An Adaptive Real-Time Group Communication Protocol

Carlos Almeida, Paulo Veríssimo
INESC - IST
Technical University of Lisboa
Instituto de Engenharia de Sistemas e Computadores,
R. Alves Redol, 9 - 6º - 1000 LISBOA - Portugal

Abstract

*In this paper, we address the problem of having real-time group communication in a system that is not completely synchronous (*quasi-synchronous*). This situation may happen when system load is not completely controlled and so, overload scenarios are not ruled out. In such scenarios, and for most of the networks used in this type of environments, temporal guarantees can only be made for the highest priority messages.*

The paper describes a group communication protocol that provides a total temporal order *in such an environment. It adapts to the uncertainty of the environment by means of a failure detection service, in order to be able to achieve its safety properties in a timely fashion.*

The protocol also tries to provide early-delivery while preserving its ordering property.

1 Introduction

There are different classes of real-time applications. Indeed, although they all have some timing constraints, the way they should behave in the case where those constraints are not met, differs from one situation to another. When the application is a very critical one, having in its dependence human life or high cost resources, then one must make sure that all timing constraints are met. This implies the use of a resource adequacy policy: all resources needed for a worst-case scenario must be reserved in advance. Applications like these are, for example, nuclear plant control and avionics.

There are, however, other real-time applications where meeting timing constraints is important, but where it is acceptable to eventually miss some of them, if the most important ones are achieved. Most of these applications are distributed, and have dynamic characteristics or run in dynamic environments, where it is difficult to evaluate a worst-case scenario or the one obtained is so far from the normal case that it makes the use of a resource adequacy policy not cost-effective. Examples of this type of real-time application can be found in settings such as: factory plant control; command, control and communications systems; air traffic control.

In such situations, it is desirable to help the application programmer in adapting to this environment by providing the adequate system support. One such kind of support of recognized utility is real-time group - multicast - communication. There are a number of published protocols that deal with the problems of reliable delivery and ordering in synchronous systems, i.e. systems where timeliness bounds (on speed of processors, message transit delay, etc.) are known [3, 9]. The problem becomes complex in dynamic and incompletely specified settings, such as the ones targeted by the application classes we are addressing. Indeed, most of previously published work treats such environments under asynchronous assumptions, and thus with no utility for real-time. In this paper, we address the problem of having real-time group communication in a system that is not completely synchronous (*quasi-synchronous*). This situation may happen when system load is not completely controlled and so, overload scenarios are not ruled out. In such scenarios, and for most of the networks used in this type of environments, temporal guarantees can only be made for the highest priority messages.

This paper describes a group communication protocol that provides a *total temporal order* in such an environment. It adapts to the uncertainty of the environment by means of a failure detection service, in order to be able to achieve its safety properties in a timely fashion.

The paper is organized as follows: in the next section we present the system model. Section 3 contains the multicast protocol. Section 4 is about the failure

detection service used by the protocol. Section 5 contains references to some related work. The paper ends with the conclusions and references to future work.

2 System Model

We begin by introducing the system model that we are assuming in the development of the protocol presented in this document. The explanation of the protocol is decomposed into two parts to make the description simpler. The difference lies in the system model, in what concerns the type of synchronism assumed. We start by assuming a completely synchronous system for the first part, and then, for the second part, we relax some of the assumptions made about the synchronism of the data channels.

Thus, as a starting point we consider that the system is *synchronous* in the sense that there are known bounds on process speed, message transmission delay and local clock rate drift. The modifications for the second part of the protocol are explained below.

There is a *global notion of time*, i.e. local clocks are synchronized. The precision of clock synchronization is π. This global time can also be related to the real time with a given accuracy α. However, for the protocol presented here, the accuracy is not relevant [1]. We also ignore clock granularity for now. We will not address here the clock synchronization it self. For now, we will assume that it is available. However, one must have in mind that it will require some resources: a dedicated network or some bandwidth in a synchronous channel.

From the point-of-view of the protocol, the system is composed of a set P of participants noted *p, q, r, ...* At this point, a participant may be a processor (or a site) or a process – for the protocol, it is an *endpoint*. The set, P, of participants forms a *group* and each element of the set is a *member* of that group. A participant that fails is removed from the group and new members can join the group. This is handled by a *membership protocol*.

