Requirements for a Group Communication Service for FLARE
TCD-CS-2001-45

N. Reijers, Raymond Cunningham, René Meier,
Barbara Hughes, Gregor Gaertner, Vinny Cahill

17th December 2001

Contents

Chapter 1

Introduction

This document explores what the requirements for a group communication service for the Framework for Location-aware Augmented Reality Environments (FLARE) are. This chapter provides an introduction to FLARE. The next chapter will explain the game rules for the first application called Quazoom that we will build using FLARE. Since network partitions are important, we first describe the game rules in the case when there are no partitions, and treat the partitioned case in a seperate section.

Next we examine the group communication requirements. We start by listing the key issues in designing the communication service. Then we examine these, first without failure handling, and then with failure and partition handling. Finally we look at the timeliness requirements. We also provide guidelines for detemining the ordering and timeliness requirements for applications built with FLARE.

1.1 Introduction to FLARE

"Shoot 'em up" style games like Doom and Quake [1] have become very popular in recent years. Paintball, a real world outdoor "Shoot 'em up" game using paint-filled pellets as bullets, and Quasar, an indoor game, which uses laser beams and sensors on the players' suits, have also become very popular.

FLARE is a framework for building augmented reality applications. The first application we will build using FLARE will be Quazoom, a Doom-like game combining the Doom/Quake experience with Paintball/Quasar play to make an augmented reality game where players move around in the real world, while interacting with both virtual and real players. Players will see a Doom-like game on their screens providing them with a virtual representation of the real world. Their real world position will determine their location in the game. There will be both real players and virtual, computer controlled, bots in the game. Players can shoot bots and other players and pick up things like ammunition or medikits as in a normal Doom game.

The bots will operate autonomously in the game under computer control. There will be one node that is responsible for initiating each bot's actions. Deciding on which node this is will probably be done using location information to keep the physical location of the node as close to the virtual location of the bot as possible. A bot will have to autonomously decide that it wants to move to another node and initiate the appropriate protocol to do this.

Players will be able to join and leave the game as they please, but the game will cease to exist when the last player leaves.

FLARE will be run on wearable computers. A wireless ad hoc network will be used for communication, and Differential Global Positioning System (DGPS) and possibly other sensors to determine location. FLARE will be developed in three stages. The first version will use group communication and support fault- and partition-tolerance using a single group for the entire game. Players will communicate through the group, broadcasting information like their new position, or the fact that they have fired their guns. Nodes will only update the game state as a result of receiving a message. The group communication service used will support non-blocking communication even when the network is partitioned, delivering messages to the partition instead of the whole group. The receiving nodes are notified of this, and can respond to this to maintain consistency.

The second version of FLARE will use multiple groups. The game area will be split into multiple zones and different groups will be used for different zones. Different groups will also be used for different interests. For instance, all bots and players in a team could use a team group to coordinate their attack. Thus, the second version will have filtering using multiple groups based on both location and interest. The third version will extend FLARE with event-based communication. This document only considers requirements for the first version of FLARE.

Perhaps the main reason why FLARE is an interesting research application scenario is because it is a game. This means we can chose the rules in such a way as to address the issues in which we are interested. They are not fixed by real world requirements, so we can make things as easy or as hard as we like and explore different directions.

1.2 Consistency

Since this is a game application, users are bound to disconnect suddenly if they get bored with the game. The use of mobile computers and wireless networks also makes failures and network partitions likely. To allow the game to progress as much as possible in this environment, the game state will be replicated on all nodes, and the game will use producer/consumer communication instead of the client-server model that is common for these games. A client-server model would be unsuitable because nodes that lose contact with the server cannot make progress and if the server failed the whole game would stop.

Having replicated data means that we need to formulate consistency require-

ments. The easiest choice would be to require all nodes to have a consistent view of the game, but the game would have to block if the network became partitioned. Since we want to make progress in the presence of partitions as well, we define different levels of consistency. We will decide whether we can update a part of the game state based on that part's required consistency level and the group status.

1.3 Related work

Most publications on augmented reality address tracking and display problems. An overview is given in [2]. The communication problems involved get much less attention from an augmented reality perspective. Related work on group membership and proximity in mobile networks can be found in [3, 5, 7]. A survey of other group communication services, and a formal specification of their properties is given in [12]. Other work on partitionable group communication can be found in [9, 6]. Much work has also been done in the context of the Transis system, for instance in [4]. Work addressing similar timeliness requirements using timestamps is reported in [10] and on causal ordering and timeliness in [11].

Chapter 2

Quazoom Game Rules

Quazoom will be a Doom-like game, whose the functionality will be a subset of the functionality usually found in this kind of game. The rules here were chosen to explore different sorts of consistency guarantees, not to make a fun game.

The game will be played with a variable number of players. Players can join or leave when they like, but at least one player must remain to keep the game running. There are 4 different game objects:

- Player
- Medikit
- Flag
- Bot

Players can move and shoot. Players move around in the game by moving around in the real world. They shoot by pressing the fire button on their keyboard; their orientation is also controlled by the keyboard.

When a player shoots, the first player in the line of fire and within a range of 50 meters, and loses 50% health. When a player's health reaches 0, he is killed and the player that shot him gets a point. A dead player cannot do anything, or be shot, for 15 seconds. Afther 15 seconds, the dead player's health is reset to 100% and he can continue in the game.

There are medikits in the game. A player can pick these up using a keyboard command, but only if he is within 2 meters of the kit. Only one player is allowed to pickup a kit at a time, after which it disappears for 30 seconds. The kit replenishes the player's health to 100%. Players can pickup medikits even if their health is already at 100%, although their health will not increase.

There are flags (possibly more than one) which the player can pick up or drop for a capture the flag type game. Flags have an initial position. When a player carrying a flag dies or leaves the game, it is reset to its original position. It is not dropped because the player will recover soon and we don't want him

to be able to immediately pick it up again. This is different from similar games where a dead player's position is reset to some starting point. Since we cannot reset a dead player's position in FLARE, we reset the flag's position.

There are bots in the world that move around and shoot randomly. They don't pick up medikits or flags, and they don't score points, but in every other aspect function just like normal players.

The first player to reach a score of 3 points wins the game.

2.1 Partitions

When a partition occurs some actions may become impossible or limited. Players should be able to move around freely. They should also be able to shoot other players in their partition, but not players outside of their partition. Players in a partition will see a different graphical representation for the players outside of their partition. Obviously the position of those players will be frozen.

Two players in different partitions may pickup the same medikit, but only one player may pickup a flag. We want all players to agree on who won the game because this signals the end of the game. Therefore we cannot decide on the winner when there is a partition. When a player reaches three points in a partition, the other players in that partition will not be able to win the game. When partitions merge and there is a player that could be declared the winner in both partitions, we will use the time on the local system clocks when they were declared winner in their partition to decide who won first.

2.2 Starting the game

The game is started by one player. After the game is started, other players that can communicate with at least one of the players in the game can join, even if the game is in a partitionedz state. Players that don't start a game can ask for a list of games they can join, and then select one to join.

Chapter 3

FLARE Communication Layer Key Issues

There are five key issues to be addressed in the FLARE communication layer:

1. Do we send events or state updates? (See Figure 3.1 I and II respectively)
2. What sort of message ordering is necessary?
3. What happens to group membership when there is a partition? Possible options are:
   (a) Group is split into two new groups;
   (b) Group is split into two partitioned parts of the same group;
   (c) Members that have lost contact with a designated main node are dropped from the group;
   (d) The membership is reduced to the majority partition.
4. In the presence of partitions and failures, what sort of delivery primitive(s) do we need? Again, possible options are:
   (a) Deliver to as many members as possible, providing an updated group view when the message was delivered to a different set of members than given by the previous view;
   (b) Deliver as many members as possible;
   (c) Deliver only if it can be delivered to member X;
   (d) Atomic delivery to all or none of the members of the group.
5. What are the timeliness requirements?

Issues 2-5 are considered in later chapters. Issue 1 will be considered in the following section.

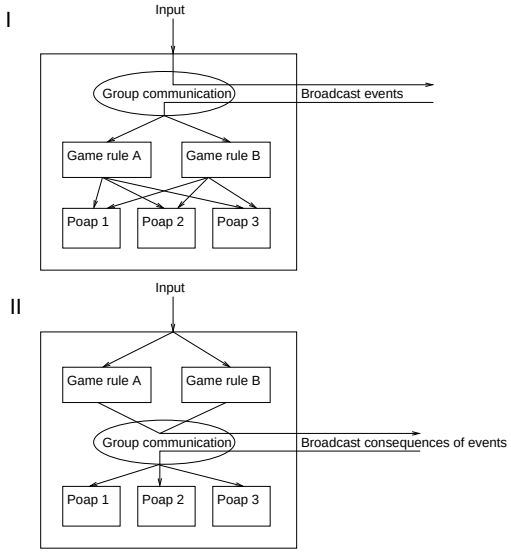

Figure 3.1: Send event or state change messages

3.1 Events vs. updates

The choice between sending events or updates is important because it has an impact on most of the other issues and on the whole design of FLARE. Both options have disadvantages which we will describe here. When we send events, we keep the game state consistent by making sure that the nodes update it in the same way as a result of receiving these events. The event is processed on all nodes. When we just send updates, the event is processed on the node where it originated and then the results are broadcast to the other nodes.

Disadvantages of sending events (option I):

1. Stronger ordering is necessary to let all members respond to the events in the same way. The exact ordering requirements will be examined in Section 4.8.

2. May cause problems when used in the presence of partitions when we want consistent update involving state information located in different groups. Because of the partitions, some replicas may not be accessible. It will be hard to achieve all-or-nothing semantics for those actions.

Disadvantages of sending state updates (option II):

1. Some sort of locking scheme is needed to ensure serializability of transactions.

2. This locking will cause problems when partitions occur, and could cause deadlocks if we're not careful.

Chapter 4

Requirements for the FLARE Communication Layer in the absence of Failures and Partitions

When we assume there are no failures or network partitions, the two issues that remain are points 1 and 2. Now, since a locking scheme will require an extra round of communication and total ordering typically requires two rounds of communication, even if we have to use total ordering everywhere, option I wouldn't be more expensive than option II. We will later see that we are able to relax the total ordering requirement in some cases. The locking mechanism required for option II will be more complicated in general, and especially when partitions occur, than the first option. Since FLARE only employs of one group, the problems that might arise with multiple groups won't surface. Therefore we will chose option I for FLARE 1. This answers issue 1, The ordering requirements will be examined in Sections 4.4 through 4.8. Now the question is, can we relax the total ordering requirement of this option?

4.1 Consistency requirements

We split the application state into things we call poaps (Part Of APplication state). These poaps will be fully replicated on all members. Poaps will *only* be updated as a result of receiving messages from the network and are only accessible in *one* group. For all members that have received the same *set* of messages, we want all copies to be consistent. When there are messages that have been delivered to some, but not all members, we don't mind if the state of different members is not the same.

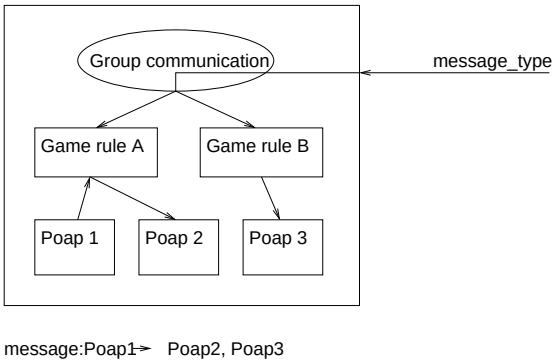

Figure 4.1: Example of a message type signature

4.2 Assumptions

- Without failures or partitions we assume every message sent will always be delivered to all members of the group
- The members of a group are *processes*
- A process runs on a *node*

4.3 Definitions

First let's define exactly what a message does. Each message that is broadcast will be delivered to all members. Upon receiving a message a member may, as a result of the game rules associated with that message, update its local copy of some poaps. A member may *only* update poaps as a result of receiving a message. For deciding if and how to update a poap it may use other poaps as input. A member may also send another message in response to the one it received.

Message signature A notation for writing the effects of a message:

$$msgT(data) : msgTIn \rightarrow msgTOut$$

with

$$msgTIn = \{pk, ..., pl\} \quad \text{and} \quad msgTOut = \{pn, ..., pm\}$$

This means that as a result of a message of type $msgT$, the game rules *may* update poaps $pn, ...pm$ and they base the value for the update of the output poaps on $pk, ..., pl$ and on the $data$ in the message. This is illustrated in Figure 4.1. The illustration shows a message which is associated with two game rules. This message type would have a signature $message\_type(\emptyset)Poap1 \rightarrow Poap2, Poap3$.

