Using Group Communication to Support Mobile Augmented Reality Applications

Niels Reijers, Raymond Cunningham, René Meier, Barbara Hughes, Gregor Gaertner, Vinny Cahill
Distributed Systems Group
Trinity College Dublin
Ireland
{Niels.Reijers, Rene.Meier, Raymond.Cunningham,
Barbara.Hughes, Gregor.Gaertner, Vinny.Cahill}@cs.tcd.ie

Abstract

Augmented reality and group communication in wireless ad-hoc networks form relatively new fields of research. When using group communication ordering and timeliness requirements are important. Moreover, when using wireless ad-hoc networks, the possibility of network partition is a serious consideration. In this paper we explore these three issues in the context of using group communication to support mobile augmented reality applications. We describe a policy that enables us to handle partitions and failures, while allowing the members in a partition to make progress, although limited by the application's consistency requirements. We introduce an approach to determining the message ordering requirements needed to maintain a desired level of consistency and timeliness requirements that should be met in order to have the application state correspond to the sequence of events perceived in the real world.

1 Introduction

"Shoot 'em up" style games like Doom and Quake have become popular in recent years. Paintball, a real world outdoor "Shoot 'em up" game using paint-filled bullets, and Quasar, an indoor game, which uses laser beams and sensors on the players' suits, have also become very popular.
This paper presents a summary of [8] in which we present Flare, a framework for building augmented reality applications. The first application that we will build using Flare will be a Doom-like game called Quazoom. It combines the Doom experience with Quasar play to make an augmented reality game where players move around in the real world, while interacting with virtual and real players. Players will see a Doom-like game on their screen, providing a virtual representation of the real world. Their location in the game will correspond to their real world position, determined using Differential Global Positioning System (DGPS) and possibly other sensors. Players can shoot other players and pick up things like ammunition or medikits as in a normal Doom game.
Quazoom will be run on wearable computers. A wireless ad hoc network will be used for communication. This means, in particular, that there is a higher probability of the network becoming partitioned than in traditional networks. Players may move in and out of network range. When this happens we don't want them to be dropped from the game, but to allow them to make limited progress in the game. We describe our policy for handling these partitions in Section 4.
Players will communicate using a group communication service providing reliable and timely message delivery and offering several ordering primitives. They broadcast information like their new position, or the fact that they have fired their guns. In Section 3 we describe our approach to determine the requirements on message ordering to maintain consistency in the design chosen for Flare. Finally, Section 5 describes the timeliness requirements that must be met to have the game state consistent with the events perceived in the real world.

1.1 Quazoom game rules

Since Quazoom will be a Doom-like game, the functionality we will provide will be a subset of the functionality of this kind of game. The rules here were chosen to explore different sorts of consistency guarantees, not to make a fun game. There are 3 different game objects: players, medikits, and flags. Players can move and shoot. Players move around in the game by moving around in the real world.
When a player shoots, the first player in their line of fire gets hit, and loses 50% of their health. When a player's health reaches 0, he is killed and the player that shot him gets a point. The first player to reach a score of 3 points wins the game. A dead player cannot do anything, or be shot, for 15 seconds. After 15 seconds, his health is reset to 100% and

Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)
0-7695-1558-4/02 $17.00 © 2002 IEEE

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 pick up medikits even if their health is already at 100%. There is a flag which the player can pick up or drop for a capture the flag type game. The flag has an initial position. When a player carrying the flag dies or leaves the game, it is reset to its original position.

Partitions When a partition occurs some actions may become impossible or limited. Players should always be able to move around freely. They will 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 there may be more potential winners and we will use the time on the local system clocks at which they were declared winner of their partition to decide who won first.

1.2 Related work

Most publications on augmented reality address tracking and display problems. An overview is given in [1]. 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 [6, 9]. A survey of other group communication services, and a formal specification of their properties is given in [3]. Other work on partitionable group communication can be found in [4, 2]. Much work has also been done in the context of the Transis system, for instance in [5]. Work addressing similar timeliness requirements using timestamps is reported in [10] and on causal ordering and timeliness in [11].

