called by the server whenever a new round is triggered. The clients get notified of new rounds piggybacked with object updates that the server's Consistency Management Block saw fit to propagate based on the client's VFC settings. They can also receive some backup data if the server's Membership Service chose that client to be a server backup.
- *Enable*: this service is used by the server when it has received an enable request from another client. When a client receives this request, it proceeds to start the game client.
- *Disable*: called by the server to terminate the current game.

- *New Server*: this service is called when the game's current server is leaving and this client is the backup server. On this event, the notified client becomes the new game server.
- *Connect to New Server*: notifies the client that it must disconnect from the current server, since that server is leaving. It then proceeds to connect to another server which was previously notified that it should assume the server duties.

#### Consistency Management Block

The Consistency Management Block (CMB) is executed exclusively on the server and is responsible for the enforcement of the VFC model. Both functions mentioned in section 3.2.2 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, this component aggregates the VFC consistency parameters (see section 3.2.1) specified by each client. This parameters can be provided by the clients to the server at the moment they are connecting, or they can be specified by the server if no parameters are provided. On the latter case, it is up to the game programmers to specify a default VFC consistency vector to be provided by the server.

Each time a client makes an update to a replicated object, the *update-received* function is triggered, and the Server Session Manager dispatches the update to the CMB where it is stored and the number of missing updates to that particular object is incremented. When a new round is triggered by the Session Manager (the *round-triggered* function), 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.

#### Object Pool

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

The Object Pool Primary 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 in the clients is updated each time a new round is triggered and the server's Consistency Management Block finds new updates to propagate to that client, so, the state of the objects in the client's Object Pool is defined by their VFC settings.

#### Membership Service

The Membership Service 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 the 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.

#### Failure Detector

A Failure Detector is implemented on both the client and the server.

The Server Failure Detector is responsible for knowing if a client has failed. It 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 *Ping* request after another pre-defined number of rounds, the Failure Detector then notifies the Server Session Manager of the assumption that that client has failed.

The Client Failure Detector only needs to periodically check the server for availability. Like the Server Failure Detector, 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.

#### Checkpoint Recovery System

This component is responsible for periodically creating checkpoints of the current 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 Checkpoint Recovery System 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.

## 3.4 Fault-tolerant Vector-Field Consistency

As was initially proposed, UbiVFC main objectives was to provide VFC with tolerance to membership changes throughout its game execution, which can occur in an orderly or disorderly fashion, and to have the possibility of recovering the game state of a previous game session. To accomplish this goals a series of protocols needed to be developed that require coordination between clients and server, and even between UbiVFC's components. This protocols are what distinguish the original VFC from ours, and are explained over the next sections.

### 3.4.1 Client Subscribes Protocol

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 (see next section 3.4.2) with the recovered data.

In order to join the game, the client must send a *subscribe* request to the server. The Server Session Manager 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:

1. If the game is not yet started, the server's Session Manager adds the client to the Membership Service, which then proceeds to register it in the system.
2. If the game is already active, the Server Session Manager adds the client to a subscription waiting queue where it stays until a *publish* request is received from that same client.

### 3.4.2 Client Publishes Protocol

When a client sends a *publish* request to the server, the server's Session Manager sends back a response of acknowledgment if the request was completed successfully. If it was successful, the server can do one of two things, depending on the state it is in:

1. If the game has not yet started, the server's Session Manager hands the objects piggybacked on the publish message to the Object Pool for storage, and notifies the Consistency Management Block of the new arrivals. It then propagates the new objects to the remaining clients, and the rest available in the Object Pool to the new client.
2. If the game is already running, the Session Manager adds the client to a *publishing waiting queue*. When a new round is triggered, the Session Manager checks the publishing waiting queue for entries, and proceeds to add the published objects to the data pool, and to notify the Consistency Management Block of the new object arrivals. After propagating the new objects to the other clients and the previously available objects from the Object Pool to the new client, the Server Session Manager sends an enable request to the client so that it can initialize the game.

### 3.4.3 Client Leaves Protocol

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 server's Session Manager simply asks the Membership Service for the removal of that client's personal data and notifies the remaining clients of the departure. If the game is already underway, the Session Manager 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 Membership Service to remove that client's information and objects from their respective structures.

### 3.4.4 Client Fails Protocol

Both the clients and the server implement a Failure Detector of their own (see section 3.3.1). 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 Server Failure Detector checks if there was a client that has not communicated with the server for a certain amount of time. This interval of time, $\lambda$, can be set by the game programmers to suit their game needs. So, if $\lambda$ has elapsed without the server getting any message from a certain client, it sends a *ping* message to that client. If another $\lambda$ time has gone by without receiving a *pong* response, or any other type of message, the Server Failure Detector notifies the server's Session Manager of this event. The Session Manager then broadcasts the departure information to every other client, and notifies the Membership Service and the Consistency Management Block to remove any data that belonged to the failed client.

### 3.4.5 Server Leaves Protocol

When a server wants to leave the game (or usually when a user which is both a client and a server wants to leave the game), the server role must be assumed by another device which is already a client. The client that assumes the server role was previously chosen as backup by the leaving server (see section 3.3.1). Each round, the server's Session Manager sends the backup clients all the updates received that round, and some optional backup data. The backup data is optional since it may already have been sent to that backup client on a previous round. For instance, if a client is a backup at *round 1*, there are only two ways where it might not be one in *round 2*, and the server has to send new backup data to another client, which are: a new client joins and is selected has backup, or a client sends a new battery update and is selected has backup.

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 game, the new server enables the game, continuing from where it left-off.

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. The server can take more time to set up than it is supposed to, so the clients must be able to wait some time for it.
- 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.

### 3.4.6 Server Fails Protocol

The worst case scenario, that our solution aims to solve, happens when the UbiVFC server fails. At this moment, the game must keep being executed, which means that a client must also assume the serve role. In order to detect a server failure, a Failure Detector is implemented on each client (see section 3.3.1). The Client Failure Detector works in a similar fashion to its server counterpart (see section 3.4.4). It waits $\lambda$ time until it sends a *ping* request to the server, if no message has been received during that time. If after another $\lambda$ 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 3.4.5.

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.

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.

## 3.5 Summary

In this section we explained in detail the architecture of our system. We began with a general overview of the system and its most important aspects. We then proceeded to describe the Vector-Field Consistency model which we extend in our solution. Afterwards, we presented in detail the architecture of our system, its main components and interactions. Finally, we described what sets our solution apart from the original VFC design, which is, the ability to withstand client and server failures.

# Chapter 4

# Implementation

This chapter present some details about the implementation of our system. We start by providing information about our development environment. We then describe the main data structures of our system, both the new and the ones adapted from the previous implementation of VFC. Further on, we present details about the algorithms and protocols used by UbiVFC system. Finally we present the application developed to evaluate our UbiVFC prototype.

## 4.1 Development Environment

Our Fault-Tolerant Vector Field consistency solution was developed in the Java programming language[31]. Specifically, it was developed using the Android Software Development Kit[19], over the Eclipse development platform[16]. Our solution was developed using only the native libraries from the Android SDK.

We used the version 2.1 of the Android SDK, so our solution is compatible with every Android device running Android 2.1 or greater. The version of eclipse used was version 3.6, codename *Helios*.

We also used the Android Development Tools plugin for eclipse[18], version 10. This plugin includes DDMS, which enables the debugging of android applications and provides the developer with emulators to test the applications in simulated environments.

In order to support the communication between the emulators, it was necessary to redirect the tcp traffic that was generated by the emulators, since they can only natively communicate with their host. The windows application *Putty*[39] was used to accomplish this goal.

## 4.2 Algorithms and Supporting Data Structures

In this section we overview the main algorithms and supporting data structures of UbiVFC.