The basic fault model assumed considers the existence of omissions, crashes and timing faults but not arbitrary faults. The omission degree is k, the maximum number of crashes is f, and timing faults are converted to omissions [2].

We denote by $send_p(m)$ the sending of message m by participant p. Likewise, $receive_p(m)$ and $deliver_p(m)$ denote respectively the reception and the delivery of message m by participant p.

The current time for a given participant is denoted by T and is the value read from the local clock, which is synchronized as said above.

When a message is sent, it is timestamped with the current time of the sender. We denote the time of a given event e at participant p by $T^p(e)$. When the superscript is omitted (for simplicity), we mean the time as seen by the participant where the event takes place. A message m sent by a participant p is timestamped with $T_m = T(send_p(m))$.

Message delivery time, of message m for participant p, $T_D^p(m)$, is the time interval between the send event and the deliver event: $T_D^p = T(deliver_p(m)) - T(send(m))$

There is a minimum message delivery time $T_{D_{min}} = min_p(T_D^p(m))$ and a maximum message delivery time $T_{D_{max}} = max_p(T_D^p(m))$.

We also assume that $T_D \approx T_{D_{min}} \ll T_{D_{max}}$.

The services offered by our protocol are: *early-delivery* and *total temporal order*. By early-delivery we mean the tentative of delivering messages before $T_{D_{max}}$, while preserving the ordering property of the protocol. By *total temporal order* we mean a total order based on message timestamps obtained from synchronized clocks, with ties broken using participant identifiers [3]. So, this is in fact a logical order based on local clocks and not an order based on an external real time clock. Due to the fact that clock precision π may be not null, the order obtained might be different from the one seen by an external observer. However, causality is preserved as long as there are no other *communication* channels with a latency smaller than $T_{D_{max}}$. In order to preserve causality even considering other events not directly related to normal communication channels, those events also need to be timestamped and ordered together with generic messages. In [15] there is a more detailed study about these causality aspects and the conditions to secure precedence (potential causality). There, it is observed that in a distributed computer system or in a physical process, it takes a finite amount of time for an input event to *cause* an output event. Supposing there is a known such minimum time δ_t for a given system, then only messages separated by more than δ_t may be causally related in that system. In consequence, it is of no utility to order messages with a smaller separation. Based on that, conditions for correct protocol execution can be derived for any setting. In this paper, for simplicity, we will consider the absence of

[1] It will be important, though, when building the complete system and having interactions with the external environment.

[2] Later, in the second part of the protocol, we will address timing faults in a more generic way.

[3] In [8] they call this *lexicographic order*.

communication outside our protocol.

We are also assuming that we do not have partitions. We will, eventually, address that subject in a future document. At least, we plan to address the inaccessibility problem [16, 14].

Relaxing synchronism

For the second part of the protocol, we make some modifications to the system model. We relax the synchrony of the system. Instead of having a completely synchronous system as defined above, we will assume a *quasi-synchronous* system. By *quasi-synchronous* we mean that the bounds referred to above (process speed, message transmission delay, and clock rate drift) exist, but some or all of them are not precisely known, or they are too far from the normal case that in practice we are going to use other values (closer to the normal case). In both cases it means that there is a non-null probability that the values we pick are not correct. This is a realistic scenario when there are situations of overload. For the protocol presented in this paper we will consider only the relaxing of the message transmission delay.

The difference between this model and the one (*Partial Synchrony*) presented by Dwork *et al.* in [5] is that here we need to pick a given value *a priori* because we want to have a timely service. In [5] the bound is not known or the point in time when it starts to hold is not known. In order to have the protocols working for a generic case, they are built without using a pre-defined value for those unknown parameters. The main difference to our case is that they are concerned essentially with safety properties not related with time, assuming a liveness perspective, whereas we are deeply concerned with time under the perspective of timeliness. This essentially means having safety in a bounded time in a *quasi-synchronous* system. In order to achieve this we need an additional mechanism. That mechanism is a **failure detection service**. However, this service must have a synchronous behavior. The main idea is to realize that we can have different assumptions for a generic data channel and for the basic support to build the failure detection service. This can be a dedicated network, as it is sometimes used for clock synchronization, or can be a virtual channel over a more traditional real-time network.