2 Flare design

Since Quazoom 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 fully 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.
The three main concepts in the design of Flare are *messages* that are broadcast in the group, *game rules* that respond to messages, and the *application state*, which we split into things we call *poaps* (Part Of APlication state).
Whenever an event occurs at some node, that node sends a message to the group to notify the other nodes. Poaps will *only* be updated as a result of receiving messages from the network. So even the node that generates the event will wait until it receives its own message before updating the application state. This is shown in Figure 1.

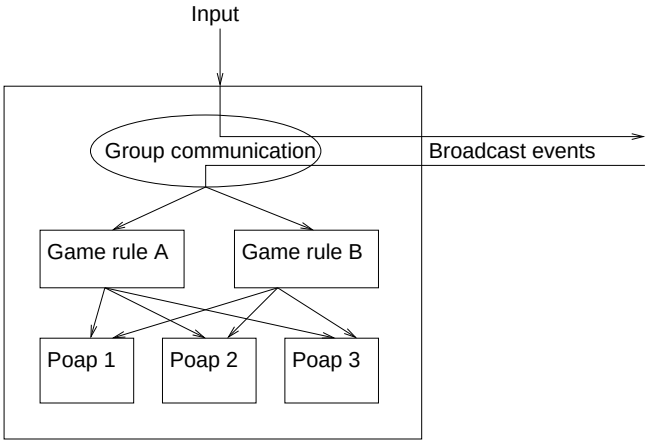


Figure 1. Flare design

2.1 Consistency

To allow the players to make progress the group communication API will support non-blocking communication even when the network is partitioned, delivering messages to the partition in which the sender is located instead of the whole group. The group communication service will also inform the members of the state of the group, providing them with new group views which contain a list of members in the current partition and a boolean indicating whether there *may* be other members outside of the current group view. So when a member receives a message, it knows which other members will also receive that message and can use this information to maintain consistency.
When a member receives a message and a game rule decides that based on the group status and poap consistency requirements it cannot perform the associated update, it will discard the message. Because all members have the same

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

Table 1. Consistency levels for poaps in Flare

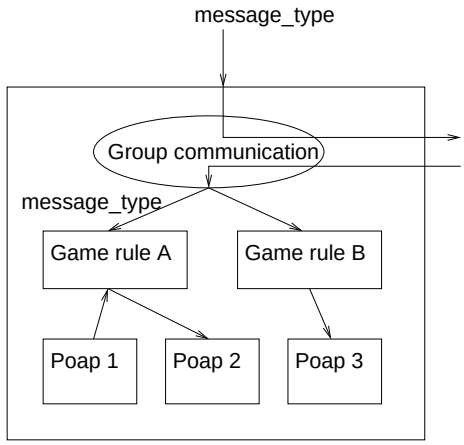
group view (we assume same view delivery [3]), all members in that partition will discard the message.
Having replicated data means that we need to formulate consistency requirements. The easiest choice would be to require all nodes to have a consistent view of the game, but this cannot be guaranteed in a partitioned network without blocking the game. Since we want to make progress in the presence of partitions, we will define different levels of consistency. We will decide whether we can update a part of the game state based on the group status and that part's consistency requirement.

Consistency requirements Because the group communication service provides different ordering guarantees two members may receive messages in a different order. When two members in a single partition haven't yet received the same set of messages, we don't mind if their states are different. We want the copies of a poap on members that have received the same set of messages to be consistent.
For copies in different partitions we may want to forfeit some consistency for some poaps to allow the application to make progress, while other poaps will have strict consistency requirements. When partitions merge, all copies will be made consistent again. We have identified three different levels of consistency we want to allow between partitions:

1. *Inconsistent*: we will allow the state in partitions to diverge. When partitions merge again, this will require a poap-specific merge function.

2. *Primary copy*: we will allow writes in the partition containing 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.

