# Timing Failure Detection and Real-Time Group Communication in *Quasi-Synchronous* Systems

Carlos Almeida
Instituto Superior Técnico - Universidade Técnica de Lisboa
Instituto de Engenharia de Sistemas e Computadores
R. Alves Redol, 9 - $6^o$ - 1000 Lisboa - Portugal
cra@inesc.pt

Paulo Veríssimo
Faculdade de Ciências da Universidade de Lisboa
Instituto de Engenharia de Sistemas e Computadores
R. Alves Redol, 9 - $6^o$ - 1000 Lisboa - Portugal
pjv@di.fc.ul.pt

## Abstract

*This paper addresses the problem of having real-time group communication in systems that are not completely synchronous (*quasi-synchronous*). This situation can happen when load is not completely controlled thus not ruling out overload scenarios. This is usually related to dynamic characteristics of the application and/or environment. Usually, in such systems, temporal guarantees can only be made to the highest priority messages.*

*In such scenario we propose to use the highest priority messages (or a dedicated synchronous channel if available) to implement a timing failure detection service. By using this service together with the communication protocols, we are able to provide safety in a timely fashion, which is of utmost importance for distributed fault-tolerant real-time applications.*

## 1. Introduction

When talking about real-time applications, one usually thinks about embedded control applications or safety critical applications with *hard deadlines* associated with them. However, there are other real-time applications, and, not all of them have *hard deadlines*. There are some distributed control applications, and some distributed data bases, for example, where although there are some timing constraints, missing some deadlines once in a while is not a major problem if the main goals are achieved. In such cases, it is not cost-effective to use a resource adequacy policy to try and meet all the deadlines in a worst-case scenario. It would be too expensive.

Another aspect is related to the characteristics of the application and environment. When the application is dynamic or runs in a changing environment, it is very difficult (if not impossible) to evaluate a worst-case scenario. Sometimes, real-time applications also need to live together with other non-real-time applications. All these scenarios make it difficult and very expensive to have a completely controlled system. However, if system load is not completely controlled, overload scenarios are not ruled out, which makes the development of real-time applications an harder job.

In situations like this, one needs to make a tradeoff between timeliness and timing failures in order to obtain a cost-effective system [1]. The idea, is to try to be able to take the maximum advantage of a given environment while satisfying applications main needs. Applications should be able to select the desired qualities-of-service and adapt whenever the system moves from a *working envelope* to another.

In this paper we propose to use a timing failure detection service that is used together with the group communication protocols in order to achieve safety in a timely fashion. Based on this, it is possible to build different communication protocols with different qualities-of-service.

The paper is organized as follow: in section 2 we give a better clarification of the problem that we are trying to solve. Section 3 introduces the system model. Section 4 explains

[1] Of course, this approach only makes sense for the type of applications we were describing. For safety critical applications having on its dependence lives or high cost components, a resource adequacy policy is supposed to be used.

the timing failure detection service. Section 5 describes the group communication protocol. The paper ends with the conclusions and references to future work.

## 2. The problem and related work

The main problem that we want to address in this paper is how to have real-time group communication in a system that is not completely synchronous. Due to the fact that we want to target applications and environments that have a dynamic behavior, we will have a system which has a load that is not completely controlled. This will lead to the existence of some timing faults.

This scenario happens because it is very difficult to evaluate a worst-case scenario in such conditions, or the cost of providing a resource adequacy policy is not cost-effective for the type of application in consideration.

Having overload scenarios, makes it difficult to know the bounds (worst-case) for all activities. Usually it is only possible to have some guarantees for the highest priority activities. The others will have, eventually, a bound, but it is not known or the known value is so large and so far from the normal case, that in practice another value must be used.

This means that instead of having a step function as the distribution function for the time it takes for the activity to complete, we will have a distribution function with a shape similar to the one that is represented in figure 1. In this paper the type of activity that we are mainly concerned with is message delivery.

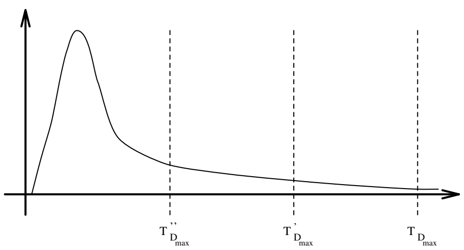

**Figure 1. Distribution function for $T_D$ (message delivery time)**

This situation poses the following problems:

- In the tentative of using the real worst-case (which may not be fully granted anyway), one is forced to choose a large value that is far from the normal case. If the communication protocol uses this worst-case to achieve some of its properties (like order, for example), this will imply a low efficiency. One possible way to try to make some optimizations, is to use an early-delivery protocol as the one we present in [1]. The basic idea is to use low cost acknowledgment messages to try to reach a decision before the worst-case time.

- Noting that the worst-case is far from the normal case, the application may want to make a tradeoff between timeliness and timing faults by adapting the assumed worst-case (see figure 1). This raises the problem of handling timing faults and being able to achieve agreement in a timely fashion. In order to solve this problem, we need an additional mechanism – a timing failure detection service. This service is used as an oracle by the communication protocols in order for them to be able to achieve safety in a bounded time. This is the main subject of this paper.

- The proposed solution to the problem addressed in the previous item, does not solve timeliness *per se*. We have safety in a timely fashion but not *full* timeliness. This is useful for some real-time applications, but does not solve all problems. The application is notified about timing faults, but it is the one that is supposed to make the final decision to handle that situation. Thus, it is desirable to have several protocols with different qualities-of-service (QoSs) that applications can select in order to adapt their needs to the environment. Aspects of these QoSs are, for example, order, agreement and timeliness. We address this subject in more detail in another document [2].