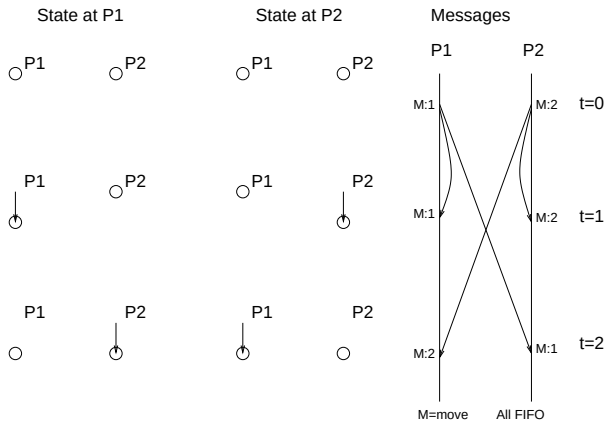

Figure 4.2: Using FIFO ordered messages for movement

Message types Using this signature, we can categorise messages into 3 types:

- $msgT(data) : \emptyset \rightarrow msgTOut$ Unconditional update: messages that don't require any input.
- $msgT(data) : msgTIn \rightarrow \emptyset$ Condition: messages that don't update any poaps, but just send another message
- $msgT(data) : msgTIn \rightarrow msgTOut$ Conditional update: messages that have poaps as input, and update poaps

4.4 Movement

Let's first consider the case where players just move. If we relax the ordering to simple FIFO ordering, the states of different members may no longer be the same.

In Figure 4.2 we see P1 and P2 both moving. They both send out a FIFO ordered message. Since they will receive their own message before the other one's message, they will process their own movement first. This is shown at time $t = 1$ when the game state is inconsistent. But when the other two messages arrive at $t = 2$ we see the game state is consistent again. So our consistency requirement, that the state should be consistent when members have received the same set of messages, is satisfied.

Since FIFO ordering is much cheaper than total ordering, and movement messages will be quite frequent in an application like FLARE, this is a big improvement over using totally ordered messages for movement. However, we have only considered movement, and have to see if this less costly approach will still work when we add the other functions to FLARE. Using our message signature, each player X would be sending messages of this signature using FIFO ordering:

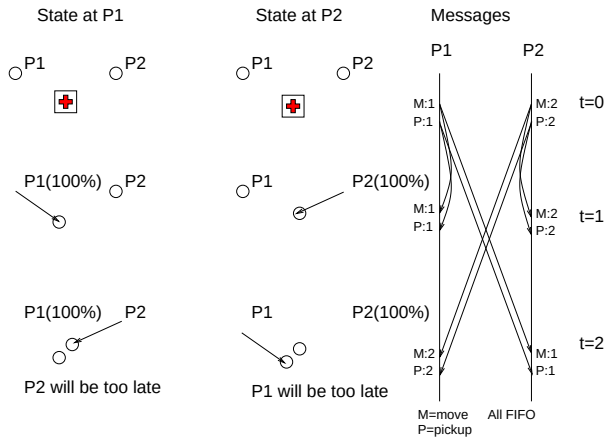

Figure 4.3: Picking up a medikit using FIFO messages

$move(newPosition)\emptyset \rightarrow playerXpos$

Decision We will use FIFO ordering as long as the other game rules permit this. Since movement messages will be the most common messages in the game, this should save a lot of bandwidth, compared to using totally ordered messages. Also, the messages should be delivered quicker, reducing the game lag.

4.5 Medikits and flags

Picking up a medikit or flag, which is essentially the same when no partitions occur, is done when the player presses the pickup medikit or pickup flag button and is close enough to the kit or flag. When a player wants to pick up the medikit he sends a pickup message to the group. When a player receives a pickup message, he checks if the sending player is in range and if the medikit is available, if so he sets the sender's health to 100%, and the medikit to unavailable. The message signature for this is:

$medikitPickUp()playerXpos, medikitStatus \rightarrow playerXhealth, medikitStatus$

In Figure 4.3 we see P1 and P2 moving towards the medikit. Both players immediately press the pickup key, so both messages are sent very close to each other. Now we see that using FIFO ordering, both players receive their own messages before receiving the other player's message, and they both decide that they get the medikit. When they receive the other player's message, they update its position, but the medikit will be gone, so the other player can't pick it up.

Here we see that after P1 and P2 have received the same messages the position is consistent, but both players think they've picked up the kit, so the health won't be consistent. Clearly FIFO ordering doesn't work for picking up medikits.

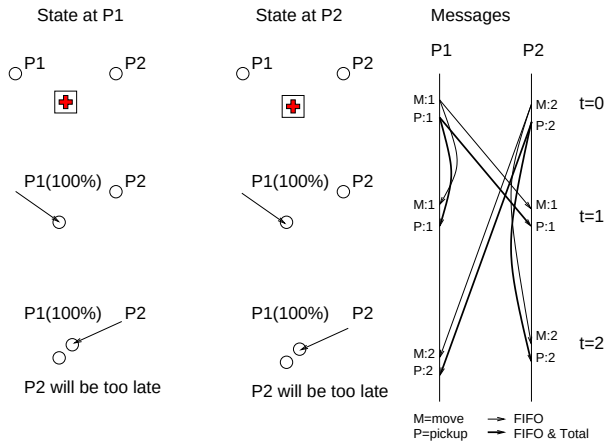

Figure 4.4: Picking up a medikit using FIFO total ordering

4.5.1 Using total ordering for the pickup message

This problem can be solved by using totally ordered messages for the pickup command. Whether this solution works depends on how the FIFO and totally ordered messages are delivered.

Totally and FIFO ordered pickup message First let's consider the case when all totally ordered messages are FIFO ordered as well, so all position messages prior to the pickup message are delivered before the pickup message and all messages sent after the pickup message are delivered after it.

In Figure 4.4 it is clear that both players will reach the same conclusion because the total order on the pickup messages ensures they process them in the same order. The FIFO order ensures that P1 has moved close enough to the kit before executing the pickup command.

Totally ordered, but not FIFO ordered pickup message This changes if we have seperate queues for totally ordered and FIFO ordered messages. This may be desirable because it would allow us to not have the pickup messages cause delays for the movement messages. In this case it is not certain that both P1 and P2 have processed the move message when they execute the pickup command. Let's forget about P2 picking up the kit for now. Figure 4.5 shows how P1 and P2 could reach different conclusions on whether P1 picked up the medikit.

In this figure we see that because there is no FIFO ordering on the movement and pickup messages P1 gets the movement message first, updating its location to be close to the medikit, and then gets the pickup message and picks up the medikit. However P2 gets the pickup message first, and then the movement message. When it processes the pickup message, P1 will not be in range of the

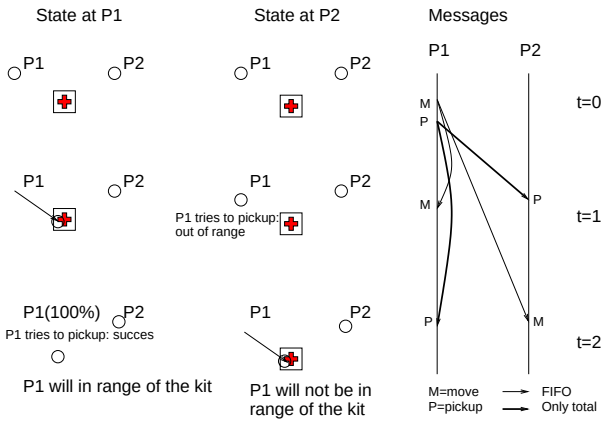

Figure 4.5: P1 and P2 reach different conclusions when picking up using seperate queues for total and FIFO ordering (so the FIFO and total ordering messages may arrive in a different sequence than the one in which they were sent)

medikit, and therefore it will not be picked up.

This problem will probably be rare, since totally ordered messages take longer to deliver than FIFO messages. However, the same thing could happen with P1 moving out of range on one node before the pickup command is processed, while it could still be in range on another.

Packing the player position in the pickup message. An easy solution for this is to have the current position of the player in the pickup message. So the message would not be 'I want to pick up the medikit', but 'I want to pick up the medikit, and I am at position X'. The message signature would change to

$medikitPickUpAt(playerXpos) : medikitStatus \rightarrow playerXhealth, medikitStatus$

This way the pickup message no longer depends on the previous movement messages, but the position in the message may not be the current location at the receiver. Since the pickup message is totally ordered, it will probably take longer than a FIFO message, and so it is likely that the position in the message is older than the location at the receiver.

4.5.2 Making a single member responsible for the medikit

An alternative to the single pick up message is to make one member responsible for the medikit. We can use a FIFO message for a pickup request. The member responsible for the medikit can then determine if the kit is available and, since the message is FIFO ordered, determine if the player is in range. If the player picks up the kit, the node responsible for the kit sends a message to inform the group that a certain player has succesfully picked up a medikit. The message signatures for this would look like:

$medikitPickUpRQ() : playerXpos, medikitStatus \rightarrow \emptyset$

$medikitPickUpGranted(playerXid) : \emptyset \rightarrow playerXhealth, medikitStatus$

Right now we don't know what the ordering requirements will be for the acknowedge message, since this will depend on how we implement the shooting. There may be a race condition on the players health when a $medikitPickUpGranted$ message for a certain player and a shoot message for a shot that will hit that player at in the system at the same time. How these relate is discussed in Section 4.8.

Also note that the players trying to pickup the medikit doesn't have to wait for the outcome. If he gets the kit he can increase his health at the moment the $medikitPickUpGranted$ message arrives. If he doesn't get it, he doesn't have to do anything.

4.5.3 Appearance to observers

What will these different options look like when a player tries to pickup a medikit? Assume we use the $medikitPickUpAt$ message and it is totally ordered, but not FIFO ordered with the movement messages.. A player might be seen, even by himself, to pickup a medikit, while he has already passed it, because the pickup message was delivered after a subsequent movement message.

Assume we use $medikitPickUp$ messages which are both totally and FIFO ordered with the movement messages. The player will then always be seen in range of the medikit by every player. However, when two players are trying to pickup the same kit, it is possible that the player that arrived last gets the kit first. The first player will not be seen attempting to pick it up until after the last player picked it up. Also the total ordering of the pickup message will cause a lag in the delivery of the movement messages.

Assume we make a single member responsible for the kit. There are now two events: trying to pick up the kit($medikitPickUpRQ$) and actually getting the kit ($medikitPickUpGranted$). Since $medikitPickUpRQ$ is FIFO ordered, all observers will see the player in the correct place when the message is processed. But since $medikitPickUpGranted$ has to come from another member, the player may already have moved on when he actually gets the kit.

4.5.4 Decision

We decided to use $medikitPickUpAt$ with total, but not FIFO message ordering. We don't use FIFO ordering because we don't want to have the pickup command delay the delivery of movement messages.

4.6 Shooting

For shooting the rules require that the first player in the line of fire gets hit. He loses 50% health. If his health reaches 0 he dies, but is later restored to 100% health, and a point is awarded to the player that fired the lethal shot.

Again we can have a race between two players firing at a third. This is equivalent to two players wanting to pick up a medikit, so we expect to need total ordering to have all members reach the same conclusion. Without total ordering, both players would probably decide that they shot the third player, like the two players picking up a medikit in Figure 4.3.

Obviously, deciding who gets shot also depends on location information. In fact it depends on all the players' locations, instead of just the player firing the shot, because any of them could be hit.

4.6.1 Totally, but not FIFO ordered messages

We could use the same approach as we decided to use for the medikit, a totally ordered shoot message containing all the players' locations, as viewed by the shooter at the time the shot is fired. In the medikit case, we could ensure all members used the same location by using total FIFO ordering, which would ensure all members had received the same sequence of movement messages when they processed the pickup message. Since the result of firing depends on the location of other players as well, this will not work. The position of the firing player would be consistent everywhere, but the positions of the others would not. For this to work, we would need to flush all the movement messages in the network before processing the shoot message, which would be very expensive. So we need to send the locations of all players along in the message for a single shoot message to work:

$shoot(player1pos, ..., playerNpos, direction) : player1health, ..., playerNhealth$
$\rightarrow player1health, ..., playerNhealth, playerXscore$

4.6.2 Using shoot and die messages with FIFO ordering

We can take another approach to deciding who died. P1 could send a message 'I shot in this direction, while I was at position X', and then all players could decide if they got hit by this shot. This is basically the same idea as making a single member responsible for the medikit pickup action.

In Figure 4.6 we see P1 and P3 shooting at P2. Both send out a message indicating their position and orientation. These messages arrive in different orders at different members. When P2 receives the first message it concludes it has died and sends out an 'I died because of P3's bullet' message. When it receives the second it will discard it because it is dead. In this scheme, which just requires FIFO messages, P2 acts as a sequencer for the shoot messages, and therefore all members will give a point to P3.