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.



message_type:Poap1(∅) → Poap2, Poap3

Figure 2. Message type signature example

Now that we have established the different consistency levels, we need to associate a consistency level with each poap in Quazoom. From the rules in Section 1.1, we deduced a set of poaps and their consistency requirements as shown in Table 1.

3 Message ordering

In this section we examine the message ordering requirements we need to guarantee the consistency requirements described previously.

3.1 Definitions

Upon receiving a message, the game rules may update a number of poaps. For deciding if and how to update a poap it may use other poaps as input. A game rule may also decide to send another message in response to the one it received.

Message signature We introduce a notation for writing the effects of a message:

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

with

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

Which means that as a result of a message of type $msgT$, the game rules *may* update poaps $pn, ...pm$, basing the value for the update on poaps $pk, ..., pl$ and on the $data$ in the message. This is illustrated in Figure 2. The illustration shows a message which is associated with two game rules.

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

- $msgT(data) : \emptyset \rightarrow msgTOut$: Unconditional updates. Messages that don't require any input poaps.
- $msgT(data) : msgTIn \rightarrow \emptyset$: Conditions. Messages that don't update any poaps, but may cause another message to be sent.
- $msgT(data) : msgTIn \rightarrow msgTOut$: Conditional updates. Messages that have input, and update poaps.

3.2 General consistency rules

Output For output consistency, we are only concerned with *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 who 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 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 do not enfore the 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 are, by definition, only used by conditions 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 when a message that reads that poap is delivered.
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.
To ensure that members 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 $update(px)$ is delivered to all or none of the members when a message of a type in $read(px)$ is delivered.

Rule specific relaxing 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.

3.3 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. Note that this is a 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 1, 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, 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 everywhere

y should be the last message delivered from $update(px)$ before x is delivered. So for global poaps we could also use FIFO or no ordering for $read(px)$ if we use a flush protocol, flushing all messages from $read(px)$, before a 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 a message type T that reads a local poap pn, writes to a global poap pm. FIFO ordering would suffice for pn, but we need total ordering for pm. 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 such a 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.

3.4 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, but this may not be possible because of the application rules.

2. Making a single node responsible for making the decision. This results in a condition type message, after which the responsible node may send 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 would pack the medikit status in the message, two players could pickup the same kit.

3.5 Example

We give a short example to see how this works in Quazoom. In this example we only consider player movement and players picking up a medikit. First let's just consider player movement. There will be a $move$ message type with the following signature:

$$move(newPosition) : \emptyset \rightarrow \{playerXpos\}$$

This is an unconditional update type. The $read(playerXpos)$ set now contains this message type and the update set is empty. Since players only send updates of their own position, there is only one member sending messages in $update(playerXpos) \cup read(playerXpos)$, and $playerXpos$ is a local poap. This means we can use FIFO ordering for this message.
Things get more complicated when we add the medikit pickup message. This message is sent whenever a player tries to pickup the medikit, and depends on the status of the medikit and the player's position. A message like the following seems appropriate:

$$medikitPickup()\{playerXpos, medikitStatus\} \rightarrow \{playerXhealth, medikitStatus\}$$

The poap $playerXhealth$ is a local poap like the position poap, but since any player can pickup the kit, $medikitStatus$ is global, which means we need to use total ordering. With this new message $update(playerXpos) \cup read(playerXpos)$ becomes $\{move, medikitPickup\}$ and we need to make sure this set is totally ordered. If $medikitPickup$ is sent using total ordering and $move$ using FIFO, this depends on whether we have a total ordering that respects FIFO as well. If it does, there is no problem, if it doesn't we have to use total ordering for the movement as well, which is expensive.
We can use one of the techniques in the previous section and change the message to

$$medikitPickup(playerXpos)\{medikitStatus\} \rightarrow \{playerXhealth, medikitStatus\}$$

Now there is only one message in $update(playerXpos) \cup read(playerXpos)$, and we can safely use FIFO for the movement message.