Having a channel with *better* characteristics is something that, we think, is reasonable to assume in some scenarios usually available in real-time environments. In a network like token-bus, for example, only high priority messages have hard real-time guarantees. In order to have guarantees for all messages, the configuration and network load must be completely controlled. That is not always possible in dynamic environments. In such situations, the best that can be done is to provide some guarantees to the highest priority messages. Another network technology example, where different guarantees are given to different channels, is ATM (Asynchronous Transfer Mode). In this case it is possible to reserve channels with a given bandwidth and with a given level of guarantee associated with them.

Having this *special channel* should be enough to implement the failure detection service. Generic data channels may have more relaxed assumptions, because validations are done using the services of the failure detector.

So, in our system model for the second part of the protocol, we relaxed the synchrony of a generic data communication channel, but we keep the assumption of a synchronous channel for control messages that will be used to implement the failure detection service. Note that this channel, although providing guaranteed latency does not need to have a large bandwidth because it is not to be used for generic data messages.

In this model, by relaxing the synchronism of data channels, the value used for maximum message delivery time, $T_{D_{max}}$, may be exceeded in some executions. This will lead to *timing faults*. The protocol must handle this situation. By "handling" we mean essentially "detecting in a timely fashion". It does that with the help of the failure detection service referred above and explained in section 4. The action to take when these timing faults are detected, can be made dependent on some configuration and the final decision should be application dependent.

A very important aspect of a given system model and a given protocol, is its validity. A model will be completely useless if it does not correspond to reality: protocols may be correct accordingly to the model, but fail in practice. In order to have a protocol that works correctly, it needs to be correct in the sense that it corresponds to the assumed model, and the model must be correct in the sense that it corresponds to the reality. Coverage [10, 11] is a measure of the representativity of the situations to which a system is submitted during its validation compared to the actual situations it will be confronted with during its operational life. Two different protocols built assuming the same fault model can have different coverage. They will both work correctly as long as faults respect the assumed fault model. But one of them can eventually tolerate additional faults. For example, acknowledge-

based protocols have a better coverage than diffusion-ones [16].

In scenarios where there is some probability that the assumed model is not completely correct, like the one we are addressing in the second part of the protocol, it may be desirable to use acknowledge-based protocols.

3 Real-Time Multicast Protocol

We present in this section a group communication protocol that aims to provide qualities-of-service (QoSs) useful for some real-time applications running on the type of environment represented by the model above.

There are mainly three aspects that we want to address with this protocol. They are: *early-delivery, total temporal order, handling of timing faults.* In order to simplify the explanation, we will start by presenting the part of the protocol that deals with the first two aspects, assuming that there are no timing faults. We will address the timing fault situation afterwards. Another simplification is to assume the existence of two basic multicast primitives – *disseminate()* and *accept()* – that are responsible for reliable message dissemination to participant sites. This way we can concentrate on the aspects of the protocol that are related to the QoSs that we want to offer. Optimizations can eventually be obtained by the integration of these two levels, but that subject will be addressed later.

So, we will now consider a synchronous system where the basic primitives referred above (*disseminate() / accept()*), have the following characteristics:

- **agreement:** If a correct participant accepts a message, then all correct participants accept that message.
- **order:** There is no assumption about order.
- **termination:** A message sent by a correct participant at time T is accepted by all correct participants before $T + T_{D_{max}}$ on the recipients clock.

We also have:

- $T_D \approx T_{D_{min}} \ll T_{D_{max}}$
- it is the responsibility of these primitives to deal with omission and crash faults.

These basic primitives can easily be obtained from other published protocols like [3] or [13], for example. We do not make any assumption about the type of protocol being diffusion-based or acknowledge-based.

Having a basic protocol that has a known maximum delivery time (latency), it is possible to obtain a *total temporal order* by always waiting for that maximum delivery time and delivering messages according to timestamps based on synchronized clocks. That is, for example, the case of the Δ-protocols from Cristian *et al.* [3]. The latency Δ of those protocols depends directly on the worst-case message delay, and message delivery is only done at time $t + \Delta$ where t is the timestamp of the message.