$shotFired(playerXpos, direction) : playerYpos, playerYhealth \rightarrow \emptyset$
$die() : \emptyset \rightarrow playerYhealth, playerXscore$

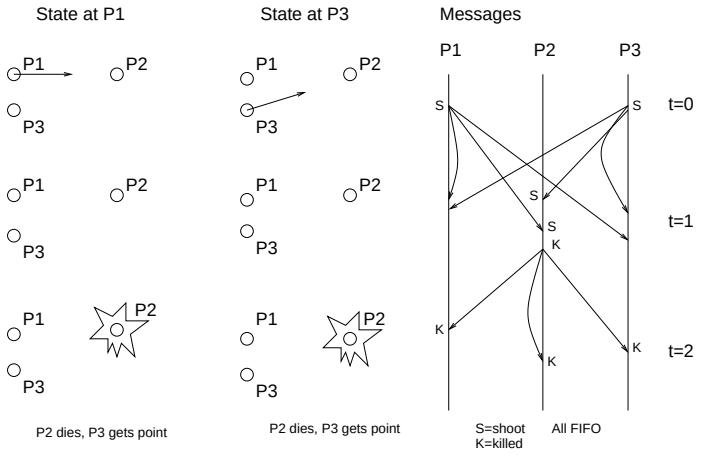

Figure 4.6: Shooting using FIFO messages

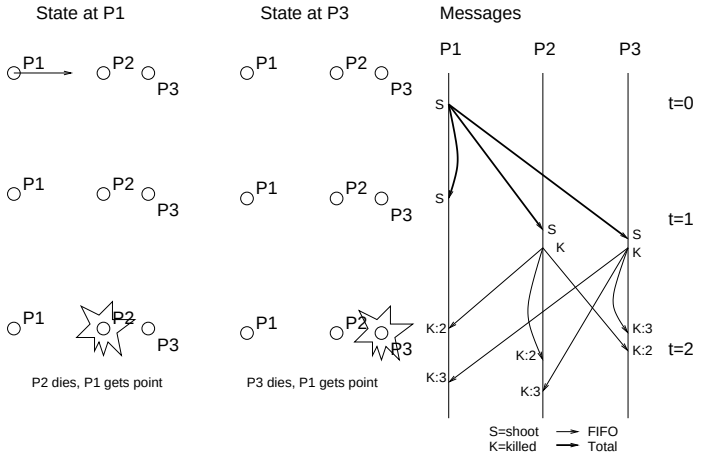

Figure 4.7: Three problems with FIFO die messages

4.6.3 Using a totally ordered shoot and request to die messages

Unfortunately, there is a problem with this approach. Although we have solved the race on who shot P3, there may also be a race as to who died. Figure 4.7 illustrates three problems with this approach. First, when there are two players in the line of P1's fire, both will think they've died, and at $t = 1$, both send a die message (the fact that players may die in different orders on different members may also be a problem).

This can be easily solved by making the message a 'request to die by bullet X' instead of a 'die' message, like the medikit message which only indicates that the user wants to pick up the kit and not that he has picked it up. The first player whose request to die is received will actually die, the others will be discarded.

This makes sure only 1 player dies, but a second problem arises, as is shown at $t = 2$. We now need total ordering to ensure all players agree on which player this is. Third, the total ordering, may be the order that is seen at P3, where the second player in the line of fire dies. This would only be solvable by waiting for a message from all players in the game stating whether they have been hit or not, which would be a costly thing to do. In fact, in this case, it would be enough just to have each player report their position at the time the shoot message arrived. Basically we have just inserted another round of communication to get the current position from all players.

4.6.4 Appearance to observers

What would these two approaches look like to the players?

Totally, but not FIFO ordered messages: Assume we use the *shoot* message type. When a player fires he sends his current information on all the other players' location in the message. This means that if a player sees another player in his crosshairs, he knows he will hit him. However the player that is being shot at may have moved out of the way, and he will get hit by a bullet that looks like it missed him. He has no time to move out of the way, because the outcome has been determined at the time of firing.

In Figure 4.8 we see P1 shooting at P2, and P2 trying to dodge the bullet. Since at $t = 0$ P1 sees it is firing directly at P2, the location information in the shoot message will cause P2 to die. However P2 is already moving out of the way. At $t = 1$ P1 receives the shoot message and concludes P2 died. P2 gets its own movement message and moves out of the way. At $t = 2$ P1 sees P2's corpse move, and P2 dies because of a bullet that looks like it should have missed him.

Shoot and die messages with FIFO ordering: Assume we use the $shotFired$ and *die* messages. This approach will allow P3 to dodge the bullets, since he will decide whether he's hit using his more recent location when he receives the shoot message. The fact that the die message is FIFO as well ensures that the

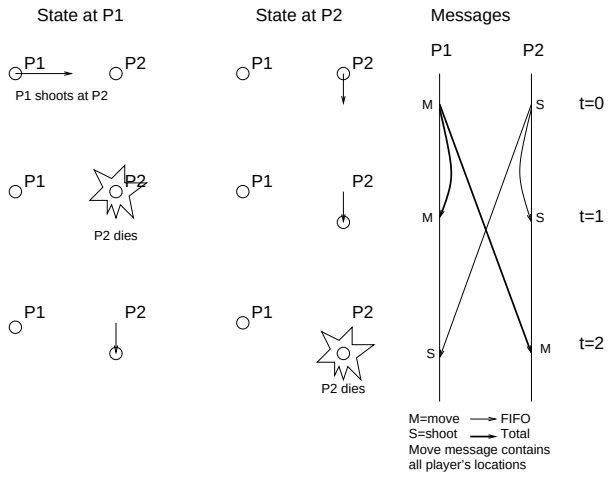

Figure 4.8: Shooting using totally ordered message containing location information

moving corpse anomaly that we discovered in the previous approach will not occur. However a player that sees it is shooting straight at another player may miss because the player has moved when it receives the shoot message.

4.6.5 Comparison

Let's compare the costs of these options. The first option, *shoot* messages, a totally ordered shoot message containing the location of each player as this was know at the shooting player, costs just one totally ordered message per shot.

For the second option, *shotFired* and *die* messages, we gave 3 versions, although the first two are not guaranteed to follow the game rules:

The first version, when two players can die from the same bullet, costs one FIFO message for the bullet and a FIFO message for the hit. If a totally ordered message costs two rounds this will be cheaper when a player doesn't alway hit another player, (but more expensive when he hits more than one with a single bullet).

The second version, only one player dies per bullet, but this may not be the first one, will cost a FIFO message for each shot, and a totally ordered message for each hit. If we just consider the case where there's only one player in the line of fire, this will be cheaper than the first when less than half of the bullets hit a player.

The third version, which, like the first option, works according to the specified game rules, will cost $1 + n$ FIFO messages per shot, where n is the number of users (or n messages if players can't hit themselves).

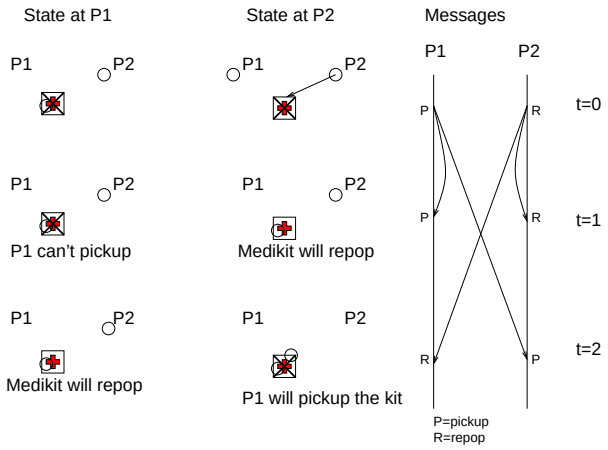

Figure 4.9: Medikit repop requires totally ordered message

4.6.6 Decision

We decide to use a single totally, but not FIFO ordered shoot message which includes the position of all the players. We expect the difference of being able to dodge the bullets or not to be minimal if the network is fast enough. This scheme will also be the simpelest to implement, since we don't have to deal with the case where a player is hit before he knows the outcome of the previous bullet that may have hit him.

4.7 Repop of medikits and bots

So far, all the activity in the game was triggered by user actions (moving, shooting and picking up). Repop of a medikit and movement of bots are triggered by time and bot actions are random. The easiest way to do this would be to assign responsibility for each medikit or bot to a single member. This member would send a totally ordered medikit repop message 30 seconds after it has decided the kit was picked up. Total ordering is required because another member may be trying to pick it up at the same time as the repop message is sent. Figure 4.9 demonstrates the problem that arises when FIFO messages are used. Total ordering would ensure all sites process the repop and pickup in the same order.

A bot would function in the same way as a normal player. So the member responsible for controlling the bot would just randomly generate the same messages he would produce for a player when he shoots or moves.

Of course, when a player quits, his responsibilities must be taken over by another member.

4.8 General case

Let's examine the general case when we want to maintain consistency as described in Section 4.1. This section will only deal with maintaining consistency, and the rules specified in this section only ensure consistency, they do *not* ensure that the game rules are respected. That is a seperate and application specific issue.

4.8.1 General consistency rules

Output For output consistency, we are only concerned with *unconditional update* and *conditional update* type messages, since these are the only ones that write to poaps. Now for each poap px, we can define a set of message types that write to this poap:

$$update(px) = \{msgT : px \in msgTOut\}$$

To maintain the required consistency all messages in such an update set must be delivered in the same order everywhere.

It is easy to see why. If there are two nodes that receive messages from an update set in different orders, then at some point, they may have received the same set, but since the last message they have received could be different, they can have different values for some poaps, thus violating our consistency requirement.

Input When the messages of an update set arrive in the same order at every member, we still need to make sure they actually write the same values, which was assumed in the previous paragraph. To do this, we need to look at the different input types. There are 3 types of data the members could use as input:

- Data contained in the message's *data* part.
- Poaps
- Local data: anything else that is not in the message or in a poap (for instance a randomly generated number)

Which message type can use which input types? The data in the message will be the same for every member, so all types can use this data. We have no way of ensuring consistency of local data, so updating poaps based on local data can lead to inconsistency. We therefore restrict the use of local data to condition type messages. Poaps can, by definition, only be used by local and conditional updates.

Now, since conditions don't update poaps, and unconditional updates only use data in the message, we have no consistency concerns for these messages. But the conditional update type message can cause problems if the input poaps are

not in a consistent state when the conditional update is processed. This can be the case given our consistency requirements, if some nodes have and others haven't yet received some update for a poap.

We define a read set for poap px, similar to the update set:

$$read(px) = \{msgT : px \in msgTIn \ \wedge \ msgTOut \neq \emptyset\}$$

This is the set of all *conditional update* message types that use px as input.

NOW TO ENSURE THAT THEY USE THE SAME INPUT VALUES WHEN THE MESSAGE IS PROCESSED, AND THEREFORE WRITE THE SAME OUTPUT, WE REQUIRE THAT ANY MESSAGE OF A TYPE IN $update(px)$ HAS BEEN DELIVERED TO ALL OR NONE OF THE MEMBERS WHEN A MESSAGE OF A TYPE IN $read(px)$ IS DELIVERED.

Rule specific relaxtion of requirements When we know the semantics of the application rules the message will trigger, we may be able to relax these requirements. For instance, if we know the message type $msgRel(data) : \{px\} \rightarrow \{px\}$ just takes the current value of px and increases or decreases it by a given amount, it is clear that the result of a given set of messages in any order will be the same.

Summary

- Update set of messages:$update(px) = \{msgT : px \in msgTOut\}$
- The messages in a poap px's update set, $update(px)$ must be delivered in the same order to all members.
- Read set of messages:$read(px) = \{msgT : px \in msgTIn \wedge msgTOut \neq \emptyset\}$
- Any messages in $update(px)$ must have been delivered to all or none of the members when a message in $read(px)$ is delivered.

4.8.2 Relation to ordering primitives

If we forget about the rule specific optimizations, we can say that the set of messages $update(px)$ should be totally ordered to ensure *output* consistency, and if the set $update(px) \cup read(px)$ is totally ordered, *input* consistency is guaranteed. This follows from the summary in the previous section. Note that this is stronger than strictly required because a set of messages in $read(px)$ may be delivered in any order as long as no message from $update(px)$ is delivered in between.

If we use total ordering, all members receive exactly the same set of messages and will therefore always be consistent. The interesting question is, in which cases we can use the cheaper FIFO ordering.

We count the number of members that can send messages from $update(px) \cup read(px)$. If this is 1, we call px a *local poap*, if it is greater than one, we call px a *global poap*.

USING THIS, IT IS EASY TO SEE THAT FOR GLOBAL POAPS, USING TOTAL ORDERING FOR $update(px) \cup read(px)$ MAKES SURE THE SET IS TOTALLY ORDERED, BUT FOR LOCAL POAPS, JUST USING FIFO ORDERING WILL ALSO MAKE SURE THE SET IS TOTALLY ORDERED.