4 Partition and failure handling

This section describes the policy we developed to handle partitions and failures in Flare. We will allow both failures and partitions occurring at the same time (failure will be a likely cause of partitions) and the splitting and merging of partitions may also occur at the same time (for instance a node moving from one partition to the other).
When we allow both partitions and failures, we encounter two problems.

1. When a group becomes partitioned, all nodes in a partition may fail, which makes it very hard to determine whether there may still be members outside of the current partition.

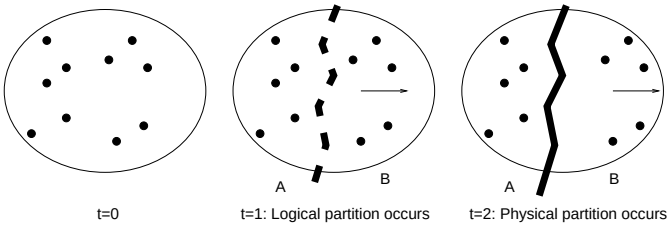


Figure 3. Preemptive partition anticipation

2. When we lose contact with a node, it is impossible to tell if this is because of a partition or because it has failed.

4.1 Partition anticipation

We will handle partitions and failures in Flare using a combination of partition anticipation and traditional majority methods. In [9] a method is described to anticipate partitions in wireless networks using a safe distance based on network card range and the speed of the user. Several other concepts like coverage estimation, network topology, signal strength information and battery life could also be used to predict partitions.
We assume to have a partition detector that will inform the group communication service that a partition is likely to occur. We will then preemptively partition the network. We call a preemtive partition a logical partition. The application will *not* be able to communicate 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 partitions will merge. The preemtive partitioning is illustrated in Figure 3, merging works in a similar way.

Limitations Note that this anticipation function can never be sure that the partition will really occur. In Figure 3, 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 anticipation function performs.
The anticipation function will never be able to anticipate all partitions, 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 4.4.

4.2 Timeout

Because all nodes in an anticipated partition may leave the group or fail, which can cause the members in the group 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'. When a member doesn't merge with the main partition within a certain timeout period, it is considered to have failed by the main partition. Both the main partition and the member can time this themselves, and with reasonably accurate clocks will draw the conclusion at more or less the same time. We set the timeout on the non-main partition sufficiently shorter than in the main partition, so 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. 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.
Note that this solves the first problem we identified above, at the cost of having members leave the group because of a timeout. The timeout period is a parameter of the policy and can be set to anything from 0 to ∞ according to application requirements.

4.3 Joining the group

To join the group a node must be in a partition with at least one member of the group. It can join whether the group is in a partitioned state or not.
When a node joins at a non main partition it needs a timeout time. This will be the latest timeout time of all members in the partition where it joins. A shorter time is pointless because the main partition will not be able to determine the group is unpartitioned before this time, and a later time isn't safe because then the main partition may decide the group is unpartitioned while the new node is still a member.

4.4 Unanticipated partitions

Although we hope the partition anticipation will take care of most partitions, there will still be unanticipated par-

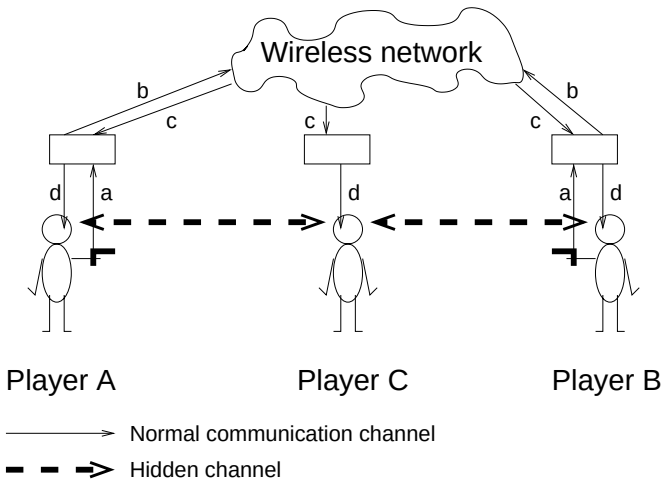


Figure 4. A hidden channel

titions 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 with how many nodes they can still communicate, they can independently decide if they are the majority partition or not. If they are not, they know the majority will have dropped them from the group, if they are, they will drop the minority from the group.