### 4.2.1 Object and User Representation

The two fundamental data structures of UbiVFC concern the representation of game objects and users. They are *DataUnits* and *UserAgents*.

DataUnits are objects that can represent any kind of shared game entity (e.g., an avatar or an user's personal score). The only required information for any DataUnit is its unique integer identifier, *id*. In order to create specific objects for the game being developed, game programmers must only create a DataUnit subclass containing new fields. This feature makes UbiVFC independent of the application being developed on top of it. Not only does it support the creation of games, but also any other type of application.

UserAgents constitute the representation of users on UbiVFC. Unlike DataUnits, their unique identifier is not an integer, but a string containing the nickname of the user. For this reason, there cannot be two users playing the same game with the same nickname. The reason for having a string as the UserAgent identifier is that there was a need to know when a user was re-joining a game session where he already have been. Since Android version 2.1 does not support the retrieval of unique hardware identifiers (MAC), and the Android emulators used on our testing environment do not keep the same IP address between connections, that was the only logical solution.

### 4.2.2 Client and Server Session Manager State Machines

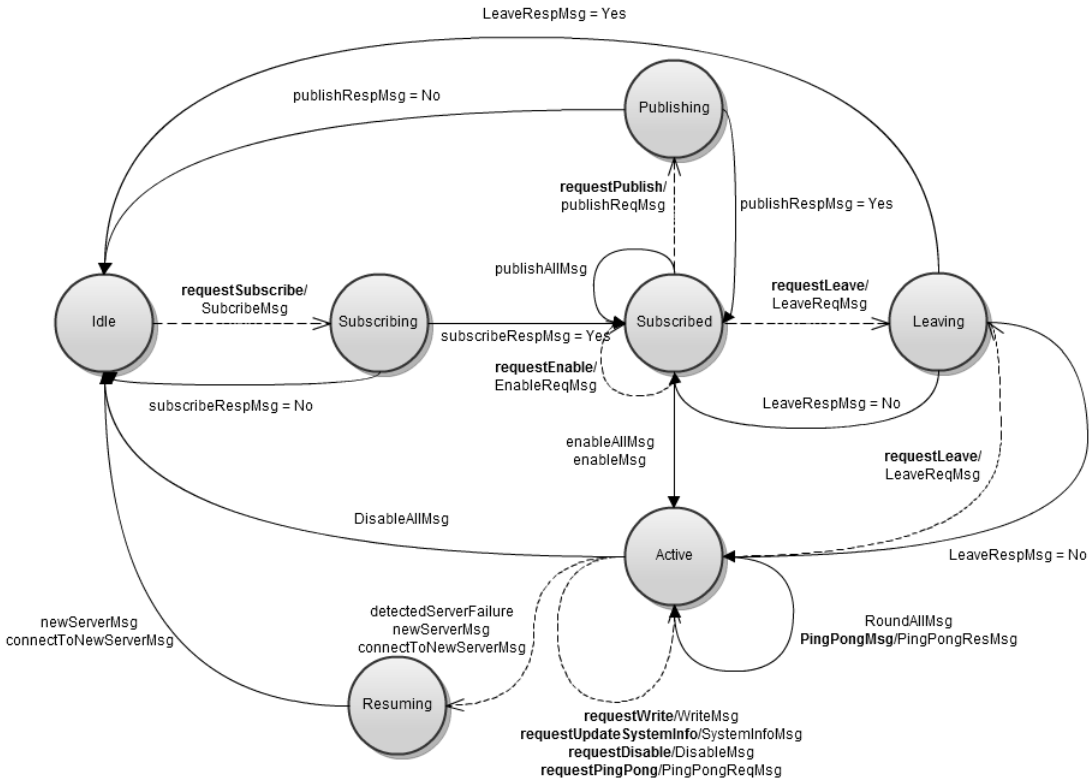


Figure 4.1: Session Client state machine.

The Session Managers of both the client and server execute the protocol that provides the UbiVFC services to the game programmers. Each implements its own state machine, shown in figures 4.1 and

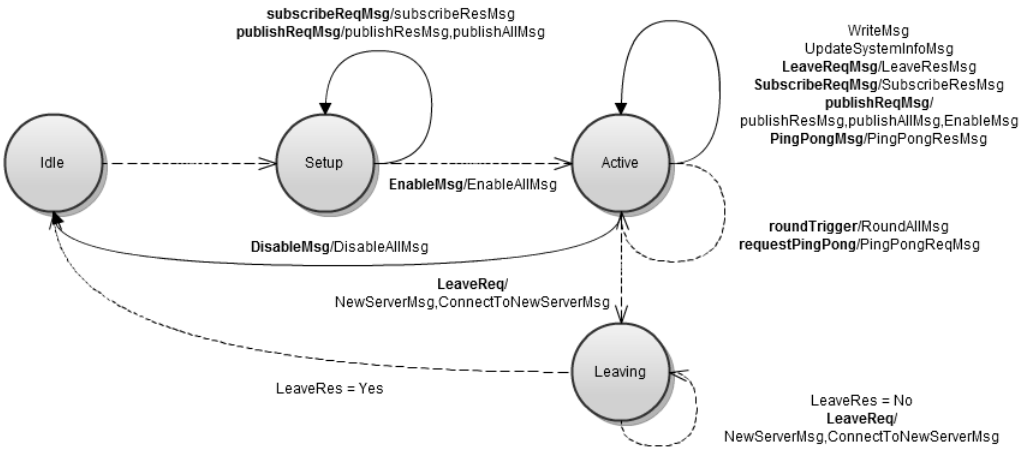


Figure 4.2: Session Server state machine.

4.2. The circles represent the states where a session manager might be in, while the arrows between states represent state transitions. State transitions are triggered by events. Each arrow description is constituted by two parts separated by a slash: the left side is the event name while the right side is the outgoing message sent to the remote peer. Straight arrows represent incoming messages while dashed ones represent API requests or internal events.

The two main server states are the *Setup* and *Active* states. The server is in the Setup state when it is accepting the initial client connections before the actual game is started. The server enters the Active state when it receives an Enable request from a client. At this point the actual game begins. When the server is in this state it can also accept client connections. This behavior is different from the original VFC, since it could only accept client connections in the Setup state. In the server's active state the *Round Trigger* event executes the VFC enforcing algorithm that propagates object updates to clients. The server can leave this state either by receiving a *Disable Message* from a client, where it goes back to the *Idle* state or, by triggering a *Leave Request*, where it enters the *Leaving* state. When the server enters the latter state it sends a *New Server Message* to the chosen backup client and *Connect to New Server Messages* to the remaining clients in the system. If the new server is successfully started, the server goes back to the Idle state, if otherwise, it tries a different backup client until there are no more available. In the Idle state the server does nothing.

The two main client states are the *Subscribed* and *Active* states. A client is subscribed after having successfully sent a *Subscribe Message* or/and a *Publish Message* to the server. The client enters the Active state once it receives an *Enable Message* from the server. At both this states (Subscribed and Active), the client is able to send *Leave Request Messages* to the server, therefore entering the *Leaving* state. At this point, if the server accepts the request, the client goes back to the Idle state, if not, it stays in the state where it was before. When the client is in the Active state, the game is being executed. This means that the client will receive periodic *RoundAll Messages* which bring object updates to the client's Object Pool. If the Client Failure Detector detects a server failure (see section 3.3.1), or the client receives a *New Server Message* or a *Connect to New Server Message* it enters the *Resuming* state. In this state the clients execute the protocol to assume the server duties or to connect to a new server (see section 3.4.5). The execution of both this protocols sends the Session Manager back to the *Idle* state where the game execution is resumed under another server.

The description of the services that process each message can be found in section 3.3.1.