There is actually no ordering requirement on the messages in $read(px)$, it is sufficient that if a message x from $read(px)$ is delivered at some node, and the last message to be delivered there from $update(px)$ was y, then y should be the last message delivered from $update(px)$ everywhere. So for global poaps, FIFO or no ordering may be used for $read(px)$ if we use a flush protocol, flushing all messages from $read(px)$, before each message from $update(px)$ is delivered. It will depend on the output ordering requirements of the messages in $read(px)$ and the ratio between reads and writes if this is advantageous. In most cases it won't be.

Things get a bit more complicated when a message type is in different sets. For instance if a message reads a local poap pn, FIFO ordering could be used, but it writes to a global poap pm, so total ordering is required for that. In this case the message should be both totally ordered with the other totally ordered messages in the system and FIFO ordered with the other FIFO messages the member sends.

If the desired totally and FIFO ordered primitive is not available, we either need to think of different messages to achieve the same effect, or use total ordering on *all* messages in $update(pn) \cup read(pn)$ as well.

4.8.3 Ways to eliminate conditional updates

The previous section shows that conditional update types can lead to expensive ordering requirements. There are two ways to eliminate the conditional update type messages:

1. Packing the input poaps' values in the message. This makes it an unconditional update type message, taking it out of $read(px)$.
2. Making a single node responsible for making the decision. This results in a condition type message, after which the responsible node sends an unconditional update type message.

Which of these types is appropriate depends on the message signature and the application rules. For instance, if the message signature has the same poap in the input and output set, although the first method would ensure consistency, it may result in incorrect behaviour.

An example of this is picking up the medikit: $medikitPickup(playerPos)$: $\{medikitStatus\} \rightarrow \{medikitStatus, playerHealth)$, where if we pack the medikit status in the message, two players may pickup the same kit.

4.8.4 FLARE

Let's see how this works in FLARE. First we examine the movement message:

Movement The signature of the messages we use to update a player's position is:

$move(newPosition)\emptyset \rightarrow playerXpos$

If we only consider movement, we can now say that $read(playerXpos) = \emptyset$ and $update(playerXpos) = \{move\}$. Since there is only one node sending messages for each player X, $playerXpos$ is a local poap, so FIFO ordering is sufficient.

Medikit pickup When we add the medikit, things change. The first idea for medikit pickup was:

$medikitPickUp()playerXpos, medikitStatus \rightarrow playerXhealth, medikitStatus$

Now we see that

$read(medikitStatus) = \{medikitPickUp\}$, $update(medikitStatus) = \{medikitPickUp\}$

$read(playerXhealth) = \emptyset$, $update(playerXhealth) = \{medikitPickUp\}$

$read(playerXpos) = \{meditkitPickUp\}$, $update(playerXpos) = \{move\}$

Since we know that a player can only pickup medikits for himself, $playerXpos$ is still a local poap. Therefore, FIFO would be enough to ensure the consistency of poap $playerXpos$. But this is different for $medikitStatus$. More than one member can send messages from $read(medikitStatus) \cup update(medikitStatus)$, so this is a global poap and total ordering must be used here.

If we use total ordering for the *medikit* message, we need to know if this will respect the FIFO ordering requirement of $playerXpos$. Since we said that FIFO and totally ordered messages will go onto different queues, it won't. We now have two options, either use total ordering for $read(playerXpos) \cup update(playerXpos)$ as well, or pack $playerXpos$ in the message, removing *medikit* from $read(playerXpos)$:

$medikitPickUpAt(playerXpos) : medikitStatus \rightarrow playerXhealth, medikitStatus$

As discussed in Section 4.5.4, we have chosen the latter, and now our consistency guarantees are satisfied.

Shooting We explored two ways of implementing shooting. This first option was a single conditional update type message:

$shoot(player1pos, ..., playerNpos, direction) : player1health, ..., playerNhealth$

$\rightarrow player1health, ..., playerNhealth, playerXscore$

This means $playerXpos$ is still a local poap since the player's positions are contained in the message, and $playerXheath$ and $playersXscore$ are global

poaps. We therefore need total ordering.

The second option for shooting was a $shotFired$ and die message:
$shotFired(playerXpos, direction) : playerYpos, playerYhealth \rightarrow \emptyset$
$die() : \emptyset \rightarrow playerYhealth, playerXscore$

Since $shotFired$ is a condition it is *not* in in any *read* set. The die message makes $playerXscore$ a global poap and $playerYhealth$ was global because of $medikitPickUpAt$, so we need total ordering for die.

If the health is just read by the player's own node, we wouldn't have to have a poap for it. And if the die message only updates $playerXscore$ by increasing it by 1 point, we can relax the ordering requirement for this specific message to FIFO since it doesn't matter in which order two messages arrive if they just increase the score by 1.

4.9 Creating and joining the group

So far, we have assumed there is a group with a number of members. Here we will define how we want to allow processes to become a member of the group, and how the group is created.

4.9.1 Joining the group

A process can get a list of groups (of a certain type) that it is able to join. A process can only join the group when it is in contact with at least one node who is a *join point* for that group.

A *join point* is a node that is running a process that is a member of the group or the node that has been *designated* as the join point for an empty group. There can be any number of normal join points, but only one designated join point, and if such a designated join point exists, there are no normal ones. When a process joins an empty group, the node running the joining process becomes a join point, and the designated join point disappears.

4.9.2 Leaving the group

When a process leaves the group a new group view is installed. If the last *process leaves*, its node becomes the designated join point for the empty group. If the last *join point fails*, the group ceases to exist. The group exists as long as there is a join point for the group.

4.9.3 Creating the group

There is a primitive 'create group' which creates an empty group with the node that is running the process that called the create primitive as the designated join point.

4.10 Required API

```
join(name);
leave();
name[] list_groups();
create_group(name);
fbcast(msg);
abcast(msg);
// Delivers a message sent by fbcast or
// abcast. No ordering is imposed on messages
// using different send primitives. Whenever
// a fbcast or abcast message is ready to be
// delivered they are put into a common queue
// and deliver delivers the first message in
// that queue.
deliver(*msg);
// Callback. Will be called whenever nodes join
// or leave. The group view contains the list of members.
view_change(new_view);
```

Chapter 5

Requirements for the FLARE Communication Service in the Presence of Partitions and Failures

When we assume failures and partitions can occur, we must define a failure model, and open issues 3 and 4 from Chapter 3 become important. Furthermore, we need to specify our consistency requirements in the presence of partitions. In the following section we will sometimes just use the word 'partition' when it really means 'group or partition'. For instance a 'when a partition splits' should actually be 'when a group or partition splits'. In most of the discussion here, it is not relevant whether the group is complete or is in a partitioned state, so the whole group and partition are equivalent.

5.1 Assumptions

- There are globally unique identifiers for nodes
- There are globally unique identifiers for members
- There are globally unique identifiers for group views
- Communication failures are bi-directional
- Communication is always transitive

5.2 Consistency requirements

When there is no partition, we want all replicas of poaps to be consistent, and we can always use the most recent information. But when a partition occurs, we may want to forfeit some consistency for some poaps in order not to block the application, while other poaps will have strict consistency requirements. We have identified three different levels of consistency we want to allow:

1. Inconsistent: we will allow the state in partitions to diverge. When partitions merge again, this will require a poap-specific merge function.
   An example in FLARE would be the medikit.

2. Primary copy: we will allow writes to some primary copy (defined by the application) of the poap, so copies in other partitions may be outdated. For some poaps it may be allowed to read old data, for others this may not be allowed. Members that have access to the primary copy can always read and write. When partitions merge, the primary copy is simply copied over the outdated ones.
   An example in FLARE would be the flag.

3. Consistent: we want every copy to always have the same state. When partitions merge the state will still be the same, so no action is necessary.
   An example in FLARE would be the poap that keeps track of who won the game.

For some poaps the consistent type is not appropriate because we want to be able to update it even when there is a partition, and there appears to be an evident primary copy. For example, it makes sense to store the position of a player in a primary copy type poap.

- For all members within the same partition (or the whole group when there is no partition) that have received the same *set* of messages, the same consistency requirements as described in Section 4.1 apply.

- For members in different partitions, we want all copies to be at least as consistent as the consistency requirement associated with that poap.

5.3 Delivery of messages

Chapter 3Issue 4 presents four different delivery guarantees we could use. Which of these is the most logical choice depends in part on whether we send updates or events.

If we had chosen to send updates, the last 3 options would map nicely to the different poap types. Best effort delivery could be used for inconsistent type poaps, delivery only if the message can be delivered to a certain member would match primary copy type poaps, and for consistent type poaps we would use atomic delivery. The important difference between send events and updates

Poap	Consistency
Player position, health, score	Primary copy
Bot position, health	Primary copy
Medikit status	Inconsistent
Flag location	Primary copy
Winner	Consistent

Table 5.1: Consistency levels for poaps in FLARE

here is that when sending updates we already know which poaps to write to. We can then select the appropriate delivery guarantee for the poap we want to update.

But since we've chosen to send events, these now make less sense. When an event is delivered to a node the node processes it and may want to update poaps. To be able to decide if it is allowed to update these poaps, it needs to know which other members got the message. We will assume that when a message is delivered at some member, it is delivered to all the members in that member's current view that survive until the next view change. The view contains all the members in that member's partition. When a member sends a message, we give no guarantees on the delivery other than that if the sending node survives, it will receive its own message.

5.4 Poap types for the different objects

Now that we have established the different consistency levels, we need to associate a consistency level with each poap in FLARE. From the rules specified in Chapter 2, we get the following list of poaps: player position/health/score, bot position/health, medikit status, flag location, winner. The types we associated with these objects, deduced from the game rules, are listed in Table 5.1

5.5 Failure model

- Nodes can fail at any time (fail-stop). If they recover, they are considered new members for the group communication.
- Partitions can occur, making communication between the partitions impossible.
- Failures and partitions can occur simultaneously, (i.e., without enough time inbetween to detect them as separate events).
- A group can be split into more than 2 partitions *at once*.

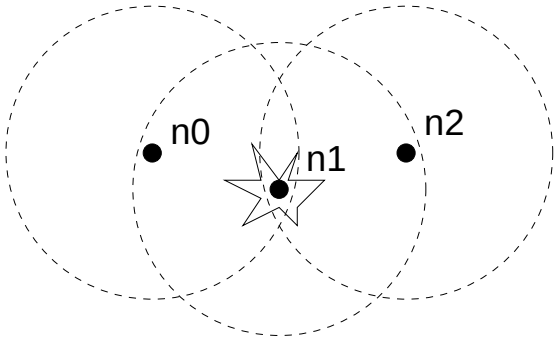

Figure 5.1: Partion caused by failure

The first two are obvious. When nodes fail, we want its members to be dropped from the group, like they would be in a normal leave operation. The third may make life rather hard for us, but cannot be excluded, since in an ad-hoc wireless network a failing node is a likely cause of partitions. In Figure 5.1, if node $n1$ fails, $n0$ and $n2$ will be partitioned. The last point will probably be less common.

When we admit both partitions and failures, we encounter two principal problems. First, when we lose contact with a node, it is impossible to tell if this is because of a partition or because it has failed. Second, when a group becomes partitioned, all nodes in a partition may subsequently fail. These two problems will be discussed in the next sections.

5.6 Detecting failures and partitions

In priciple, we need some way of detecting when a node fails and when the network is partitioned. What the exact requirements are depends on how we decide to handle partitions and failures. The ideal detection primitive would be able to detect which nodes fail and which are partitioned, even if both happen at the same time, and would immediately detect partitions and failures when they happen. Also it would be able to know which nodes are in the other partitions when a partition into more than two parts occurs.

In practice, this is impossible to realize. When a failure or partition occurs, the only thing that nodes detect is that communication with other nodes has become impossible. But if we want to build a policy that is able to handle both partitions and failures, we can't do without a function providing some subset of this, and it becomes important to determine how well our policy can handle the limitations of whichever detection function we use.

Two possible primitives that come close to providing such a service will be discussed next. When discussing the partition handling policies in Sections 5.8 and 5.9, we will examine the consequences of incorrect results from this

primitive, and how its characteristics may be relaxed in each case.

5.6.1 Partition anticipation

In [5] a distinction is made between logical and physical partitions. Based upon location information and wireless connection range, a 'safe distance' is determined. If a node moves beyond the safe distance, a logical partition occurs, meaning that the algorithm preemptively partitions the network. It can do this in a consistent way upon which all nodes agree because there is enough time between the moment a node moves out of safe distance and the moment physical network communication becomes impossible in order to properly inform all the nodes. If the node moves back to within safe distance again, the logical partitions are merged. This way partitions can be handled nicely.

If communication is suddenly lost, we still don't know if it's due to a partition or a failure. If we assume that our anticipation function works well, it would be reasonable to assume that the nodes with which contact is lost have failed. When a node fails and thereby causes a partition things get a bit more complicated. The policy could be made smart enough to conclude that there was only one link and that this link has failed. A group being split into more than two partitions at once will not be a problem.