Note that regarding any unanticipated partition as a failure for the minority partition has 'solved' the second problem we identified. We no longer need to determine whether a partition has occured or a failure, because our only notion of partitions are anticipated partitions. The price we pay for this is dropping some nodes when an unanticipated partition occurs. Again, the better our partition anticipater functions, the less this will happen.

5 Timeliness

This section discusses what the timeliness requirements are to ensure the order of events in the game matches what is observed in the real world. 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 Quazoom, this will probably be vision. In Figure 4, 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, 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 will see A shooting at C first, then B. 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.

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 they 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.

5.1 Definitions

The four 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_o The minimum time between two events being perceived through the hidden channel required for an observer to be able to tell the sequence in which they occurred.

μ_r The minimum time for a player between 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 time at which event n occurred.

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

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

Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)
0-7695-1558-4/02 $17.00 © 2002 IEEE

Finaly, lets 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.

5.2 Temporal ordering

For Quazoom, 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 $\Theta^{hc} > 0$.

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

Our goal here is to determine the criteria that must be met in order to assure that two events perceived in sequence through the hidden channel are processed in the same sequence on the normal network. First lets 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 $t_b^{hc} - t_a^{hc} > \mu_o$, which implies $t_b - t_a > \mu_o$ when $\Theta^{hc} = 0$. So the message delivery should respect the $a \overset{\delta_t}{\rightarrow} b$ ordering, with $\delta_t = \mu_o$. This is shown in Figure 6 .

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_{max}^{net} to deliver and the second T_{min}^{net}. This is illustrated in Figure 6. In this case

$$\begin{aligned} t_a^{net} &= t_a + T_{max}^{net} \\ t_b^{net} &= t_b + T_{min}^{net} \\ t_b &= t_a + \delta_t \\ \Rightarrow t_b^{net} &= t_a + \delta_t + T_{min}^{net} \\ \text{we want } \quad t_a^{net} &< t_b^{net} \\ \Rightarrow t_a + T_{max}^{net} &< t_a + \delta_t + T_{min}^{net} \\ \Rightarrow T_{max}^{net} - T_{min}^{net} &< \delta_t \\ \Rightarrow \Theta^{net} &< \delta_t \end{aligned}$$

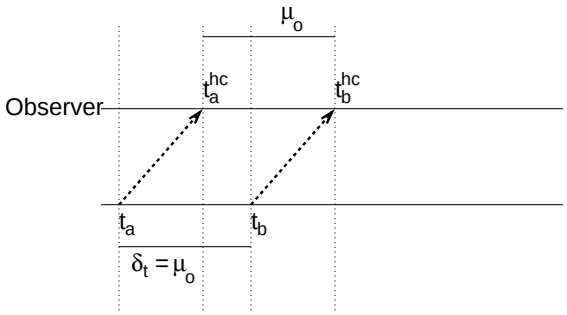


Figure 5. $t_b - t_a = t_b^{hc} - t_a^{hc}$ **when** $\Theta^{hc} = 0$

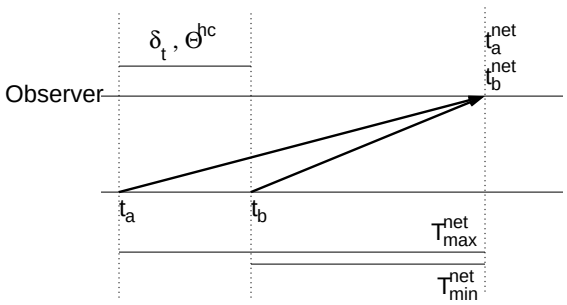


Figure 6. Worst case for delivery of messages

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 $\delta_t = \mu_o$, the timeliness requirement becomes $\Theta^{net} < \mu_o$. The uncertainty in the message delivery time needs to be smaller than the observer accuracy.