### 4.2.3 Object Pool

The Object Pool is implemented by the class *DataPool*. Even though this class was inherited from the original implementation of VFC it received a major overhaul. The original VFC's DataPool was internally implemented as a fixed size array of DataUnits. The addition of objects to the pool could only be accomplished during the server's *Setup* state, since no clients could join the game at any other given moment. This pool was also immutable because its objects could not change places. For this reason the position of the objects in the pool matched their id, allowing the access to them to be achieved in constant time. Our solution, however, allows clients to join and leave the game at any moment they wish, so, to support this operations we implemented the Object Pool internally as a dynamic array of DataUnits.

### 4.2.4 Consistency Management Block

The Consistency Management Block is responsible for enforcing the VFC model and is implemented by the class *CMB*. The methods *computeUpdatesToDiffuse* and *updatesReceived* implement the algorithms described in section 3.2.2. More than implementing VFC, this component offers a generic interface allowing UbiVFC to support different consistency models. In our solution, three consistency models were implemented by sub-classing CMB, but only one provides UbiVFC with full support to VFC: *FullCMB*.

FullCMB contains three important data structures that were adapted from the original VFC implementation: *timesUpdated*, *lastUpdates* and *updateQueue*. *timesUpdated* is an hash table that maps UserAgents to a list of integers that represent the amount of times each object was updated. *lastUpdates* is another hash table that maps UserAgents to a list of the last DataUnits sent to that client. Finally, *updateQueue* is implemented by the class *UpdateQueue* that manages the last updates that arrived at the CMB.

When a list of updates reaches CMB through *updatesReceived*, CMB adds the DataUnits to the *updateQueue* and increments the number of updates in *timesUpdated* for each DataUnit received, and for every other UserAgent.

When the Server Session Manager triggers a new round, *computeUpdatesToDiffuse* is executed. This function is able to determine which updates are due to which clients using an algorithm that is straightforward: for each client, identify which objects are closer to their pivots; then check in which *consistency zone* of the closest pivot that object is located; finally, verify if the object violates the *consistency degree* associated with the *consistency zone* it is located in; if that is the case, the object must be sent. CMB is able to make this verification because a reference to the *phi* data structure owned by the Server Session Manager is available (see section 4.2.5).

### 4.2.5 Session Manager

As the Session Manager state machines section 4.2.2 showed, the client's Session Manager is very different from the server one. The same holds true for their supporting data structures. Both Session Managers need many support structures in order to process the protocol that provides the UbiVFC's services. Many of this structures were inherited from the original VFC implementation. The server's Session Manager is implemented by the class *SessionServer* and the client's one is implemente by the class *SessionClient*. The remainder of this section presents the most relevant support data structures available in this classes and the reason for their presence.

#### Server Session Manager

*phi* is an hash table that holds the VFC's consistency requirements for every player present in the game. It is indexed by UserAgents that are mapped to their respective phi values. Each phi value is implemented by the class *Phi*. This class stores the player's consistency zones (in the form of an array of integers corresponding to the zones' radius) and degrees (stored on a bi-dimensional array). This class also holds references to the objects owned by the players.

*duAssociation* is another hash table that maps shared objects to UserAgents. The has tables' key is just the DataUnit's unique identifier. This hash table is used to maintain a record of which DataUnit belongs to which user.

Both this structures were also available in the original VFC implementation.

Three hash tables were implemented specifically for our design: *clientsLeaving*, *clientsSubscribing* and *clientsPublishing*. Each of these hash tables has got a unique integer identifier as key. This identifier is provided by the Network Layer when a new client connects to the server, and can be used as a lighter key to identify users. Every one of the above mentioned hash tables constitute waiting lists for requests that clients make when the server is in the Active state, and can only be processed when a new round is triggered. When this happens is triggered we have a controlled environment where we can make changes to fundamental data structures without having conflict issues. For instance, when a client leaves we must remove their owned objects from the Object Pool and their phi settings from the *phi* hash table. As soon as the clients' actions on this waiting lists are executed, the hash tables are cleared. Both the *clientsLeaving* and *clientsSubscribing* hash tables use private classes as values. This private classes (*ClientLeaving* and *UserAgentSysInfoPair*, respectively) hold important data to process those requests. The *clientsPublishing* hash table's values are fixed size arrays of DataUnits that contain the objects the user wishes to register in the game.

Another important new addition is a boolean named *resuming* which represents a "super" state the server can be in. The server reaches this state when it is assuming the server role that has previously belonged to another. When the server is in this "super" state, it must go through the same states shown on section 4.2.2 to reach an active one, but this way, the server knows that it is waiting for specific clients to join, and that it must resume the game after some time even if not all expected clients have arrived. The information about the clients that are expected to re-join the game is kept on a string dynamic array named *clientsRemainingToResume* that contains the nicknames that identify those clients.

#### Client Session Manager

The client's Session Manager does not have to maintain many data structures. Only the clients chosen has backups by the current server have replicas of important data structures that are essential to assume the server duties in case the current one fails or leaves. So, this session manager can hold copies of the most important data structures implemented in the server:

- *registeredUsers*: an hash table containing the registered users in the system;
- *duAssociation*: an hash table containing the association between shared objects and users;
- *phi*: an hash table containing the phi settings of each user in the system;

- *usersURLs*: an hash table that maps user nicknames to their "url" (e.g., an IP address);
- *connectionType*: an integer with the connection type that is currently being used;
- *cliIdCount*: the number of identifiers that was given to clients by the network layer;
- *objectIdCount*: the number of identifiers that was given to objects by the server's Data Pool.

The one data structure that exists in every client's Session Manager is the *backupUsers* fixed size string array. This array contains the "urls" of the backup clients in an ordered fashion. This "urls" can have many forms, depending on the type of connection being used. In our implementation, it can be an IP address if real mobile devices are being used on a wifi network to play the game, or it can simply be a port, if the game is being played on android emulators. This structure is used to inform the clients to which "url" they must connect to when the current server leaves or fails. This structure is piggybacked to round update messages by the server's Session Manager when there is a change to the list of backup clients.

### 4.2.6 Membership Service

The *MembershipService* class manages the users on the system. So, the main data structure it holds is the *registeredUsers* hash table, which maps unique integer identifiers to UserAgents. This integer identifiers are provided by the Network Layer when a new client connects to the server. This structure is the only one present in this class that was also available in the original VFC implementation.

*usersURLs* is an hash table that maps users' nicknames to their "urls". This hash table includes the information available on the *backupUsers* array presented in the previous section as well as the "urls" for the rest of the clients in the game.

*clientsOrdered* is a dynamic array of *BatteryRecords*. BatteryRecords is a private class that holds information on a client's battery state over time. As the name of the structure implies, it is an ordered list. It's order is defined by the estimated time left available for each client. In order to make this estimation, the BatteryRecords class holds the last two battery values that were sent by the client to the server using the *Update System Info* service, as well as the respective rounds which that information arrived in. Using only this information, the Membership Service is able to estimate the last round that client will have battery, its *round of death*, using the following formula:

$$x = \frac{level_{init} * (round_{last} - round_{init})}{level_{init} - level_{last}} + round_{init} \tag{4.1}$$

$x$ is the estimated *round of death* of said client. $level_{last}$ is the last battery level received from the client while $level_{init}$ is the previous update received. Finally, $round_{last}$ is the round when the last update to the battery level was received from the client while $round_{init}$ corresponds to the previous update received.

The last two structures available in the Membership Service are the dynamic arrays of UserAgents *backupUsers* and *previousBackupUsers*. Each round, a certain amount of clients are selected from the

top of the *clientsOrdered* list to be backups[1]. Both this arrays contain the selected backup clients of the present round, and the previous one, respectively. Both this arrays are necessary to know if there was a real change to the backup roster, in the event of a client entry or departure, or an updated client's battery level.