This policy will probably perform quite nicely in most cases, but it is not hard to think of scenarios where it will produce incorrect results. For example, a partition caused by a failure will probably not be anticipated, a partition may occur because of some outside interference or because the communication path is blocked by a car or building.

Another partition anticipation policy is proposed in [3].

5.6.2 Counting failures

The probability of a number of nodes failing 'simultaneously' (without enough time to establish a new group view) depends on both the time it takes to detect a failure or partition and the number of nodes in the group. We could use this to distinguish between failures and partitions. If we take n to be the maximum number of nodes we expect to fail simultaneously, we can say that any number of nodes $\leq n$ with which communication is lost are presumed to have failed, and any number $> n$ are presumed to be in a partition.

It is important to note that in this policy, both partitions will decide for themselves what has happened. This causes some complications, and requires partitions to maintain some history of events until the partitions are all merged again.

For instance, in Figure 5.1, n will probably be set to 1. Now if $n1$ fails, $n0$ will presume there is a partition with $n1$ and $n2$ in it. And $n2$ will presume a partition with $n1$ and $n0$. When they merge again, they will have to realize that there is now a number of nodes $\leq n$ missing and conclude that those nodes ($n1$) must have failed. Therefore, using this policy it is impossible to detect failures and partitions occuring simultaneously until the partitions have merged again.

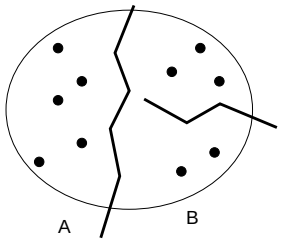

Figure 5.2: A partition splitting in two parts

Another complication is that in this example both $n0$ and $n2$ have no way of knowing that the other sees 2 nodes missing, and they must deal with the possibility that the other nodes have wrongly concluded that they have failed.

Partition in more than two partitions at once is not a problem, but again, won't be detected until partitions merge. Using this policy it is not possible to establish a consistent group view for all nodes when partitions and failures occur at the same time or a partition in more that two parts occurs.

Again it is not hard to think of scenarios where this policy doesn't work.

5.6.3 Conclusion

This is a very hard problem indeed. The first solution will probably work reasonably well, but requires location information (although this could possibly be replaced by signal strength information from the WaveLan cards). The second is easier to implement. Possibly the coverage estimation algorithm in [3] could be useful to determine if nodes have failed. A solution will probably include a combination of various ideas, possibly including the ones mentioned above.

This problem is very complex, and the policies suggested are just ideas. For us it is important to get a clear picture of the consequences of incorrect results from this primitive. Assuming that it should be possible to determine that communication is no longer possible with a node, there are two possible mistakes the partition and failure detection primitive can make.

- Assume a node is in a partition, when in reality it has failed. This will most likely be due to the failure of a node linking two partitions.
- Assume a node has failed, when in reality it is in a partition.

Also note that although it could come in helpful to have a consistent and agreed upon group view when a partition occurs, this cannot be achieved when a partition in an already partitioned group splits again. By definition the splitting partition has no way of informing the other. For instance, in Figure 5.2, B has no way of informing A that it is being partitioned, even if it is able to achieve agreement on this among its own nodes.

5.7 Dying partitions

When the network becomes partitioned, it is possible that all nodes in one of the partitions fail. When this happens the other partition will never be able to reestablish contact with them. This means that any group that has an explicit 'partitioned' state will not be able to recover if it waits for the other partition indefinitely, because the other partition no longer exists. Any state that was only stored in the failed partition will also be lost.

5.8 Partition handling policies

This section describes some possible policies for handling partitions and failures. We will compare several aspects of them, including the demands they put on the partition and failure detection and how they respond to malfunction of this primitive. Using these ideas we developed a policy for FLARE that will be described in Section 5.9.

5.8.1 Splitting groups policy

Policy When a partition occurs, the group splits into two (or more) independent, new groups. Both of these are unpartitioned and contain only the members that can be reached.

Group membership Membership of a group extends to all processes that have successfully joined the group or one of the parent groups, have not left the group or failed, and have always been in contact with each other.

Poap types Since a poap can only be accessible from one group, there needs to be a way to decide which poap ends up in which group. If all poaps go to the same group, the other will not be very useful, and we basically end up with the policy in Section 5.8.3.

Whichever algorithm is used, in this policy groups are never partitioned, so all types are equivalent. Of course the number of poaps that are in the group is less than before the partition occurred. Also nodes that don't have access to a poap can no longer know in which group it is located, just that it is in one of the subgroups of the one it was in originally.

Dying partitions After splitting the group, both groups will contain different poaps. When either one of the new groups dies, all its poaps will be lost. Recovery will be made more difficult by the fact that it is no longer known in which group a poap is located.

Failure and partition detection primitive When a failing node is wrongfully thought to be in a partition, the policy that determines which poaps end up in which partition will probably loose some poaps.

For instance if there was some virtual poap located close to $n1$ in Figure 5.1 and both nodes $n0$ and $n2$ assume the other two nodes are in another partition, neither will claim the virtual poap and it is lost. This means we need to be able to accurately detect failures and partitions happening simutaneously, which will be hard.

When a partitioned node is wrongfully thought to have failed, the other nodes will redistribute the ownership of its poaps among the remaining nodes. The partitioned node will be alone in a new group, and will retain its ownership. The result is that the poap has been duplicated!

A partition into more than two parts will not be more difficult than a partition into two parts.

Coding complexity The programmer must handle the fact that the poaps in the group are not statically defined and they may disappear suddenly. Also he doesn't know which poaps will be in a group.

Consequence of partition For the application it will appear as if a number of members and poaps disappear from the group.

Pro

- The fact that all poap types are equivalent will make programming easier.

Con

- The fact that we don't know in advance which groups there are and in which group poaps are, will make programming more difficult.
- Risk of loosing poaps when partitions die.
- Strict demands on detection of failure/partition to avoid losing or duplicating poaps.

5.8.2 Partitioned groups policy

Policy When a partition occurs, the group is split into two (or more) 'partitioned' subgroups. The group membership stays the same, and the partitions are aware of their partitioned state.

Group membership Membership of the group extends to all processes that have successfully joined the group, and have not left the group or failed.

Poap types By definition: Each partition may do as it pleases with inconsistent type poaps. Primary copy type poaps can be updated by members in one partition, and may be read by others, knowing that the data they read may be out of date. No partition may update consistent type poaps, and therefore all partitions can read them.

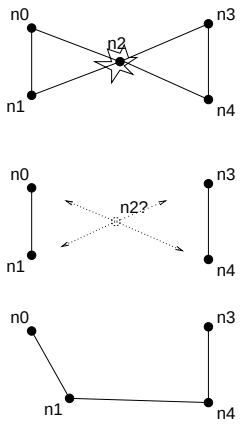

Figure 5.3: Discovery of n2's failure may be postponed until partition merge.

Dying partitions When a partition dies, the other partitions will not be able to recover from their partitioned state. And therefore the application will be severely crippled.

Failure and partition detection primitive When a failure is seen as a partition the group will never be able to recover from the partitioned state. Note that in some cases detecting it is a failure instead of a partition can be postponed until partitions meet again. In Figure 5.3 we see the centre node failing, causing two partitions containing $n0, n1$ and $n3, n4$. In this policy it doesn't matter if the two partitions assume $n2$ to be in the other partition, but when $n1$ moves close enough to $n4$ to reestablish communication between them, they must conclude that $n2$ has failed and therefore the group is no longer partitioned.

A partition in more than two groups will not be a problem as long as the detection primitive is able to detect if there is still a partition missing when the two others merge.

When a partition is seen as a failure, the group will just drop that member. The member that is in the partition may think it is still in the group and the group is in a partitioned state. The group will just use whatever failure handling function it has and continue the application. When they merge again, the group will realize the mistake and appropriate action must be taken.

Coding complexity The programmer must handle the partitioned state when some poaps will become read only.

Consequence of partition For all members some poaps will become read only.

Pro

- Will allow access to a lot of relevant poaps if the primary copies are wisely allocated.

Con

- Group gets locked in partitioned mode when failed nodes are wrongfully presumed to be in alive in another partition. This can be caused by mistaking a failure for a partition, or because a partition has died. Therefore it is very vulnerable to both these problems.

5.8.3 Main node group policy

Policy In each group there is a main node. When a partition occurs, the membership is reduced to the partition that contains the main node. Other members are removed from the group.

Group membership Membership of a group extends to all processes that have successfully joined the group, have not left the group or failed, and are in contact with the main node.

Poap types There is no partitioned state here, and therefore all poap types are equivalent. Each member has read/write access to all poaps.

Dying partitions This is only a problem if the partition of the main node dies, which will be a problem regardless of whether any partitions are present or not. When a partition without the main node dies, this partition would no longer be a member of the group, so there is no problem.

Failure and partition detection primitive Since this policy just drops every member that loses contact with the main node from the group, it is irrelevant whether they have failed, or have been partitioned from the main group. It also doesn't matter in how many partitions a group is split.

The critical requirement here is that we must be able to detect when the main node fails, whether that coincides with a partition or not. In that case a new main node must be elected.

Detecting this may be very hard for the reasons stated in Section 5.6 and it will probably be impossible to guarantee accurate detect in all cases. If we fail to detect the main node has failed, the group will cease to exist. If we think the main node has failed, when in reality it is partitioned, the groups will be duplicated. In that case reconciliation will be hard when the two main groups reestablish contact because both will have updated poaps.

Luckily, we have control over which node will be the main node, and we can include several factors in deciding this. Given the characteristics of our failure and partition detection primitive, we can pick a node for which it is very likely

we can properly detect failures. Another factor we would like to keep in mind is the fact that when a partition occurs we would like the main node to be in the largest group, so the most nodes can make progress. This should also help to prevent the group from dying.

Coding complexity All poaps will be available all the time, but the programmer must handle the fact that a member can be dropped from the group at any time.

Consequence of partition A number of members will be dropped from the group. Note that this does not mean they will have to start from scratch again when they rejoin. They may save state information and be able to rejoin the *game* when they can communicate again. The group will be able to play on as normal, and the partitioned or crashed nodes would be blocked.

Pro

- Most members will be able to make progress.
- Loose demands on failure and partition detection primitive.
- Little extra danger of a dying group as compared to a situation without partitions.
- Control over main node reduces risk of not detecting when it fails.
- Simple programming model.

Con

- Partitioned members will be dropped from the group entirely. Although it would be possible for the application to have a recovery policy so they won't have to start from the beginning, they certainly won't be able to make progress.

5.8.4 Majority partition policy

Policy When a partition occurs the members of each partition count the number of members in their partition. If this is more than half of the members in the group, they consider the others to have failed and drop them from the group. If not, they know the others will have dropped them. Unless there is a partition in two equal parts, but this can be easily avoided. See Section 5.9.3.

Group membership Membership of the group extends to all processes that have successfully joined the group, have not left the group or failed, and have never found themselves in a minority partition.

Poap types There is no partitioned state here, and therefore all poap types are equivalent. Each member has read/write access to all poaps.

Dying partitions This is only a problem when the majority partition dies. The chance of this will be quite small when we let the majority go on after a partition.

Failure and partition detection primitive Only has to detect how many members are in the current partition. It is irrelevant whether they have failed or are partitioned.

Coding complexity Same as main node policy.

Consequence of partition Same as main node policy.

Pro Same as main node policy, plus even more relaxed demands on the failure and partition detection.

Con Same as main node policy.

5.9 The partition handling policy for FLARE

From the previous it is clear that the splitting groups policy causes a lot of problems, and, for FLARE, doesn't have that many advantages. Also the fact that this version of FLARE will only use one group makes it useless. The partition handling policy we will use is a combination of the partitioned group and the majority group policies. It allows most members to make progress like the partitioned policy does, while avoiding the risk of mistaking a failure for a partition.

5.9.1 Partition anticipation

To detect partitions we use a partition anticipation function like the one described in 5.6. It will detect when a partition is likely to occur soon and preemptively partition the network. We call a preemtive partition a logical partition. The application will *not* be able to communicate with nodes in another partition when there is a logical partition. But when there is a logical, but no physical partition, the group communication layer *will* be able to communicate. When the partition anticipation function determines the risk of partition is gone, the group communication will merge the partitions. The preemtive partitioning is illustrated in Figure 5.4, merging works in a similar way.