However, in a system where $T_{D_{min}} \ll T_{D_{max}}$, and where *"at same time"* delivery is not needed, always waiting the worst-case delay may be a very high cost to pay, specially if the normal message delivery time is closer to $T_{D_{min}}$ than to $T_{D_{max}}$, which is usually the case. For such a scenario, it makes sense to try to provide early-delivery. If needed, the *"at same time"* delivery can be done at an upper layer thus avoiding message buffering at low layers of the system.

From the point-of-view of providing early-delivery, our protocol works tentatively, by using cheap communication primitives, and exploiting the fact that $T_D \ll T_{D_{max}}$. If an attempt does not succeed, we have the latency provided by the basic dissemination primitive. The protocol must also provide *total temporal order*. So, early-delivery can only be done when that order is secured.

Our protocol is inspired on the protocols described in [8]. Those protocols assume a diffusion primitive with a latency Δ, and try to obtain early-delivery by the use of acknowledgments. They also provide a *total temporal order* (lexicographic order) based on timestamps obtained from synchronized clocks. Early-delivery is obtained for the sender when it receives an acknowledgment (*ack*) from all other processors. A given processor only sends an acknowledgment to a message m when there are no *conflicting messages* – an accepted but not yet delivered message m' that precedes m ($m' \prec m$, usually $T_{m'} < T_m$).

In our protocol, we introduced some modifications in order to be able to provide early-delivery to all participants and not only the sender. We also removed the restriction about the acknowledgment. We always send the *ack*, but with additional information to handle the situation of conflicting messages.

Providing Early-Delivery

Figure 1 shows how the protocol works to provide early-delivery. After receiving a message each participant sends an *ack* to the sender. When the sender

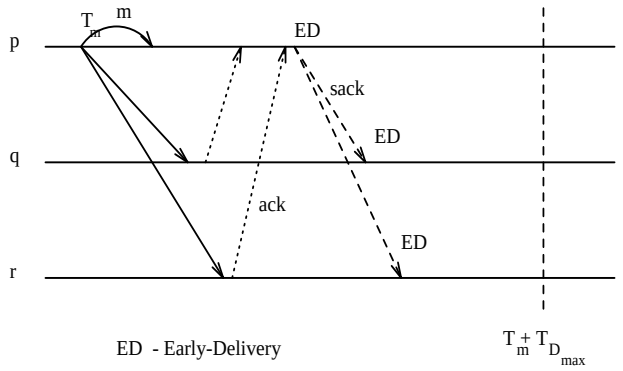

Figure 1: Early-delivery protocol

has received an *ack* from all members of the group, it can early-deliver because it knows that they all have received the message and they will deliver it at most by the time $T_m + T_{D_{max}}$. To provide early-delivery for the other participants, the sender sends a special acknowledgment (*sack*) telling that it has received all the *acks*. When a participant receives this *sack* it can early-deliver. Note that early-delivery does not depend on the maximum message delivery time but on the minimum message delivery time (a participant can make an early-delivery as soon as the special *ack* is received) [4].

Preserving order and causality

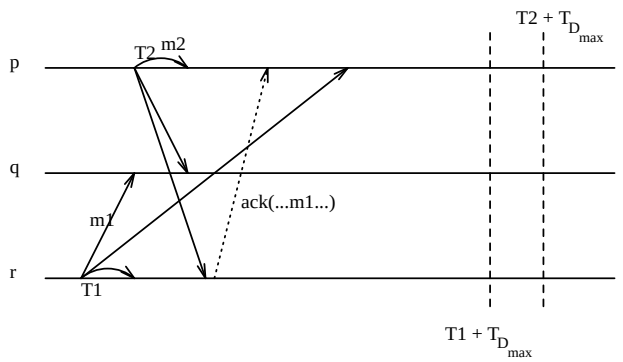

Figure 2: Preserving order

In order to make sure that the order of delivery is correct, an option is to only send the *ack* when there is no *pending message* – an accepted or sent message not yet delivered – that precedes the current one (*conflicting message*). This is what is done in the original protocol of [8]. However, this way (omission of *acks*), messages not acknowledged will need to wait the maximum message delivery time, until $T_m + T_{D_{max}}$ to be delivered, because it will be the only way to grant a correct order. To avoid this delay and make an early-delivery as soon as possible, the *ack* is always sent but with additional information needed for correct order delivery. This information is about pending messages. It is easy to see that there will be at least one participant (the sender of the message that is pending) that knows the correct order and will provide that information in the *ack* message. This information will be disseminated in the *sack* thus reaching (eventually) every participant. Note that if an *ack* is lost (or arrives too late), message delivery will be based on the timestamp, at time $T_m + T_{D_{max}}$, and the order will be correct. In this case, early-delivery is not obtained at those participants that have not received (in time) the acknowledgment. When receiving an *ack* with information about a conflicting message, the participant knows that if it has not delivered yet that conflicting message, it can not deliver the current one (the message the *ack* is related to). But, as soon as all the conflicting messages are delivered, the new one can also be delivered.