### 4.2.7 Failure Detector

A Failure Detector is implemented in the client by the class *ClientFailureDetector*, and on the server by the class *ServerFailureDetector*. The Server Failure Detector is called by the server's Session Manager whenever a new round is triggered. However, it only checks for client failures every four rounds. The Client Failure Detector executes on a separate *thread* from the client's Session Manager, because if the server fails, no round is triggered. Like the server's Failure Detector, the client's one also checks for server failures every four rounds. The frequency of this failure detections can be changed by the game programmers to suit their needs.

The supporting data structures of both failure detectors are quite similar, the only difference being that the server must check upon every client, where the client must only check upon the one server. The Client Failure Detector's supporting data structures are a *long* value containing the time of the last message received from the server called *lastNewsFromServer*, and a boolean, *pingSent*, that informs the Failure Detector if it has already sent a *ping* message to the server (see section 3.3.1). The Server Failure Detector's supporting data structure is an hash table, called *lastNewsFromClient*, that maps a user's unique integer identifier to the class *ClientNews*. This class contains a long value, *lastNews*, containing the time of the last message received from that client and a boolean named *pingSent* that informs the Failure Detector if it has already sent a ping message to that client.

### 4.2.8 Checkpoint Recovery System

The Checkpoint Recovery System only needs to keep two supporting data structures: *ownedObjects* and *objectsToBackup*. The former is a fixed size integer array that keeps a record of the objects that are owned by that client. The latter is a dynamic array of integers containing the identifiers of the objects owned by the client that were updated that round. This last structure is important in order to only create checkpoints of objects that have changed. If it has not changed, it was already backed up before by the Checkpoint Recovery System.

### 4.2.9 Other Supporting Data Structures

The data that may be piggybacked in round messages to backup clients (see section 3.4.5) by the server's Session Manager is implemented by the class *ServerSyncData*. The structures that constitute this class are the same that are present in the backup client's session managers and are described in section 4.2.5. This backup data is only piggybacked to round messages when a new client has joined the game that

[1] At the moment, the number of backups is limited to one until there are 10 clients playing the game, and then the number of backups is determined by the the integer result of the division between the number of clients and 5. This way, when there are 5 clients in the game, there is only 1 backup, when there are 10 clients in the game there are 2 backups, and when there are 15 clients in the game there are 3 backups, and so forth. The number of backups that a game has at any given time can obviously be changed by the game programmers.

| | 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 4.1: Summary of supporting data structures implemented in UbiVFC.

round, since that is the only event that can bring new data to the structures that need to be backed up[2]. Furthermore, if the backup clients list (see section 4.2.6) has not changed since the previous round, the backup clients need only receive the new objects that entered the system, adding themselves that data to their backup structures.

### 4.2.10 Summary of Implemented Data Structures

Table 4.1 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,

[2]When a client leaves the game, the server notifies every other client of that event and the backup clients remove the required data from the backup structures they have.

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.

## 4.3 Game Programming Interface

Game programmers must follow certain steps in order to use our UbiVFC system. They need only to interact with the API layer of the architecture shown in figure 3.3. The rest of the system may be treated has a "black box".

The server part of the game must implement the interfaces *IHostConfigListener* and *IUbiVFCServerListener*. The former interface implements two methods that are required to create a new server: *callbackNeedInfo* requests information from the application (e.g., a port number), and *callbackConfigResults* provides the server application with a new connection. The latter interface provides the application with the *Notification Services* that are presented in figure 3.3. This interface implements three methods that inform the application when messages arrive and when clients join or leave the system. When the new application is provided with a new connection, it must then instantiate the class *UbiVFCServer* to actually create a new server.

The client part of the game must implement the interface *IServerFinderListener*. This interface contains two methods that are required to connect to a UbiVFC server: *callbackNeedInfo* and *callbackSearchResults*. As in the server, the former method requests information from the application (e.g., an IP address to connect to). The latter provides the client application with a connection to the server that is used to launch a new game.

The game part of the client application must implement the interface *IUbiVFCApp*. This interface provides the application with an *Activity Manager* that is presented in figure 3.3. This interface provides the game with every service that the server may use, or that the client may trigger:

- *stateUpdated*: When a new round is triggered and there are object updates.
- *newData*: When a new client has joined the game and, for this reason, new objects have entered the game.
- *enable*: When an enable request has reached the client, to start the game.
- *leave*: A confirmation to leave the game or information regarding another client that has left.
- *connectToNewServer*: The client must connect to a new server since the current one has left.
- *newServer*: The client must assume the server duties since the current one has left.
- *recoverSession*: The client calls this method when it is starting up and finds checkpoint recovery data on the storage system.

When the game part of the client application is provided with a new connection it instantiates the class *UbiVFCClient* to actually start the game and connect to the server. This class provides the game with the necessary methods to use the services provided by the server's Session Manager and described in section 3.3.1.

## 4.4 Summary

In this chapter we presented the most important implementation details of our system. We started by describing our development environment and followed by explaining in detail the algorithms and supporting data structures used by UbiVFC. Finally, we explained how programmers can use our system to develop game applications.

# Chapter 5

# Evaluation

This chapter presents the experimental results of our evaluation of UbiVFC. In order to evaluate our solution we developed a multi-player distributed game called *Snakes VFC*. This game is based on the code sample provided by Google to demonstrate game development with the Android SDK, called Snake[20]. The original code sample suffered many changes that transformed a single-player game controlled by the user, into a multi-player game that plays by itself.

The tests were performed on a computer equipped with an i5 750 processor running at 3.6GHz, with 4 GBs of RAM and Windows 7 64bit as its operating system. The developed game application was executed on android emulators provided by the ADT plugin for eclipse.

Over the following sections we present and analyse the experimental results obtained with the described simulation environment. We start by presenting the Snakes VFC game in detail. We then proceed to present the quantitative and qualitative evaluation discussing the results obtained.

## 5.1 Snakes VFC

Snakes VFC is a vastly modified version of the original Snake game that became famous since it was pre-loaded on Nokia phones around 1998[23].

The original Snake game (see figure 5.1) presented a black line, representing a snake, surrounded by four walls, that chased apples across the screen, controlled by the user. The snake could only make 90° turns and the apples were displayed as four dots. When the snake ate an apple, that is, when the ”head” of the line went through the four dots, the snake would get bigger, start to move faster, and another apple would appear somewhere else on the screen. The player would loose the game as soon as the snake touched the walls, or it ate itself, that is, if the ”head” of the snake touched its body. If the snake did not touch the wall or itself, the game would go on until the snake was so big that it could not move anywhere else, filling up the screen. The Snake code sample provided by Google presented a similar game, only with improved graphics.

In order to test our Fault-Tolerant Vector-Field Consistency system, we decided to develop a multi-player version of this game that did not require user intervention to be played. Each player starts with a snake that randomly wanders the map searching for apples. On the contrary of the original version,


Figure 5.1: Snake running on a Nokia 3210 phone.