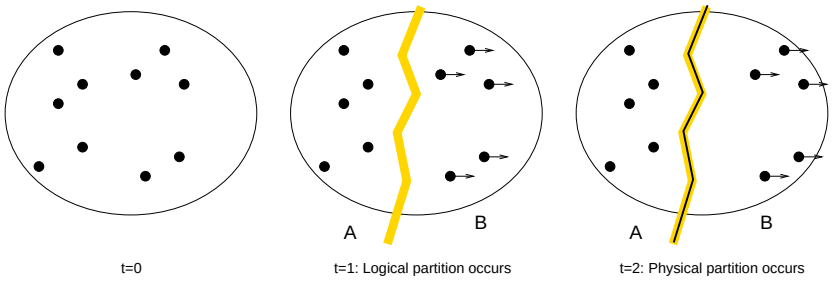

Figure 5.4: Partition anticipation preemtively partitions the network

Limitations Note that this partition anticipation function can never be sure that the partition will really occur. In Figure 5.4, partition B might turn around and in that case the preemptive partition was unnecessary. So we sacrifice some connectivity for time to have all nodes agree on the partition. The number of unnecessary partitions will be a measure of how well our partition anticipation function performs.

The partition anticipation function will never be able to anticipate all partitions, for example partitions due to failure, so there will be unanticipated partitions. We expect the majority of these partitions to be partitions of types that occur in fixed networks as well, instead of partitions caused by mobility. However the tuning of the partition anticipation function may cause more unanticipated partitions. We can probably (depending on how it is implemented) make the anticipation more optimistic or pessimistic, resulting in either more unanticipated partitions, or more unnecessary logical partitions. How unanticipated partitions are handled is described in Section 5.9.3.

Also note that by definition, this function cannot mistake a failure for a partition, or a partition for a failure.

Timeout

Because all nodes in an anticipated partition may leave the group or fail, which can cause the members of the group to be stuck in the partitioned state forever, we set a timeout on the anticipated partitions. When the anticipated partition is formed, there is some negotiation going on, and during this phase, we will assign a 'main partition'. The main partition is only formed when a partition occurs in the previous main partition, not when a non-main partition splits. The non-main partition is given a timeout time. If it doesn't merge with the main partition before its timeout time is over, all members in it are considered to have failed. We set the timeout in the non-main partition sufficiently shorter than in the main partition so that we know the members will declare themselves failed before the main partition does.

When the main partition is split *again*, the nodes that split off will have the

same timeout period, and will therefore timeout at a later point in time. The main partition doesn't timeout any members unless it can timeout *all* missing members, in which case it drops all missing members from the group and the group returns to the unpartitioned state.

When non-main partitions merge, one may have a higher timeout time than the other. Because the main partition doesn't timeout any members before it can timeout all missing members, each member in the partition can safely take on the highest timeout time. Each partition keeps track of the timeout times it knows for all members. Because the timeout time for a members can only increase, when partitions merge, the highest value from both tables is taken. This ensures that if a partition passed on its higher timeout time to another partition and at a later time merges with the main partition, the main partition will know about this and won't timeout the members that initially had a lower timeout time (they may be the only nodes that are still missing).

There will only be one main partition and the only way for a non-main partition to avoid the timeout is to reestablish contact with the main partition. If partitions never meet again, the membership will eventually be reduced to the main partition. The main partition should therefore be chosen in such a way as to ensure maximum probability of survival for the group. This will probably mean selecting the partition containing the most nodes, although other factors like battery life and connectivity could be included in the decision as well. The timeout period is a parameter of the policy and can be set to anything from 0 to ∞ according to application requirements.

5.9.2 Creating and joining a group with partitions

In the presence of partitions, we allow more than one designated join point, and they can exist even when there are still members in other partitions. Basically the same thing that happened for a group in the case without partitions, now applies for a single partition. Instead of representing the entire group, a designated join point now represents a partition (which may be the whole group).

Joining the group The process for joining is basically the same. A joining process has to be in contact long enough for the joining process, the join point, and any members with which it is in contact to install a new group view. As before, when a process joins at a designated join point it becomes a join point and the designated join point disappears.

When a process joins a non-main partition, it needs a timeout time. We take this to be the latest timeout time, known to the partition where it joins, a shorter time is pointless because we know the main partition cannot decide the group is unpartitioned earlier, a later time is risky because the main partition may decide the group is unpartitioned, while the new node hasn't timed out yet. Each node keeps track of the latest timeout time it has seen or heard about, and tells other nodes about this when partitions merge.

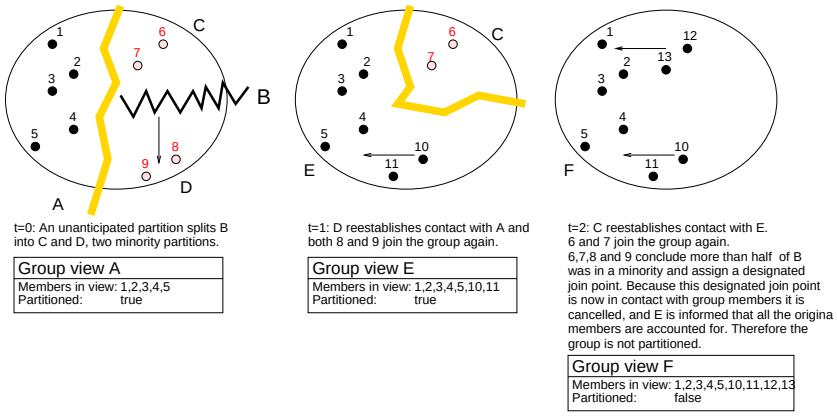

Figure 5.5: Minority partitions concluding there is no majority

Leaving the group Essentially leaving the group is the same, except now when the last member of a *partition* leaves, its node becomes a designated join point, which is why we can have more than one join point. This designated join point also has a timeout time, and this time is the latest timeout it has seen or heard about. Again, earlier is pointless and later is risky.

Merging Using this scheme there may be more than one designated join point. However they can only be created by seperate partitions. At some stage two designated join points may reestablish contact, in that case one of the two stops being a join point. When a node that is a designated join point reestablished contact with a partition of the group, it stops being a join point.

In either case, the same operations are performed as when normal partitions merge. So if a group is split in A and B by an anticipated partition, all members in B leave, and A reestablishes contact with B's designated join point, A will know the group is now complete.

5.9.3 Unanticipated partitions

Although we hope the partition anticipation will take care of most partitions, there will still be unanticipated partitions and failures. In both of these cases the only thing we notice is that contact is lost with the nodes. When we unexpectedly lose contact with one or more nodes, we call this an unanticipated partition. In this case, neither partition has a way of knowing if the other is still alive.

When an unanticipated partition occurs, the majority partition, based on the number of nodes, assumes the other nodes have failed, and they are dropped from the group. Since all partitions can determine how many nodes they can still communicate with, they can independently decide if they are the majority partition or not. If they are the majority, they install a new group view among

themselves, and inform the application that the members running on the missing nodes are dropped from the group. If they are not they consider themselves dropped from the group and inform the application of this. The surviving nodes in a minority partition are no longer a member of the group.

The fact that the member was dropped from the group doesn't mean the player has to be dropped from the application as well. The game may attempt to reestablish contact with the rest of the group and use some rejoin protocol to reintegrate the dropped player.

Minority partitions A very undesirable scenario would be that there are no majority partitions. When a node is dropped because of a minority partition, the application may either try to join the group again or give up. If it tries to rejoin it can be succesfull either by establishing contact with some join point for the group, or by establishing contact with more than half of the members that were dropped from the same partition. In that case, they can draw the conclusion that there was no majority partition. They will then create a designated join point, and the application can join at that point. This way the recovery would be transparent for the application.

This is illustrated in Figure 5.5. Note that members 6-9 get new IDs because they were dropped from the group when the unanticipated partition occurred. When they rejoin the group, they are treated as new members.

Breaking even partitions Since FLARE will be played by a relatively small number of players, it is more likely that a partition will split into two parts of equal size. If the partition was unanticipated, this would cause two minority partitions. Now, although the nodes would recover from it when they reestablish contact, we would prefer if one was able to make progress.

We achieve this by giving one node double weight whenever the total number of members in a partition is even. Therefore, whenever it is split in two, there will always be a majority.

It is very easy to see that this can only have a positive effect. When there is an even number of $2n$ nodes, a partition must have $n+1$ nodes to be a majority. When there is an uneven number of $2n+1$ nodes, a partition must still have $n+1$ nodes to be a majority. So adding the extra weight to one node doesn't make the requirements for deciding a partition is a majority any stricter. However, since there is now 'more weight' in the group, the probability of a partition being a majority is larger.

Who is the double weighted node will be decided whenever a group view is installed, the double weighted node should be picked on the likelihood that it will not fail.

5.9.4 Group view

Whenever a partition is anticipated a new group view will be installed. This view will be different for members in different partitions. The group view consists

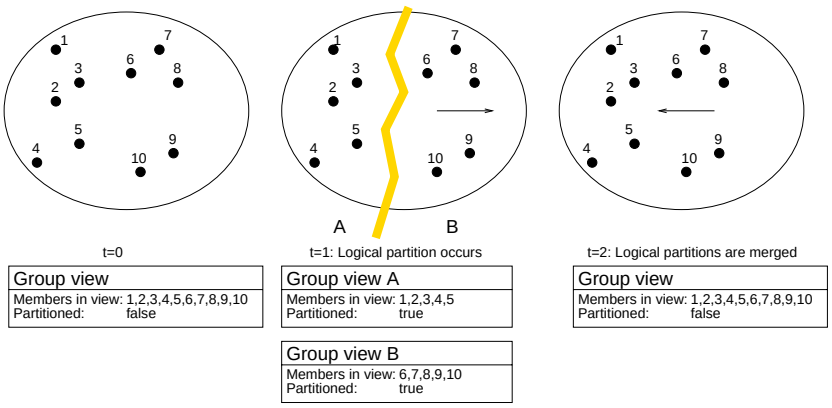

Figure 5.6: Installing a new group view for an anticipated partition

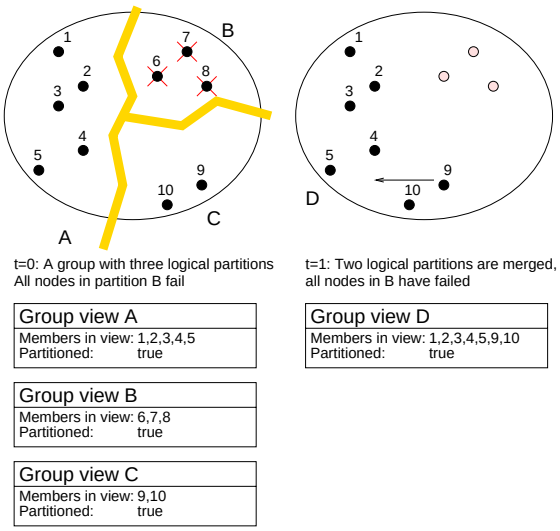

Figure 5.7: Partitioned flag remains set, although there are no more partitions.

of a list of members that can be communicated with, and a flag that indicates whether the network is partitioned or not. This is illustrated in Figure 5.6. The flag only indicates that the group *may* be partitioned. When it is $false$, we know for sure that there is no partition. When it is *true*, we do *not* know for sure that there is a partition, since all the members in the other parttition may have failed or left. This is illustrated in Figure 5.7.

Requirements A process should not receive messages when it is not a members of the group. When a process is a member and it sends a message, it should eventually receive its own message. When it receives a view, it must be a member in that view and it knows that it is a member of the group. When a member is dropped from the group the group communication must inform it about this. When a member receives a view, it can also be sure that all members in that view are really members. All members in a partition get the same view. So if a gets a view containing b, then b gets the same view as a, and that view also contains a. When a member receives a message, it can be sure that all members in its view will receive this message.

The exact requirement are described in greater detail in [8].

5.9.5 Group membership

Processes can only become a member of the group through a succesful join operation. They are a member of the group until they leave, or are declared to have failed by *any* member because of unanticipated partition or timeout.

For an unanticipated partition, some partition may decide that it is the majority before other partitions realize they are the minority. So the processes in the minority partition are no longer members, but they may not be aware of this. We require that they don't receive any messages sent after the partitions split, and any messages they send are never delivered. Eventually they will realize they are no longer members.

For a timeout, the shorter timeout on the non-main partition nodes ensure a member will decide it has timed out before the main partition does. The leave operation is simply installing a new view, so this will happen simultaneously for all members in the partition. So with our two policies for handling anticipated partitions and unanticipated loss of contact, each process can be sure whether it is a member or not.

Whenever the group communication layer of a member decides another member has left the group, either because of an unanticipated partition, a timeout of an anticipated partition, a leave operation, or because it has learned this from another member, it will inform the application about this, and present the application with a new group view (which may be the same as the previous one).

5.9.6 Problems

Earlier on, we identified 2 possible problems:

Dying partitions This could cause a group to remain locked in the partitioned state. For anticipated partitions, our timeout scheme ensures that, when the timeout is set to $< \infty$, the main partition will eventually recover. When all members in the main partition fail, the other nodes will eventually timeout, and the group will no longer exist, because all join points are gone. So, in our scheme, the group will *never* get stuck in a partitioned state, although in an unfortunate case, it may cease to exist.