In order to implement this functionality, the protocol must handle some data structures. Each participant must keep some information that it adds to an *ack* message. More precisely, it must keep some history information about messages not delivered yet – that is, information about messages with timestamps belonging to the interval $[T - T_{D_{max}}, T]$, where $T_{D_{max}}$ is the latency of the protocol and T is the current time (as read in the local clock). When receiving a given message m with timestamp T_m, a participant p sends an *ack* to the sender of m. This *ack* "contains" an "OK" if p does not know of any message that can prevent an early-delivery of m. If p, based on the kept history, knows about any message that was not delivered yet but is supposed to be delivered before m, i.e., $T_{m'} < T_m$, then the *ack* "contains" a "NOT_OK" and the timestamps of those pending messages. A participant keeps a list of missing messages. When one of these messages is delivered it checks if there are others that can also be delivered at that time. One possible implementation is to add dummy messages in the right position of the queue of messages to deliver as soon as they are "discovered". Those dummy messages are replaced by the appropriate messages when they arrive. Assuming that the model is correct, those substitutions are done before $T_m + T_{D_{max}}$. If the replacement of the dummy message, by the right one, is not done in time, a failure is detected.

[4] It might need to wait for some pending messages as explained below.

Figure 3 shows a simplified sketch of the protocol (garbage collection is not represented).

```
When p wants to rtcast (m) do
    T_m ← current_time
    insert (T_m, p, m) in SENT_p
    disseminate(T_m, p, m) od
When p accepts (T_m, q, m) do
    insert (T_m, q, m) in ACCEPT_p
    hist_m ← SENT_p ∪ ACCEPT_p
    send(q, ack_m, hist_m) od
Delivery:
    if p received (ack_m, hist_m) from all r
        and current_time < T_m + T_Dmax then do
            obtain sack_m and shist_m
            send (all, sack_m, shist_m)
    od fi
    if p receives (sack_m, shist_m)
        and current_time < T_m + T_Dmax then do
            if there is no m' ≺ m such that
                p has not delivered (m') yet then do
                    insert (T_m, q, m) in DELIVER_p
                    deliver (m)
                    check delivery of pending messages
    od fi od fi
    When current_time = T_m + T_Dmax do
        for each (T_m, q, m) that p has accepted
            in lexicographic order do
                if p has not delivered (T_m, q, m) then do
                    deliver (T_m, q, m)
                    check delivery of pending messages
    od fi od od
```

Figure 3: Sketch of the protocol

Since clocks are not perfectly synchronized, a difference of π between clocks of two participants is possible. If the value of π is of the order of the message delivery time (and that is usually the case for most systems), then it is possible to have a situation where, from the point-of-view of the receiver, the message delivery time is negative ($T(receive_q(m)) < T(send_p(m))$). If by the time participant q receives that message, it also wants to send a message, then the correct order of delivery is not secured. To avoid this situation, either the reception of message m must be delayed, or participant q must delay the transmission of its own message (this is not represented in the sketch of the protocol in figure 3). Note that this delay is always less than π. With this delay the protocol provides a global (and causal) order [5].

[5] Causal order is only granted if there are no hidden channels with a latency smaller than the one provided by the communication protocol.

Handling timing faults

We will now consider the case where the assumption about a given maximum message delivery time might be violated. In this case a failure detection service is needed. The protocol uses the failure detection service to preserve safety. As we want this to be done in a known and bounded time, there are some requirements that the failure detection service must satisfy.