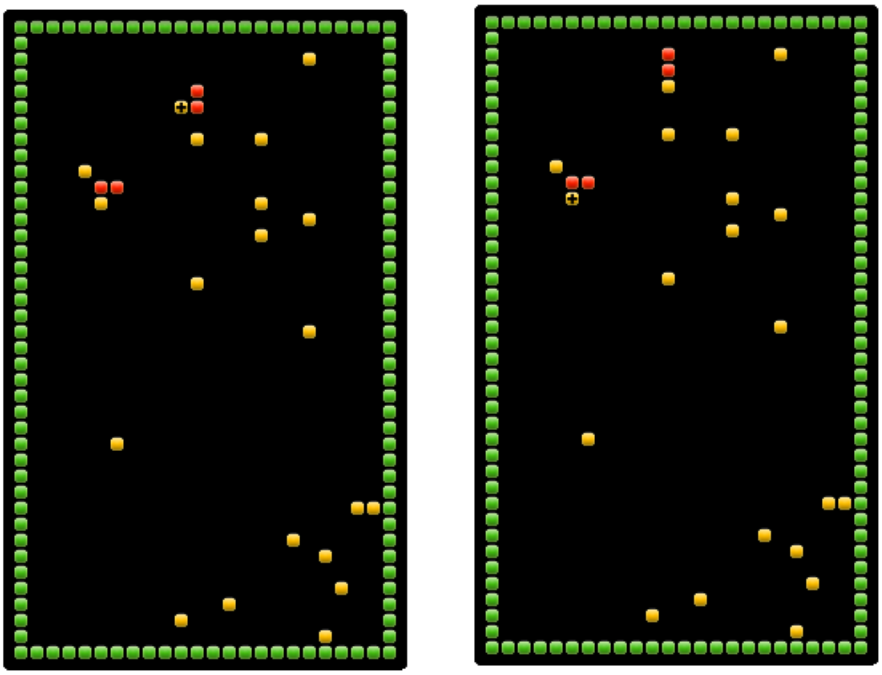

Figure 5.2: Two Snakes VFC clients running side-by-side on Android emulators.

our snakes do not grow or start to move faster, as apples are found. However, with each found apple, the player wins one point.

Figure 5.2 shows two clients of Snakes VFC playing the game side-by-side on two Android emulators. The line of green squares that surrounds the "game map" represents the wall, and the snakes cannot cross it. The green dispersed squares are the apples. The snakes are a sequence of two red squares and a green one. Finally, the snake with the cross on its "head" represents the snake owned by the player.

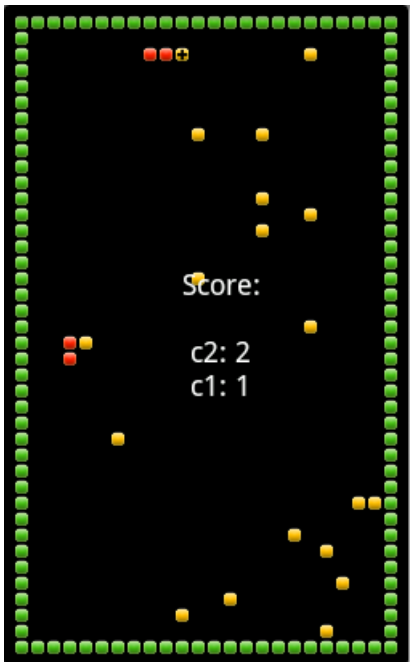


Figure 5.3: One Snakes VFC client running a two-player game and tracking the score.

When the game is running, each player can easily track the score of their snake as shown in figure 5.3, where user with nickname *c2* is leading with a score of 2, and user *c1* only got one apple.

In Snakes VFC, the host of the game must also play it. The user is presented with the menu show in figure 5.4 when he launches the game. From here the user must choose if he is going to host a new game, or connect to an existing one. He then introduces his nickname and selects the type of connection. At the moment there are two available: *Localhost* which is to be used on the android emulators testing environment, and *Tcp Direct* that is used on real devices, on a real wifi network. If the player chooses to connect to an existing server by selecting the *Client* radio button, he must then enter the port (if using android emulators) or the IP address (if using real devices) to establish the connection.

### 5.1.1 Implementation Details

In order to represent the three different game entities, we extended the class *DataUnit* and created the classes: *SnakeDataUnit*, *AppleDataUnit* and *ScoreDataUnit*. The first and third classes represent pivot objects owned by the players, and the apple objects are owned by whatever player is the server at the moment. An auxiliary class that represents a position in the game map, called *Coordinate* was also implemented. The apple objects have a coordinate and a *boolean* that shows if it was eaten or not, and the snake objects possess a dynamic array of three coordinates.

Each turn, the UbiVFC server propagates to every client their dirty objects (see section 3.2.2). When

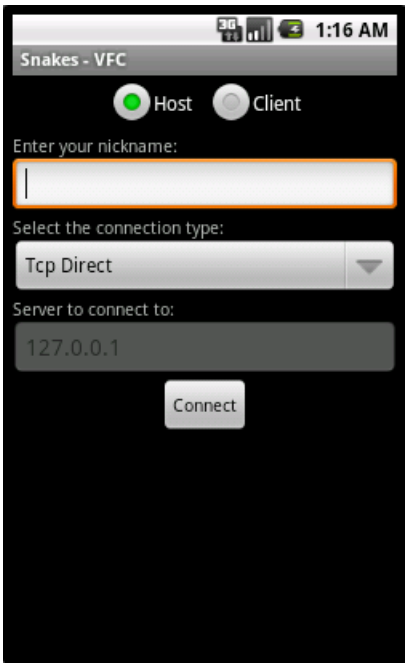


Figure 5.4: Launch menu of Snakes VFC.

a client receives the updates sent by the server it randomly moves the snake one position. The snake will go forward 50% of the time, turn left, 90°, 25% of the time, or turn right the last 25% of the time. This movement changes the snakes inner coordinates, so it sends that object update to the server. If the snake happens to eat an apple, the client also sends the server object updates with the player's score and the eaten apple.

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[1] radius of the pivot. The second zone is the area beyond the 10 tile radius. According to each dimension of the consistency vector, the objects in the first zone are updated when 2 rounds have passed, or there have been 2 updates to the object, or there's a difference of at least 200% from the object's replicated value in comparison to the actual value. This value refers to the distance between the coordinates of two different replicas of the same snake object. When an object is in the second consistency zone it only needs to be updated when 10 updates have been made or when there is a 500% difference between its values. The amount of time that went by since the last update is irrelevant, since a player's snake only moves once per round. So, if the snake object has received 10 updates, that means that 10 rounds have passed.

## 5.2 Quantitative Evaluation Methodology

With regards to the quantitative evaluation, our tests are divided in two different categories: *amount of messages exchanged* and *elapsed time between events*. The amount of messages exchanged between server and clients is an important metric that can show us if our extended VFC system can still present a better solution than not using VFC at all. The elapsed time between certain events, like the departure of the game server, is also a every important metric, because it can show if our solution does not bring more

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

hassle than simply restarting the game when a player crashes, or wants to join.

### 5.2.1 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*.

In all three models the amount of messages sent from the clients to the server is roughly the same. *subscribe* and *publish* messages are always required to be sent from a client for it to play the game. Moreover, each round clients must send the server an updated version of their snake object, so 100 snake update messages are to be expected from each client. The only variable is the amount of messages sent from updates to the score and apple objects, since a player's snake may catch a different amount of apples each game.

Some round update messages using UbiVFC can have a bigger size than usual when a client joins or leaves the game. In this event, as shown in section 4.2.9, a *ServerSyncData* object must be piggybacked to the round message. However, this object can be only partially sent if the backup client is the same as in the previous round. Since the other two consistency models do not support the entry and departure of nodes in the system as the game is being played, we assume that there are no membership changes, so the size of messages sent is all roughly the same.

#### Non-VFC Model

The non-VFC consistency model is simply a client-server architecture where each update the server receives is propagated to the remaining clients at the beginning of the next round. There are no computations required to know when an update is ready to be diffused. At the beginning of each round, every client is up-to-date with the server, but that comes with the cost of constant flooding of the network with update messages.