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

Now lets examine what happens when the propagation delay on the hidden channel is not constant. Again the worst case is when the two events occur with the minimal amount of time such that an observer *may* be able to tell the order.

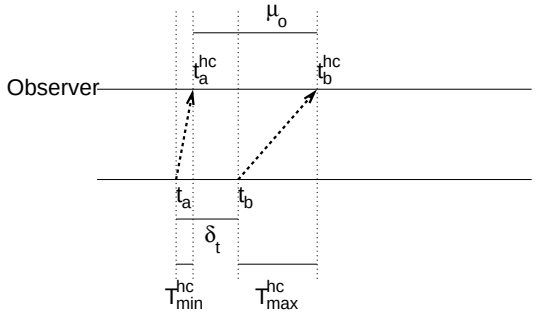


Figure 7. Worst case for temporal ordering with $\Theta^{hc} > 0$

Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)
0-7695-1558-4/02 $17.00 © 2002 IEEE

From Figure 7 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

$$\begin{aligned} t^{hc}_b &= t^{hc}_a + \mu_o \\ t^{hc}_a &= t_a + T^{hc}_{min} \\ t^{hc}_b &= t_b + T^{hc}_{max} \\ \Rightarrow t_b + T^{hc}_{max} &= t_a + T^{hc}_{min} + \mu_o \\ \Rightarrow t_b &= t_a + \mu_o - \Theta^{hc} \end{aligned}$$

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

5.2.3 Case 3: $\Theta^{hc} > \mu_o$

The previous result implies that if $\Theta^{hc} > \mu_o$, proper ordering cannot be guaranteed. This makes sense, because it implies δ_t-precedence should be respected with $\delta_t = \mu_o - \Theta^{hc} < 0$. 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 $\Theta^{hc} \leq \mu_o$. 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 8.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 8.b).
So it is easy to see that if $\Theta^{hc} < \mu_o$, then the maximum time between events arriving out of order on the hidden channel $\Theta^{hc} - \delta$ is also smaller than μ_o, 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.

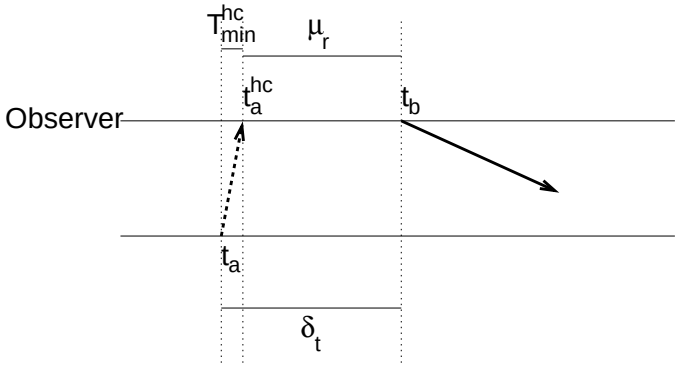


Figure 9. Minimum time between events needed for a causal relation.

5.3 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 9 it is clear that the minimum time between a and b for a causal relationship is $T^{hc}_{min} + \mu_r$. So if we apply our result for δ_t-precedence with $\delta_t = T^{hc}_{min} + \mu_r$, we see that causality is respected when $\Theta^{net} < T^{hc}_{min} + \mu_r$.