For unanticipated partitions, only the majority will continue, so it can never be locked in a partitioned state. When the group splits without a majority, the group will no longer exist, but it may be revived when enough minority members meet.

Detecting faillure and partitions Our inability to distinguish partitions and failures when communication is lost can cause all sorts of problems. We solved this by making our only notion of partition the anticipated partition, which can be properly detected and agreed by all nodes. By considering any loss of communication caused by a failure, we may sometimes assume a node has failed when it is really partitioned. But in our scheme this node would always be aware of the fact that it is no longer a member of the group and this way we avoid a lot of problems.

5.10 Algorithm that implements the partitioned state indicator

This section will describe an algorithm to decide whether the group may be partitioned or not using the policy described in the previous sections. It will split and merge anticipated partitions, detect when the group is unpartitioned, handle timeouts and unanticipated partitions, and detect when an unanticipated partition splits into just minority partitions. It just uses a list of members it can communicate with and the results of the partition anticipator as input. Whether this list of members is installed as a view or not is not important for the algorithm.

First we will develop a policy that just deals with anticipated partitions. Then we will expand this with timeouts and finally with unanticipated partitions.

Again in this section we will mostly speak about 'a partition', even when the group is in an unpartitioned state because this doesn't matter for the algorithm. Also we speak about 'a partition' when this can be empty and just represented by its designated join point.

5.10.1 Anticipated partitions

The algorithm is based on a *partition information* data structure. This is a partition wide data structure that stores information on all known past and

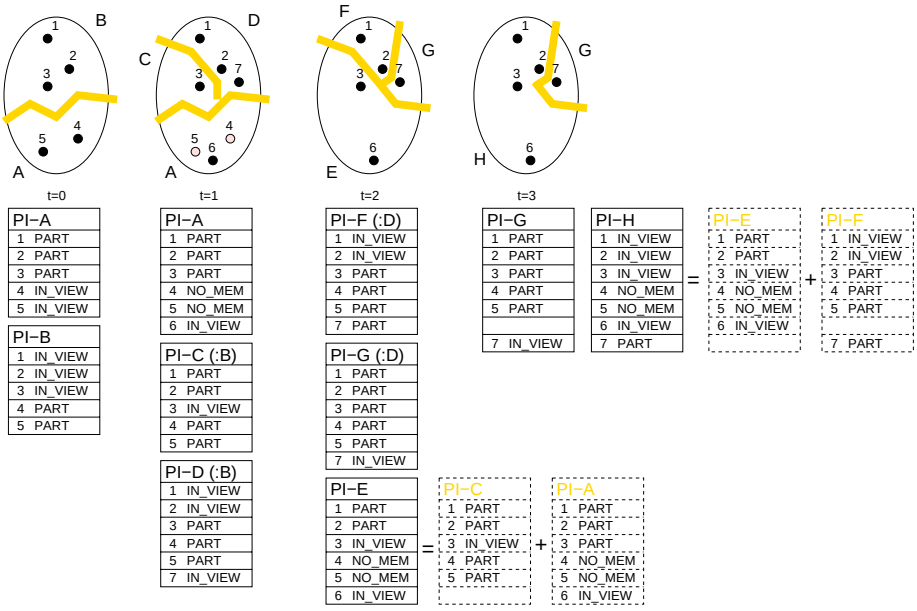

Figure 5.8: Use of the partition information datastructure

present members. It is replicated on all nodes in the partition. The idea behind the algorithm is that when an anticipated partition occurs, a number of nodes will be missing. We gather information on what happened to these processes and at some stage we will hopefully be able to determine the group is in an unpartitioned state again.

The partition information is a map of (past) member IDs to their state:

$$partition\ information:\ member\_id \rightarrow member\_state$$

where member state is IN_VIEW, PARTITIONED, or NO_MEMBER.

When a partition occurs, a partition receives a new list of members it can communicate with. It now updates its partition information in the following way: for each member in the map that has IN_VIEW status, but is not in the new list of members in the partition, the status is changed to PARTITIONED. When two partitions merge, their maps are merged, and the result is used for the merged partition. For each member that only appears in one of the maps, the entry is copied. For each member that appears in both of the maps the highest precedence status is used, with the order from high to low being IN_VIEW, NO_MEMBER, PARTITIONED.

Note that since we map members to a status and members don't rejoin the group, the IN_VIEW and NO_MEMBER can never occur simultaneously. If we map nodes, then this can happen and the given order is correct. After each merge operation, we check if there are any members left with PARTITIONED

status. If not, we draw the conclusion the group is unpartitioned and remove all entries with NO_MEMBER status. If there are PARTITIONED status entries, the group *may* still be partitioned.

Joining When a process succesfully joins the group, he is given a new member ID. This is put in the partition information list with IN_VIEW state.

Leaving When a member leaves the group for whatever reason, there are two options. If there are members with PARTITIONED status, the leaving member's status is changed to NO_MEMBER. If there are no members with PARTITIONED status, his entry is removed from the map.

This is illustrated in Figure 5.8. At $t = 0$, we see a group split into two partitions. At $t = 1$, partition B splits again and D admits a new member (7) *after* the split, and in A the two nodes leave, and a new member (6) joins. We see that A's list now has an entry for member 6, and D's list has an entry for 7. At $t = 2$, D splits and A and C merge into E. Here we see our merge algorithm working. At $t = 3$ E and F merge and we see that H has learned that there must be another member out there, even though this member wasn't in the original group when it split. From G and H is it clear that when they merge, there will be no more partitioned state members.

5.10.2 Timeouts

Adding timeouts requires relatively little modification. We expand the partition information structure with a timeout field:
$partition\,information : member\_id \rightarrow member\_state, timeout$

The main partition stores timeout values for all members that have status PARTITIONED, the other partitions store timeouts for all members of which they know the timeout time.

When the main partition splits, a new main partition is agreed on. The main partition enters the timeout specified by the policy parameter for all new PARTITIONED members. The new partition enters a timeout time sufficiently lower to ensure it will timeout first for all IN_VIEW members. The timeout times of all other nodes are just copied. When a non-main partition splits the timeout values don't change.

When two non-main partitions merge each node gets the same timeout time, and the highest timeout time of the two partitions is used for this. When a partition merges with the main partition, its timeouts are cleared.

Joining When a process succesfully joins the group in a main partition, it doesn't get a timeout. When it joins in a non-main partition, it gets a timeout equal to the timeout of the other IN_VIEW members in the partition.

Leaving When all members leave in a non-main partition the designated join point also gets a timeout. The value is the same as that of the member that left.

Timeout When a member discovers that it has timed out, it will leave the group. When a node in the main partition discovers that the timeout time of *all* members with PARTITIONED status have expired, it drops them all from the group and sends a message to the other nodes in the main partition saying this member has been dropped from the group. If only a subset of the PARTITIONED status members have an expired timeout time, *no* members are timed out because these may have copied a higher timeout time from one of the other nodes.

Again, when there are no members with PARTITIONED status in the partition information, the group is unpartitioned.

5.10.3 Unanticipated partitions

This is a rather simple extention of the previous algorithm. When an unanticipated partition occurs the nodes know with which nodes they can still communicate. If this is more than half the nodes that had IN_VIEW status in the previous view, these nodes know they are the majority and will drop the other members from the group. When it is less than half, they know they are dropped from the group, and no longer consider themselves members.

Now the only thing we should make sure of is that if there are only minority partitions, and they try to join the group again, we should create a designated join point if they may reestablish contact with more than half of the members of the previous partition. This joint point should have the same partition information map and timeout that would have been formed if all members left.

To achieve this we let each node, when it is in a minority partition, remember the last group view. When a node reestablishes contact with any of the nodes that were members with IN_VIEW status in that view, it asks them if they have saved the same group view. If they haven't, either

- at least one of them was a member of the majority partition,
- or there was no majority partition but at least one of them was already involved in recovery from that.

In either case, the node deletes the group view. If on the other hand these nodes have also stored this group view, they may form a majority together. If they don't, nothing happens. If they do, they create a designated join point, with a partition information map, replacing all IN_VIEWs with NO_MEMBERs, and with a timeout equal to the timeout in the previous group view. After this, there may be a merge operation of this designated join point with another partition if some of the nodes already reestablished contact with. If not, processes can now join again at the designated join point.

The nodes give up their attempt to recover when their timeout time as it was in the previous group view expires.

5.11 Required API

```
join(name);
leave();
name[] list_groups();
// Specify the partition timeout
// when creating the group.
create_group(name,timeout);
fbcast(msg);
abcast(msg);
// Delivers a message sent by fbcast or
// abcast. No ordering is imposed on messages
// using different send primitives.
deliver(*msg);
// Callback. Will be called whenever nodes join
// leave, fail, or a partition occurs. The group view
// contains a list of members with their status, either
// IN_VIEW, PARTITIONED, or NO_MEMBER. When a node has
// status PARTITIONED, this means we cannot communicate
// with it, and therefore do not know if it is still a
// member of the group.
// When a member has left the group or has failed, the new
// group view will list it as 'NO_MEMBER', and it will not
// appear in subsequent views.
// Whenever there's a member with PARTITIONED status, the
// boolean 'partitioned' will be true.
view_change(new_view, partitioned);
```

Chapter 6

FLARE Communication Layer Timeliness

This chapter discusses what timeliness guarantees are requiremented in order to ensure that the order of events in the game matches what is observed in the real world. From the system's point of view, events in the real world are observed through a 'hidden channel', which is a channel through which information about events is sent outside of the system. In the case of FLARE, this will probably be vision. In Figure 6.1, we see two players shooting at each other. When a player physically shoots, this information is registered by his computer (a), and sent over the network (b). When it is received back (c), the outcome of the shooting is determined and rendered on the screen, and the player sees this on the screen (d). This is the normal communication channel. The players can also see other players through the hidden channel. For example, they may see A shooting at C first, followed by B shooting at C. They will visually see other players pull their triggers, in some sequence of events. From this they form an expectation of what should happen in the game. If something different happens in the game, for example, B gets the point for killing C, the game will seem to be malfunctioning.

Regardless of whether two players are shooting at each other, are shooting at the same player, or are observed by a third player, we want the events that are observed in the real world to match those in the game. The important thing to realize here is that we are only interested in how the events are observed, not in the sequence in which the actually took place. Given infinitely precise observation, an observer could always tell the sequence in which two events occur, but since such observations cannot be made, we should examine how the accuracy of the observation is related to the timeliness requirement.

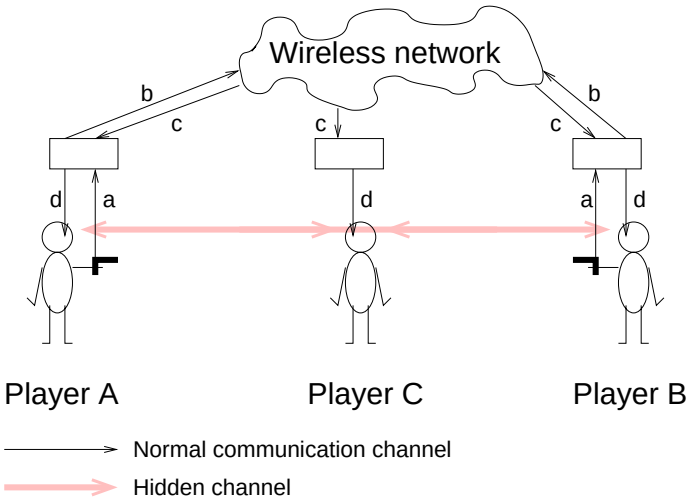

Figure 6.1: A hidden channel

6.1 Definitions

The three main measures that will be of importance are:

T^{hc}_{min}, T^{hc}_{max} The minimum and maximum time before an event can be perceived through the hidden channel.

T^{net}_{min}, T^{net}_{max} The minimum and maximum time before information about an event can be delivered through the network.

μ_{obs} The minimum time between two events being observed on the hidden channel such that an observer will distinguish the events as having happened sequentially. If the time is smaller than μ_{obs} the events will be perceived as being concurrent.

μ_{resp} The minimum time betwee a player perceiving an event (through the hidden channel) and responding to that event.

We derive two uncertainty measures from this

$\Theta^{hc} = T^{hc}_{max} - T^{hc}_{min}$ The uncertainty of the delivery time through the hidden channel.

$\Theta^{net} = T^{net}_{max} - T^{net}_{min}$ The uncertainty of the delivery time through the network.

And we define symbols for important points in time

t_n The physical time at which event n occurred.