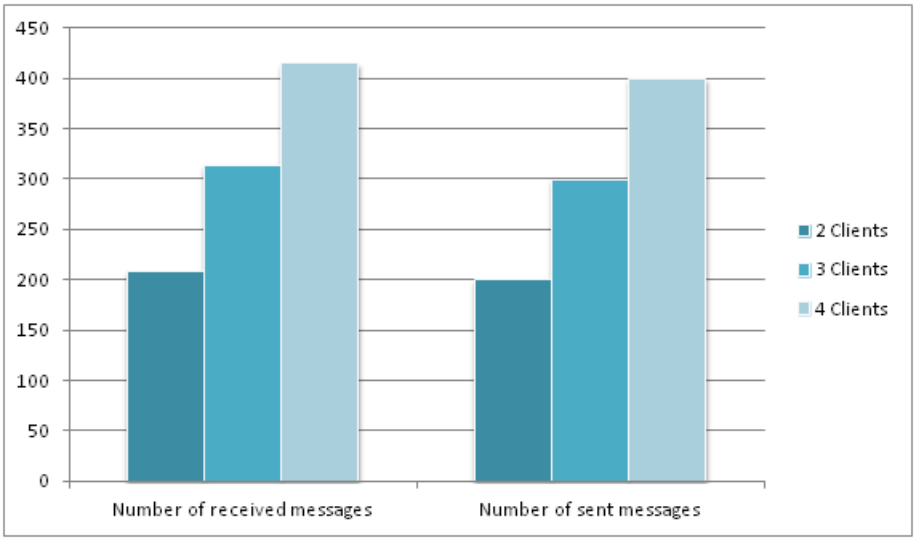


Figure 5.5: Data transmission from a non-VFC server in 100 rounds.

In figure 5.5 we can see the number of exchanged messages between clients and server using a non-VFC consistency model over the course of 100 rounds. As expected, the number of messages sent from the clients to the server is just slightly higher than the proportion of 100 messages per client. The number of messages sent from the server to the clients, however, is exactly the sum of an update sent to each client per round.

### VFC Model

The VFC consistency model executes the consistency settings presented in section 5.1.1. Since each snake object update is only supposed to be propagated to the other clients every other round, if they are close enough, or once every 10 updates if the snakes are at least 10 tiles away, the number of messages sent should be about half the number of messages received by the server.

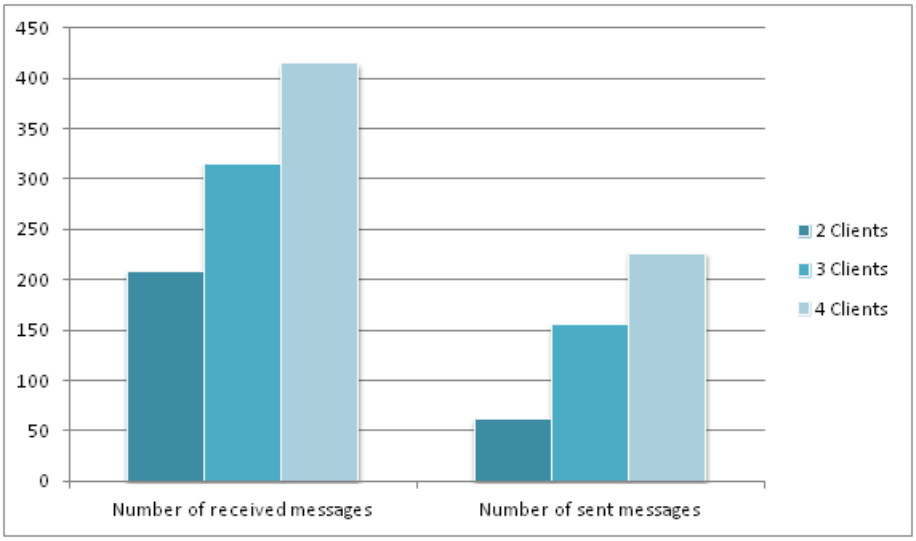


Figure 5.6: Data transmission from a VFC server in 100 rounds.

In figure 5.6 the test results show that the number of messages sent is indeed roughly half the number of messages received when 3 or 4 clients are present. The number of messages sent when there are only 2 clients playing the game is almost a quarter of the number of messages received. This numbers were obtained by calculating the average value of four runs of the game, but we believe that if more runs were performed, the number of messages sent by the server when 2 clients are playing the game would get closer to 100, as expected.

### UbiVFC Model

The UbiVFC consistency model also executes the consistency settings presented in section 5.1.1. However, UbiVFC needs data to be backed-up in order to be able to recover from a server failure. For this reason, and as explained in section 3.4.5, the server must send all available updates to a chosen backup client. The UbiVFC server can also receive a type of message that the previous two models do not: *Battery Info Update*. However, this message is only received sporadically and not relevant for the purpose of our current test.

Figure 5.7 shows the experimental results for the number of messages exchanged between clients and server for 100 rounds of the Snakes VFC game using the UbiVFC consistency model. As with the previous

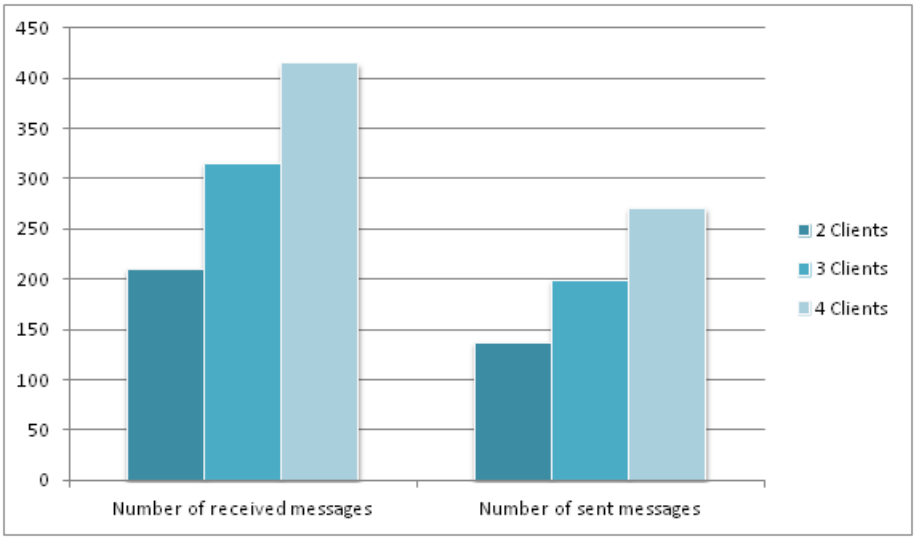


Figure 5.7: Data transmission from an UbiVFC server in 100 rounds.

two consistency models tested, the numbers shown are the average of 4 different runs with 2, 3 and 4 clients playing the game. The test results show that the number of messages sent is slightly less than two thirds of the number of messages received. For instance, for an average of 316 messages received from three clients, the server sent an average of 199 messages.

### Comparison Between Consistency Models

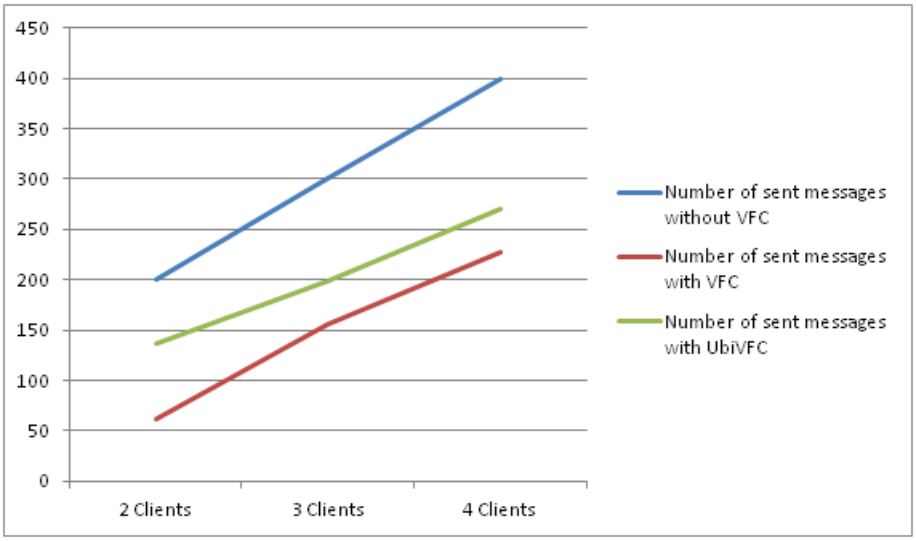


Figure 5.8: Comparison between the number of messages sent from all three models.