One of the most important issues in real-time protocols is the need to have known and bounded times for message delivery. If the underlying system (including network) is not very deterministic, then it is difficult (if not impossible) to provide a protocol that is always correct and whose timings are acceptable for most applications. In these cases, we need to have some kind of tradeoff between time and failures. If the delivery of a message is delayed for a long time in order to make sure that we are close to the worst-case of message delivery time, then the protocol will be useless for most applications. On the other hand, if we assume a low message delivery time, then the protocol will fail many times (the assumption is not realistic).

If the assumptions made about message delays are not correct and there is a non null probability of having some violations, then it is not possible to make guarantees about the properties of the protocols without additional mechanisms. One solution is to assume some "reasonable" values, and add some failure detection mechanisms.

The major problem with the protocol described above is that it can use the maximum message delivery time to achieve order. The value $T_{D_{max}}$ is determined assuming a given worst-case scenario. However, due to the non-determinism of the supporting system, it is possible to have sometimes a message delivery time that is greater than the assumed value. This is because the worst-case was not the real worst-case, or, for practical reasons, we used a different value. In that case some care must be taken to avoid the failure of the protocol. We must detect and handle some particular situations.

When a message arrives at a participant after $T_m + T_{D_{max}}$, it knows that there is a timing fault. However, it is not clear what is the best action to take. Should the receiver fail? Should the sender fail? Should the message be ignored? The problem of ignoring the message is that, as the protocol works, the sender will deliver the message by time $T_m + T_{D_{max}}$. To avoid that situation, the protocol can be modified to make the delivery at the sender only possible if there is at least one acknowledgment (or a majority). However, there is always the problem of losing an *ack*

and not knowing if other participants have received the message.

If there is only one participant that receives the message after the deadline, then it should fail (or call an exception handler that will decide what to do). But, without any other mechanism, this faulty participant only detects that situation when the late message is received. In that case it is possible that it has already sent some other messages, causing a contamination problem [7]. If the communication is only done inside the group, then, by adding some history to the messages, it is possible to detect the contamination problem at the receiver. But, if messages are sent outside the group, then it might not be possible to detect the situation at reception time.

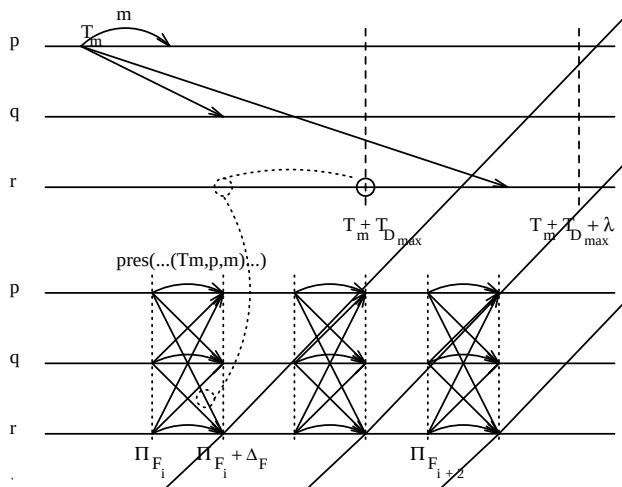

Figure 4: Use of periodic messages to detect faults

A possible solution to this problem is to avoid communication outside the group or to have periodic messages (presence) from all participants thus allowing a given participant to detect its failure (see figure 4). These periodic messages contain information about pending messages and current members of the group. Like this it is possible, not only to detect timing faults, but also to have a bounded time for the detection of the leaving of a member. The cost associated with this modification to handle timing faults is an increase on the number of messages in the network (periodic messages), and an increase on the latency of the communication protocol. The new latency is $T_{D_{max}} + \lambda$, with $\lambda \geq \Pi_F + \Delta_F$. Π_F and Δ_F are respectively the period and the latency of the failure detection service. This increase in the latency is necessary in order to make sure that the information needed to make a decision is received.

Note that if a presence message is not received in time, it means that there is a fault. The difference to other messages is that this one was expected. For now, we will assume that we have a synchronous channel to implement these presence messages. We plan to address the problem of also having some timing faults on this channel in a future document.

Information needed to make a decision

Before making the decision of delivering or not, a given message, or declaring someone faulty, the protocol must have the information needed to make that decision. Some of that information is obtained using the failure detection service. Below, associated with each item, is the time at which is possible to obtain the given information.