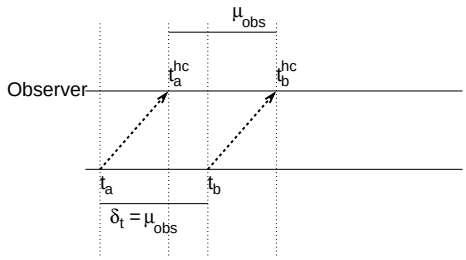

Figure 6.2: $t_b - t_a = t_b^{hc} - t_a^{hc}$ when $\Theta^{hc} = 0$

t_n^{hc} The physical time at which information about event n was perceived through the hidden channel.

t_n^{net} The physical time at which information about event n was delivered through the network.

Finaly, let's define what we consider the 'correct' delivery order:

- Events which are perceived in a certain sequence by any observer, must be delivered in the same sequence on the network.
- Events which are perceived to be concurrent can be delivered in any order.

6.2 Temporal ordering

For FLARE, the hidden channel will most likely be vision, so the speed of light determines the propagation time and we can assume $\Theta^{hc} = 0$. We will first examine this case, and then look at the more general case when $\Theta^{hc} > 0$.

6.2.1 Case 1: $\Theta^{hc} = 0$

Our goal here is to determine the criteria that must be met in order to enssure that two events perceived in sequence through the hidden channel are processed in the same sequence on the normal network. First let's introduce a definition from [10]:

δ_t-precedence order($\overset{\delta_t}{\rightarrow}$): *An event a is said to δ_t-precede an event b, $a \overset{\delta_t}{\rightarrow} b$, if $t_b - t_a > \delta_t$.*

Assume two events a and b with $t_a^{hc} < t_b^{hc}$. We only care about the case when $t_b^{hc} - t_a^{hc} > \mu_{obs}$ (i.e. when the events are perceived in sequence), which implies $t_b - t_a > \mu_{obs}$ when $\Theta^{hc} = 0$. So the message delivery should respect the $a \overset{\delta_t}{\rightarrow} b$ ordering, with $\delta_t = \mu_{obs}$. This is shown in Figure 6.2.

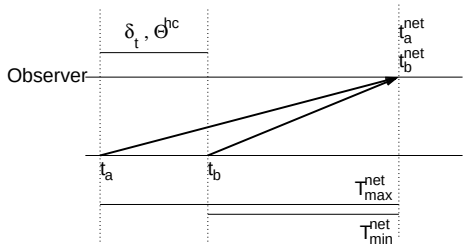

Figure 6.3: Worst case for delivery of messages

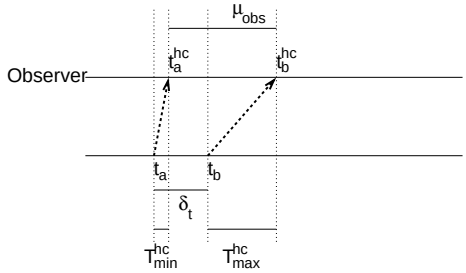

Figure 6.4: Worst case for temporal ordering with $\Theta^{hc} > 0$

Criteria for δ_t-precedence What are the criteria for message delivery times in order to guarantee that the delivery will respect δ_t-precedence? Obviously the more time between events, the easier it is to order them properly, so from here on we will examine the worst case. This is when the time between events is δ_t, $t_b - t_a = \delta_t$, the first messages takes T^{net}_{max} to deliver and the second T^{net}_{min}. This is illustrated in Figure 6.3. We want to deliver a before b, so $t^{net}_a < t^{net}_b$.

$$
\begin{array}{rrcl}
 & t^{net}_a & = & t_a + T^{net}_{max} \\
 & t^{net}_b & = & t_b + T^{net}_{min} \\
 & t_b & = & t_a + \delta_t \\
\Rightarrow & t^{net}_b & = & t_a + \delta_t + T^{net}_{min} \\
\text{we want} & t^{net}_a & < & t^{net}_b \\
\Rightarrow & t_a + T^{net}_{max} & < & t_a + \delta_t + T^{net}_{min} \\
\Rightarrow & T^{net}_{max} - T^{net}_{min} & < & \delta_t \\
\Rightarrow & \Theta^{net} & < & \delta_t
\end{array}
$$

So if the uncertainty in the message delivery time is smaller than δ_t, the delivery will respect δ_t-precedence. Since in this case $\delta_t = \mu_{obs}$, the requirement becomes $\Theta^{net} < \mu_{obs}$. When the uncertainty in the message delivery time is smaller than the observer accuracy, our ordering requirement is met.

6.2.2 Case 2: $\Theta^{hc} > 0$

Now let's examine what happens when the propagation delay on the hidden channel is not constant. Again the worst case is when the two events occur within the minimal amount of time such that an observer *may* be able to tell the order. From Figure 6.4 it is clear that the time between events so that an observer may see the order is now smaller. It is smallest when the first event is propagated in T^{hc}_{min} time through the hidden channel and the second event with T^{hc}_{max} time, since this maximizes the period between delivery times on the hidden channel.

In that case

$$\begin{array}{rrcl} & t^{hc}_b & = & t^{hc}_a + \mu_{obs} \\ & t^{hc}_b & = & t_a + T^{hc}_{min} \\ & t^{hc}_b & = & t_b + T^{hc}_{max} \\ \Rightarrow & t_b + T^{hc}_{max} & = & t_a + T^{hc}_{min} + \mu_{obs} \\ \Rightarrow & t_b & = & t_a + \mu_{obs} - \Theta^{hc} \end{array}$$

We only care about the case when $t^{hc}_b - t^{hc}_a > \mu_{obs}$, which implies $t_b - t_a > \mu_{obs} - \Theta^{hc}$. So the messages should respect the $a \xrightarrow{\delta_t} b$ ordering, with $\delta_t = \mu_{obs} - \Theta^{hc}$. From our previous result we know that this is the case when $\Theta^{net} < \mu_{obs} - \Theta^{hc}$.

So the result is that the limit on the uncertainty of delivery time on the network has been brought down by the uncertainty on the hidden channel.

6.2.3 Case 3: $\Theta^{hc} > \mu_{obs}$

The previous result implies that if $\Theta^{hc} > \mu_{obs}$, proper ordering cannot be guaranteed. This makes sense, because it implies δ_t-precedence should be respected with $\delta_t = \mu_{obs} - \Theta^{hc} < 0$. δ_t-precedence doesn't make sence for a negative δ_t: two events a and b, could now have $t_b - t_a > \delta_t$ and $t_a - t_b > \delta_t$, implying $a \xrightarrow{\delta_t} b$ and $b \xrightarrow{\delta_t} a$!

It is also intuitively correct because the uncertainty of delivery times on the hidden channel is now greater than the observers accuracy. This means that two events may be seen in reverse order, while this couldn't happen in the case when $\Theta^{hc} \leq \mu_{obs}$. Clearly, if the events may be delivered out of order on the hidden channel, and the system has no knowledge of the channel, it can never guarantee the messages on the network are delivered in the correct order.

General effect of Θ^{hc} on the perceived order: Events may be delivered in the wrong order on the hidden channel if they are less that Θ^{hc} apart ($t_a > t_b - \Theta^{hc}$, Figure 6.5.a). Whether they are *perceived* in the wrong order on the hidden channel depends on the arrival time. The maximum difference in arrival times of two events a and b arriving in the wrong order, with $t_b = t_a + \delta$, is at most $\Theta^{hc} - \delta$. This happens when the first event, a, is delivered with maximum delay T^{hc}_{max}, and the second event with minumum delay T^{hc}_{min} (Figure 6.5.b).

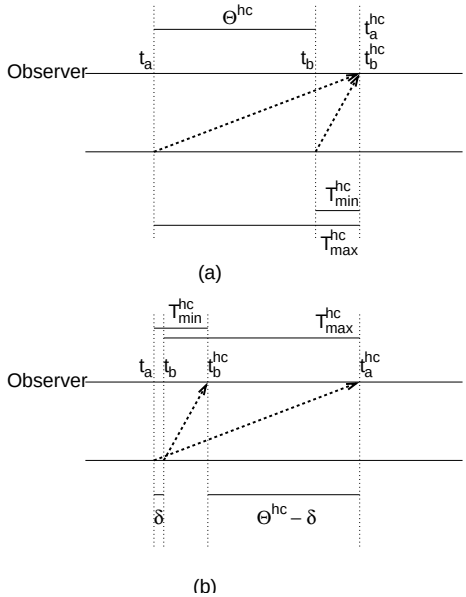

Figure 6.5: Events happening more than Θ^{hc} time apart are delivered in order, events perceived less than Θ^{hc} time apart may be perceived in the wrong order.

So it is easy to see that if $\Theta^{hc} < \mu_{obs}$, then the maximum time between events arriving out of order on the hidden channel $\Theta^{hc} - \delta$ is also smaller than μ_{obs}, and the events are perceived to be concurrent. This means that with enough certainty on the delivery delay on the network, we can make sure that the events are processed in the correct order.

For any two messages perceived with Δ time inbetween, we can say that they occurred with $\Delta - \Theta^{hc}$ to $\Delta + \Theta^{hc}$ time in between.

6.3 Causality

To ensure causality, we can take the same approach again. Two events a and b, with $t_a < t_b$, can be causaly related when a was perceived by the player sending b at least μ_r time before t_b. From Figure 6.6 it is clear that the minimum time between a and b for a causal relationship is $T^{hc}_{min} + \mu_{resp}$. So if we apply our result for δ_t-precedence with $\delta_t = T^{hc}_{min} + \mu_{resp}$, we see that causality is respected when $\Theta^{net} < T^{hc}_{min} + \mu_{resp}$.

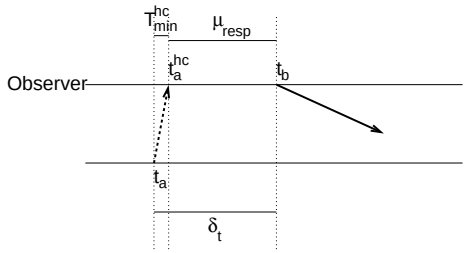

Figure 6.6: Minimum time between events needed for a causal relation.

Chapter 7

FLARE Communication Layer
Use of Location Information

In FLARE 1, location information will be used for two purposes. First, location information will be used to determine which node hosts objects like medikits and bots. As described in Section 4.7, some node needs to be responsible for generating the messages for these objects. Which node is hosting a node only becomes important when a partition occurs. Since it makes sense to assign the primary copy of a poap to the node that is hosting the associated object, and we expect users to interact more with objects that are close by, it makes sense to host an object on a node that is close by that object. We will host the object on the node that it is closest to, and periodically reevaluate the locations to move the object if neccesary.

Second, we need a way of assigning the primary copy for the other objects when partitions occur. We and we will do this using location information. We will assign the primary copy of a poap to the partition containing the node closest to it. In order to reach agreement on this, we will flush all movement messages that were in the network before deciding this. This way wel know all nodes have the same view.

Of course the primary copy of the poaps representing a player are in the partition that player is in.

Bibliography

[1] Id software source code. http://www.idsoftware.com/archives/sourcearc.html.

[2] Ronald T. Azuma. A survey of augmented reality. *Presence: Teleoperators and Virtual Environments*, 6(4):355–385, August 1997.

[3] M.-O. Killijian; R. Cunningham; R. Meier; L. Mazare; V. Cahill. Towards group communication for mobile participants. *Proceedings of the 1st ACM Workshop on Principles of Mobile Computing (POMC) August 29-30, 2001, Newport, Rhode Island, USA*.

[4] Idit Keidar; Danny Dolev. *Dependable Network Computing, D. Avresky Editor*, chapter Chapter 3: Totally Ordered Broadcast in the Face of Network Partitions. Exploiting Group Communication for Replication in Partitionable Networks. Academic Publications.

[5] G.-C. Roman; Q. Huang; A. Hazemi. Consistent group membership in ad hoc networks. *ACM 23rd international conference on Software Engineering*, pages 381–388, May 2001.

[6] Ozalp Babaoglu; Renzo Davoli; Alberto Montresor. Group communication in partitionable systems: Specification and algorithms. *IEEE Transactions on Software Engineering*, 27(4):308–336, April 2001.

[7] R. Prakash and R. Baldoni. Architecture for group communication in mobile systems. *Proceedings 17th IEEE Symp. on Reliable Distributed Systems*, pages 235–242, 1998.

[8] N. Reijers. A formal specification of a group communication service for flare. *Department of Computer Science, Trinity College Dublin Technical Report TCD-CS-2001-46*, 2001.

[9] Alan Fekete; Nancy Lynch; Alex Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, May 2001.

[10] Paulo Verissimo. Ordering and timeliness requirements of dependable real-time programs. *Journal of Real-Time Systems*, 7(2):105–128, September 1994.

[11] Paulo Verissimo. Causal delivery protocols in real-time systems: A generic model. *Journal of Real-Time Systems*, 10(2):45–73, May 1996.

[12] Gregory V. Chockler; Idit Keidar; Roman Vitenberg. Group communication specifications: A comprehensive study. *To appear in ACM Computing Surveys*, December 2001.