Since the amount of messages received by the server is roughly the same using the three consistency models, in figure 5.8 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.

| | **2 Clients** | **3 Clients** | **4 Clients** |
|---|---|---|---|
| Test 1 | 0.594 | 0.824 | 0.956 |
| Test 2 | 0.487 | 0.733 | 0.846 |
| Test 3 | 0.505 | 0.775 | 0.788 |
| Test 4 | 0.483 | 0.670 | 0.768 |
| *Average* | **0.517** | **0.751** | **0.840** |

Table 5.1: Time elapsed until a new client joins a running game (in seconds).

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.

### 5.2.2 Time Elapsed

UbiVFC's main goal was to enable the entry and departure of nodes from the system. To accomplish this, a series of protocols were implemented. The execution of this protocols takes some time so, the objective of this tests is to know exactly how much time do they take to perform. Furthermore, with this results, and comparing them to the time it takes to execute other important protocols in our system, we can draw some conclusions.

The protocols that are tested over the remainder of this section are the most relevant ones from section 3.4, where they were described in detail.

#### Client Publishes Protocol

Table 5.1 shows the amount of time the UbiVFC server takes to process a new client entry when the game is in the active state. The bottom row shows, respectively, the average time it takes to process a second, a third and a fourth client into the system. In this time, the server introduces the required data into the appropriate structures, broadcasts the new objects to the remaining clients, and finally, sends the joining client a subscribe confirmation, the objects that were already on the server's Object Pool, the identifiers that were used by the server to register the objects the client published and an enable message to start the game in the client.

This protocol in reality joins the *subscribe* and *publish* protocols when it is being processed in the active state. Both this protocols already existed on the original VFC design, they could not, however, be executed when the server's session manager is in the active state.

It takes on average **0.517** seconds to process this protocol when there are two clients in the game, **0.751** when there are three, and finally **0.840** when there are four.

#### Client Leaves Protocol

Table 5.2 shows the experimental elapsed time from the moment the server starts the *client leave* protocol until it ends. Contrary to the previous protocol, the leaving client does not need to wait for the server to execute this protocol in order to leave. As soon as the server receives a *leave* request, if it is in the

| | **2 Clients** | **3 Clients** | **4 Clients** |
|---|---|---|---|
| Test 1 | 0.109 | 0.136 | 0.265 |
| Test 2 | 0.109 | 0.151 | 0.279 |
| Test 3 | 0.115 | 0.130 | 0.212 |
| Test 4 | 0.102 | 0.146 | 0.140 |
| *Average* | **0.109** | **0.140** | **0.224** |

Table 5.2: Time elapsed until a client leaves a running game (in seconds).

right state, the client immediately receives a response with the permission to leave the game. When a new round is triggered in the server, it starts this protocol. It takes the time presented in the table to remove the client's data from the appropriate structures and to broadcast to the remaining clients the identifiers of the objects owned by the client that left.

It takes on average **0.109** seconds to process this protocol when there are two clients in the game, **0.140** when there are three, and finally **0.224** when there are four.

#### Client Fails Protocol

A client can fail in two different ways: it can *crash* or it can *stop communicating with the server*. When a crash happens, the connection between the client and the server is immediately closed. This situation can occur in many ways, such as: when the battery runs out, when a client's mobile device looses connection to the network or simply if the game application encounters an error. When a client stops communicating with the server it usually is because its mobile device is under heavy load. When this situation happens the connection can stay up for some time.

Since they can fail in two different ways, two different measures must also be taken in order to remove the failed client from the game:

- *Crash*: In this situation, the server automatically detects the connection failure and removes the failed client from the game at the beginning of the next round;
- *Stops communicating*: The server uses its Failure Detector to acknowledge this failure, and when it is, it removes the failed client from the game at the beginning of the next round.

The actual protocol of removing the client from the game is the same for both this measures. It takes the same amount of time as to process the *client leaves* protocol, less the time it takes to send the client the message that allows its departure. This acknowledgment message only takes on average **0.02** seconds to send, so the impact on the results presented in table 5.2 is minimal.

It takes on average **0.089** seconds to process this protocol when there are two clients in the game, **0.120** when there are three, and finally **0.204** when there are four.

#### Server Leaves Protocol

Table 5.3 shows the amount of time UbiVFC takes to execute the *Server Leaves Protocol*. Since the required backup data is already at the selected backup client, this time is spent by the leaving server

| | **2 Clients** | **3 Clients** | **4 Clients** |
|---|---|---|---|
| Test 1 | 1.031 | 1.434 | 1.951 |
| Test 2 | 1.010 | 1.591 | 1.921 |
| Test 3 | 1.241 | 1.598 | 1.649 |
| Test 4 | 1.121 | 1.502 | 1.689 |
| *Average* | **1.101** | **1.531** | **1.803** |

Table 5.3: Time elapsed until a server leaves a running game and another takes its place (in seconds).

| | **2 Clients** | **3 Clients** | **4 Clients** |
|---|---|---|---|
| Test 1 | 0.390 | 1.002 | 1.016 |
| Test 2 | 0.307 | 0.681 | 1.108 |
| Test 3 | 0.492 | 0.767 | 1.583 |
| Test 4 | 0.365 | 0.601 | 1.091 |
| *Average* | **0.389** | **0.763** | **1.200** |

Table 5.4: Time elapsed until a server takes over the server role of a failed one (in seconds).

on sending a *New Server* message to the backup client and *Connect To New Server* messages to the remaining ones, and by the new server, waiting for the other clients to establish a connection.

It takes on average **1.101** seconds to process this protocol when there were two clients in the game, **1.531** when there were three, and finally **1.803** when there were four.

#### Server Fails Protocol

As it was explained in section 5.2.2, servers can also only fail by *crashing* or by *stopping communicating* with the clients. If the connection goes down, the backup client automatically starts the protocol to assume the server duties and the remaining clients try to establish a connection with the new server. If the server stops communicating, the clients need first to detect that the server has failed with their Failure Detectors. Only then they initiate the game resuming protocol.

Table 5.4 shows the time elapsed from the moment the backup client detects that the server has failed until it resumes the game again. It takes on average **0.389** seconds to process this protocol when there were two clients in the game, **0.763** when there were three, and finally **1.200** when there were four. It takes little time to actually resume the server, since the backup client already possesses the required data to assume the server role. However, as the number of clients increases, the server needs to wait more time for them to reconnect, since they can find that the previous server has failed on different moments. That is the reason why there is a big difference between some of the test results obtained with three and four clients.

#### Comparison Between Protocols and Algorithms Execution Times

Figure 5.9 presents a comparison between the average times UbiVFC takes to execute different protocols and algorithms when there are three clients in the game. We used three clients as our base of comparison since it is the most likely number of players a multi-player mobile game would use.