Information needed:

- any other message sent before and not yet delivered? $(T_m + \Pi_F + \Delta_F)$
- was the message received by anyone? $(T_m + T_{D_{max}} + \Pi_F + \Delta_F)$
- what is the criteria for failure decision? (deadline to receive the message, $T_m + T_{D_{max}}$)
- when should a failure be assumed? as soon as possible? $(T_m + T_{D_{max}} \ \vee \ T_m + \Pi_F + \Delta_F \ \vee \ T_m + T_D)$ only when there is some inconsistency? (reception time, $T_m + T_D \ (T_D > T_{D_{max}})$)

The failure detection service should provide an interface to obtain this information and specify configuration options.

The information that in the first part of the protocol was disseminated in the *ack* messages, is now also obtained from the failure detection service.

4 Failure Detection Service

The Failure Detection Service is a key component in our system. It provides the means to achieve safety in a timely manner in *quasi-synchronous* systems. It works as an oracle that is used by the protocols in order for them to be able to grant safety in a not completely synchronous system. By implementing this service using special channels (synchronous), we are also able to build protocols that achieve safety in a known and bounded time. This is of utmost importance for real-time.

The Failure Detection Service can provide two different modes of operation. It can use periodic messages and/or react to requests. Every machine involved in the communication will have a copy of this service. Messages exchanged are not simple "I'm alive messages". They carry additional information that can be provided by the communication protocols that are using the service. Of course, the amount of this information provided by the communication protocols is limited. Otherwise, it would be a generic data channel and there would be no need for the generic communication channels used by the protocols.

Being a restricted bandwidth channel, makes it possible to obtain the desired properties (synchronism) in a larger setting. Usually, in the type of environments we are addressing, temporal guarantees can only be made for the highest priority messages. Our idea, is to split the communication protocol so as to use a *better* channel to carry the vital information that is used to validate the rest of the protocol. As a rough example, this can be seen as using a telephone system to validate the delivery of messages sent using a low priority surface mail. This does not provide timeliness *per se*, but provides the ability to achieve safety in a bounded time. Applications can then adapt to a given situation in the way that best suits them. And with the *right* replication level, it may be possible to obtain the desired quality-of-service.

In the previous section we have already introduced some parameters related to the Failure Detection Service, namely period, latency and some of the information exchanged with communication protocols. Those parameters are configurable and depend on the system support and application needs. There is also a trade-off between the number of messages (period) and the increase on the latency of the communication protocol.

Having a synchronous channel, the implementation of this service is easy to obtain. It can be based on some published protocols like [2], for example. We mainly need to add an interface to exchange information with the communication protocols. A detailed description of that interface will be the subject of another document for space availability reasons. The group membership protocol can also be implemented using this synchronous channel if we have a bound on the number of groups and members.

5 Short reference to some related work

The scenario that we are addressing, considering a *quasi-synchronous* system, is somehow similar to the one related to the impossibility result in asynchronous systems [6]. That would be the case without any bound in message delay. In our case, it is not exactly the same thing because that bound is assumed to exist, we only pick the wrong value – the bound exists but it is not known [6]. Our situation is also similar to the one described by Dwork *et al.* in [5] about *partial synchrony*. However, they are mainly concerned with safety properties – being able to achieve consensus in partial synchronous systems. Timeliness is not a major goal. In our case, we would like to have both things. The problem is that in order to have timeliness (known latency) we have to assume known bounds for message delays, but if the bounds used are not correct then we are in a situation similar to the one where the impossibility result of [6] holds.

In order to solve this problem we use an approach similar to the one used by some research groups to solve the asynchronous problem – a failure detector [12, 1]. However, there is a major difference between these two scenarios – we want to reach agreement in a known bounded time. For that, we need to have (real) synchronous channels to implement the failure detector. Our claim is that it is reasonable to make that assumption on most real-time environments, if those channels have a restricted bandwidth.

6 Conclusions and Future Work

The protocol presented here is part of a more generic architecture that aims at providing the infrastructure to build fault-tolerant real-time applications that have some specific characteristics. Those characteristics are related to the dynamic behavior of the application and/or to some uncertainty (non-determinism) of the environment.