6 Summary and conclusions

In this paper we explored three issues that arise when building augmented reality applications supported by group communication on a wireless ad-hoc network. A policy was presented that allows members to make progress in a partitioned network while maintaining a certain level of consistency. It does not need to distinguish between physical partitions and failures to do this, and will eventually recover from its partitioned state. We also described an approach to determine the requirements on message ordering to maintain consistency and on timeliness to have the application state reflect the perceived sequence of events in the real world.
Using the formal method employed in [3], we have completed a formal specification of our group communication service [7]. At this stage, we are beginning our implementation of the various parts described in this paper. An algorithm for deciding whether the group may be partitioned or

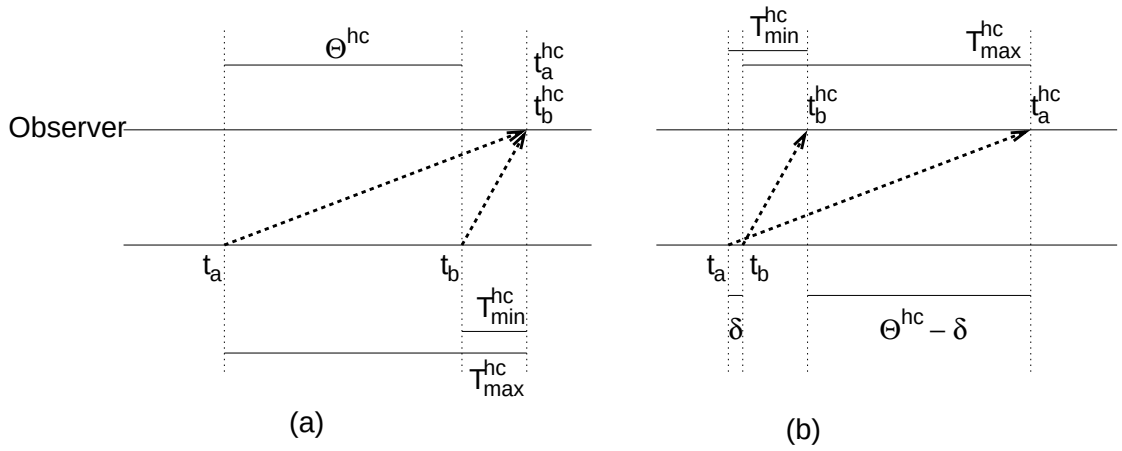


Figure 8. Events happening more than Θ^{hc} time apart are delivered in order, events perceived less than Θ^{hc} time apart may be perceived in the wrong order.

not can be found in [8]. As mentioned before, the partition anticipator forms a vital part of our policy. There is much work to be done in the area of partition anticipation. Many potentially useful sources of information need to be evaluated in order to build a reliable anticipator. This should be configurable to be more optimistic or pessimistic depending on application requirements.

Once Quazoom has been implemented, other applications should be examined to see how well our policy suits them. Although we think the requirements we identified and the policy we developed are quite general, this should be tested by looking at other applications.

7 Acknowledgements

The work described in this paper was partly supported by the Irish Higher Education Authority's Programme for Research in Third Level Institutions cycle 0 (1998-2001) and by the Future and Emerging Technologies programme of the Commission of the European Union under research contract IST-2000-26031 (CORTEX - CO-operating Real-time senTient objects: architecture and EXperimental evaluation).

The authors are grateful to past and current colleagues at Trinity College Dublin including Marc-Olivier Killijian, Greg Biegel, Adrian Fitzpatrick and Peter Barron, as well as to Paulo Verissimo of the Faculdade de Ciências da Universidade de Lisboa for their valuable input.

References

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

[2] O. Babaoglu, R. Davoli, and A. Montresor. Group communication in partitionable systems: Specification and algorithms. *IEEE Transactions on Software Engineering*, 27(4):308–336, Apr. 2001.

[3] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *To appear in ACM Computing Surveys*.

[4] A. Fekete and N. L. A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, May 2001.

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

[6] M.-O. Killijian, R. Cunningham, R. Meier, L. Mazare, and 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*.

[7] N. Reijers and V. Cahill. A formal specification of a group communication service for flare (draft). *Department of Computer Science, Trinity College Dublin Technical Report TCD-CS-2001-46*, 2001.

[8] N. Reijers, R. Cunningham, R. Meier, B. Hughes, G. Gaertner, and V. Cahill. Requirements for a group communication service for flare. *Department of Computer Science, Trinity College Dublin Technical Report TCD-CS-2001-45*, 2001.

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

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

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