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

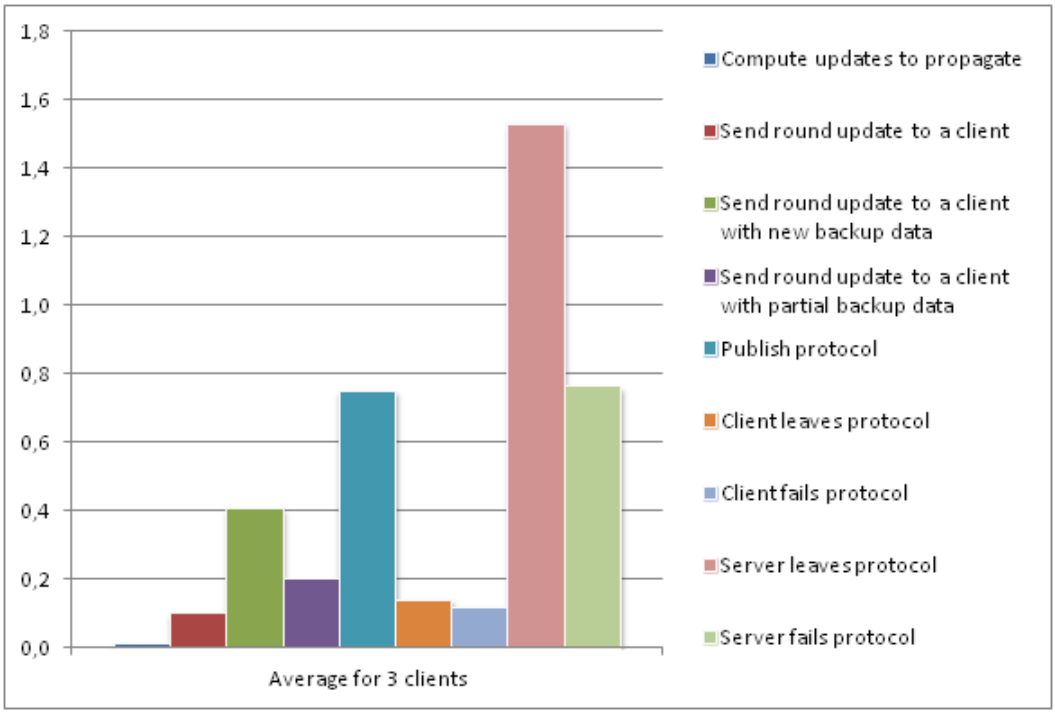


Figure 5.9: Comparison between the time it takes to execute each protocol or algorithm in seconds.

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

The results of the tests performed on the rest of the protocols presented in figure 5.9 were described in detail on the previous sections. 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.

## 5.3 Qualitative Evaluation Methodology

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 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 server's Failure Detector acknowledges this event, the failed client's snake lingers in the game. The time elapsed between a client's failure and the server's Failure Detector finding it can, however, be tweaked by the game programmers. If the time it takes for a Failure Detector 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 Client Failure Detectors wait the number of rounds they are programmed to wait, until they assume that the server has crashed. As it happens with the Server Failure Detector, the amount of time the client's must wait to take over a failed server can be changed by game programmers.


Figure 5.10: Recovery dialog that is presented to the user.

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 (see figure 5.10). 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.

## 5.4 Summary

In this chapter we presented the evaluation results of our system. We started by presenting Snakes VFC, a multi-player distributed game developed to evaluate our UbiVFC solution. We then proceeded to present both the quantitative and qualitative evaluation of our solution, discussing the results obtained.

# Chapter 6

# Conclusion

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 3.2). It is enforced by Mobihoc, which is a middleware platform for multi-player distributed games in ad-hoc networks (see section 2.5.4). 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.

## 6.1 Future Work

In this work we provided the Vector-Field Consistency model with support for the dynamic entry and departure of nodes from the game being played. However, a variety of improvements can always be made to improve on our design and implementation. In this section, we enumerate some of the ideas for future work that arose during the creation of this document:

- Increase the number of connections that can be used by UbiVFC. At the moment, only local android emulators or wifi direct connections are supported. In the future we would like to include bluetooth connections and automatic host discovery over wifi networks.
- Improve the performance of the communication between nodes. This can be accomplished in many ways, such as: improving the protocols implemented, reducing the size of the objects being transferred and improving the performance of the network layer.
- Implement UbiVFC in other games and comparing the effectiveness of our solution to the game's original one.
- Adapt the solution developed to other environments, such as massive multi-player online games or network real time strategy games.
- Expand UbiVFC to other application contexts, like cooperative editing of online documents.
- Implement the VFC generalization of *multi-view* allowing the definition of different views with different consistency requirements for specific objects.

# Bibliography


[1] L. Alvisi. Understanding the message logging paradigm for masking process crashes. *ACM Transactions on Computer Systems (TOCS)*, 1996.

[2] M.-K. Alvisi, L. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, 1998.

[3] L.-S. R. Bhargava, B. Independent checkpointing and concurrent rollback for recovery - an optimistic approach. *In Proceedings, Seventh Symposium on Reliable Distributed Systems*, pages 3–12, 1988.

[4] J.-T. Birman, K. Exploiting virtual synchrony in distributed systems. *11th ACM Symposium on Operating Systems Principles*, 1987.

[5] T.-S. Chandra, T. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[6] L.-L. Chandy, K. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[7] R. R. Chang, E. G. An improved algorithm for decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, 22(5):281–337, 1979.

[8] K. I. V. R. Chockler, G. Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR)*, 33(4):427–469, 2001.

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

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

[11] M. D. Doley, D. The transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 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)*, 34(3):375–408, 2002.

[13] P. F. Ferro, E. Bluetooth and wi-fi wireless protocols: a survey and a comparison. *IEEE Wireless Communications*, 12(1):12–26, 2005.

[14] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *In 11th Australian Computer Science Conference.*, pages 55–66, 1988.

[15] L. N. A. P. M. S. Fischer, Michael J. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[16] T. E. Foundation. Eclipse development platform, October 2011.

[17] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Trans. Comput.*, 31(1):48–59, 1982.

[18] Google. Android development tools, October 2011.

[19] Google. Android software development kit, October 2011.

[20] Google. Snake android code sample, October 2011.

[21] W. J. M. Herlihy, M. P. Linearizable concurrent objects. *ACM SIGPLAN Notices*, 24(4):133–135, 1989.

[22] S. J. M. K. D. D. Keidar, I. Moshe: A group membership service for wans. *ACM Transactions on Computer Systems (TOCS)*, 20(3):191–238, 2002.

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

[24] L. B. S. L. G. S. Ladin, R. Providing availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, 1992.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[26] S. R. P. M. Lamport, L. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[27] D. K. J. Mockapetris, P. Development of the domain name system. *Symposium proceedings on Communications architectures and protocols*, pages 123–133, 1988.

[28] M.-S. P. A. D. B. R. L.-P. C. Moser, L.E. Totem: a fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, 1996.

[29] X.-J. Netzer, R. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, 1995.

[30] Oracle. *Oracle7 Server Distributed Systems Manual.*, volume 2. 1996.

[31] Oracle. Java programming language, October 2011.

[32] S.-M. T. D. T. M. D. A. Petersen, K. Flexible update propagation for weakly consistent replication. *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 288–301, 1997.

[33] R. Rajaraman. Topology control and routing in ad hoc networks: a survey. *ACM SIGACT News*, 33(2), 2002.

[34] B.-K. G. B. G. K. H. M.-H. T. M. D. V. A. V. W. Renesse, R.V. Horus: A flexible group communications system. 1995.

[35] S.-M. Saito, Y. Optimistic replication. *ACM Computing Surveys (CSUR)*, 37(1):42–81, 2005.

[36] 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.

[37] N.-A. C. S. Singh, K. Transman: A group communication system for manets. In *Proceedings of the 8th International Conference on Distributed Computing and Networking (ICDCN), Lecture Notes in Computer Science*, pages 430–441, 2006.

[38] Y.-S. Strom, R. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(3):204–226, 1985.

[39] S. Tatham. Putty, October 2011.

[40] 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*, pages 140–150, 1994.

[41] T.-M. P. K. D. A. S. M.-H. C. Terry, D. Managing update conflicts in bayou, a weakly connected replicated storage system. *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182, 1995.

[42] Y. Wang. Space reclamation for uncoordinated checkpointing in message-passing systems. 1993.