This work shows some possible solutions to take advantage of the environment in such a way that it is possible to have some real-time applications in situations where a more traditional approach would not work. The ability to adapt to a changing environment also allows for a graceful degradation of quality-of-service which is acceptable for many applications.

As future work, we plan to pursue the development of the system referred above. We also plan to have other more relaxed qualities-of-service in the protocol. Some applications (soft real-time), can live with some timing faults. Being able to provide information about their occurrence may be enough for the application. It will, eventually, adapt to that scenario and continue to offer an useful quality-of-service. Aspects to consider in the qualities-of-service to offer are: agreement,

[6] Or the known bound is so large and so far from the normal case, that in practice we need to use a different value.

order (FIFO, causal, total, temporal), and timeliness. These properties are not always needed together. It is possible to have several combinations of them. Applications should not pay for something that they do not need.

In cases where timing faults are important, it is also possible to minimize their impact by providing a *light weight* membership service. At a low level, a participant will continue to receive messages, thus making it possible to have a fast recovery. At a high level, a participant that failed to receive a given message in time, will be temporarily "disconnected", making sure that safety is preserved. As soon as recovery is done, it can rejoin the group.

Another aspect that we plan to address as future work, is the study of the implications of also relaxing the synchronism assumptions of the failure detection service. We would like to know exactly what type of qualities-of-service can still be provided in that case. This study is closely related to the work done by Dolev *et. al.* in [4].

References

[1] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems (preliminary version). Technical report, Department of Computer Science, Cornell University, Ithaca, USA, July 1991.

[2] Flaviu Cristian. Agreeing on who is present and who is absent in a synchronous distributed system. In *Digest of Papers, The 18th International Symposium on Fault-Tolerant Computing*, Tokyo - Japan, June 1988. IEEE.

[3] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. Technical report RJ-5244, IBM, July 1986. An early version appeared in Proc. of the Fifteenth International Symposium on Fault-Tolerant Computing, Michigan, June 1985.

[4] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *24th Annual Symposium on Foundations of Computer Science*, November 1983.

[5] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery*, 32(2):374–382, April 1985.

[7] A. Gopal and S. Toueg. Inconsistency and contaminations. In *Proceedings of the Tenth ACM Annual Symposium on Principles of Distributed Computing*, pages 257–272, Montreal, Quebec, 1991.

[8] Agei Gopal, Ray Strong, Sam Toueg, and Flaviu Cristian. Early-delivery atomic broadcast. In *Proc. of Ninth ACM Symposium on Principles of Distributed Computing*, Quebec City, Canada, August 1990.

[9] H. Kopetz and G. Grunsteidl. TTP - a Time-Triggered Protocol for Fault-Tolerant Real-Time Systems. In *Digest of Papers, The 23th International Symposium on Fault-Tolerant Computing*, pages 524–533, Toulouse, France, June 1993. IEEE.

[10] J. C. Laprie. Dependability: A Unifying Concept for Reliable Computing and Fault-Tolerance. In T. Anderson, editor, *Dependability of Resilient Computers*. BSP Professional Books, 1989.

[11] David Powell. Failure mode assumptions and assumption coverage. In *Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing Systems*, page 386. IEEE, 1992.

[12] Aleta M. Ricciardi and Kenneth P. Birman. Using process groups to implement failure detection in asynchronous environemnts. Technical Report TR 91-1188, Cornell University, Department of Computer Science, 1991.

[13] L. Rodrigues and P. Veríssimo. *x*AMp: a Multi-primitive Group Communications Service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 112–121, Houston, Texas, October 1992. IEEE. INESC AR/66-92.

[14] J. Rufino and P. Veríssimo. A study on the inaccessibility characteristics of ISO 8802/4 Token-Bus LANs. In *Proceedings of the IEEE INFOCOM'92 Conference on Computer Communications*, Florence, Italy, May 1992. IEEE. also INESC AR 16-92.

[15] P. Veríssimo. Causal Delivery Protocols in Real-time Systems: a Generic Model. *Journal of Real-Time Systems, Kluwer Eds.*, to appear 1995.

[16] P. Veríssimo, J. Rufino, and L. Rodrigues. Enforcing real-time behaviour of LAN-based protocols. In *Proceedings of the 10th IFAC Workshop on Distributed Computer Control Systems*, Semmering, Austria, September 1991. IFAC.