The way we address these problems has some similarities with other work. For example, the idea of using early-delivery protocols appear in other papers, and we used the work of Gopal *et al.* [8] as a starting point in [1], where we introduced some extensions in order to be able to provide early-delivery to all participants of the communication group.

The idea of using a failure detection service together with the communication protocols, that we address in the present paper [2], can also be related to work done by the Isis group [10, 3] to *solve* the problem of consensus in an asynchronous system [7]. The major difference between the two scenarios is that we want to reach agreement in a known bounded time. This puts some special requirements on the failure detection service – it must be implemented using a synchronous channel.

Another work that deals with systems that are not completely synchronous is the one described by Dwork *et al.* in [6] about *partial synchrony*. However, they are mainly concerned 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. That is why we have to pick *a priori* a given value as the bound for message delay. In [6] the bound is not known or the point in time when it starts to hold is not known.

[2] In [1], we made already some references to this problem, but in a more superficial way.

## 3. System model

We will now introduce our system model. Most of the published work about real-time assumes the existence of a *synchronous* system in the sense that there are known bounds on process speed, message transmission delay and local clock rate drift. Having a synchronous system simplifies the design and implementation of real-time applications and system support tools, as real-time communication protocols, for example. However, we want to address the problem, explained in the previous section, of having a more generic scenario where the dynamic characteristics of applications and environment do not allow, in a *realistic* way, to assume that the system is completely synchronous.

Thus, we will relax in part the synchrony of the system. We only assume as synchronous a small bandwidth communication channel for high priority messages [3]. Otherwise, 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 [4] 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.

The assumption of a channel with *better* characteristics is something that, we think, is reasonable to assume in scenarios usually available in real-time environments, and being a restricted bandwidth channel, makes it possible to obtain the desired properties (synchronism) in a larger setting. 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.

In our system, there is a *global notion of time*, i.e. local clocks are synchronized. The precision of clock synchronization is $\pi$. This global time can also be related to the real time with a given accuracy $\alpha$. We will ignore clock granularity at this point. The clock synchronization it self is not addressed in the paper. 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 communication protocols, 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 *end-point*. 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 handled with the help of a **timing failure detection service**, as explained later in the paper.

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 an *assumed* maximum message delivery time $T_{D_{max}} = max_p(T_D^p(m))$.

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

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 a timing failure detection service implemented using the small bandwidth synchronous channel. 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.

## 4. Timing failure detection service

The adaptability and configurability characteristics that we want our system to support, are achievable by the use of a Timing Failure Detection Service (TFDS). The TFDS

[3] 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.

[4] For the work presented in this paper we will consider only the relaxing of the message transmission delay.

is a key component in our system and 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.

The main idea, is to realize that we can 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. This allows the application to make progress eventually by selecting a lower quality-of-service (QoS). For the type of applications that we are targeting (dynamic applications – mission-critical), the situation of having phases of degraded service is acceptable and most of the times unavoidable. The best that can be done is to provide the means to be able to switch between QoSs in an application controlled way.

The way TFDS is used is represented in figure 2 and a sketch of the TFDS protocol is represented in figure 3. Its main function is to disseminate, in a timely fashion, vital information among nodes. This information is used by the group communication protocols to achieve safety in a bounded time. For that, each TFDS (there is one in each node) sends periodic messages using the synchronous channel. $\Pi_F$ and $\Delta_F$ are respectively the period and the latency of the TFDS.

There is no specific restriction on the type of information to be disseminated by the TFDS. The only limitation is on the available bandwidth. Otherwise, the communication protocol can specify a given amount of information that it wants all group members to receive. Anyway, besides group membership information, communication protocols may want to disseminate some information related to message activity, as for example:

- messages accepted in $[T - T_{D_{max}},\ T]$
- messages sent in $[T - T_{D_{max}},\ T]$
- messages delivered in $[T - T_{D_{max}},\ T]$
- messages received after the deadline

With this information they will be able to make a decision accordingly to the QoS they want to provide.

Communication protocols and group management can be organized in a hierarchical way making it possible to have groups with a lower overhead on join and leave operations. This can be of great importance in the handling of timing faults.

The interaction with TFDS is done using the following interface:

| |
|---|
| *configTFDS(*$\Pi_F$, $\Delta_F$*)* |
| *rid = reserveBDW(size,* $\Pi$*)* |
| *releaseBDW(rid)* |
| *info_to_send(rid, info)* |

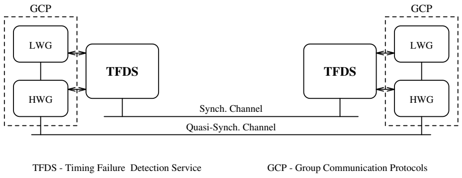


**Figure 2. Use of the Timing Failure Detection Service**

**Every** $\Pi_F$ **do**
  get info to send
  disseminate info to all members
**od**
**When** info *accepted* **do**
  update info received
  **if** info is late ($> \Pi_{F_i} + \Delta_F$) **then**
    *exception_handling*
**fi od**
**When** $current\_time = \Pi_{F_i} + \Delta_F$ **do**
  **for each** $p\ in\ MEMBERS$ **do**
    **if** info not *received* from $p$ **then do**
      run membership protocol
      eventually remove $p$ from $MEMBERS$
**od fi od od**

**Figure 3. Sketch of the TFDS protocol**

The use of *configTFDS* allows to configure the period of TFDS. Note that there is a tradeoff between the number of messages (period) and the increase on the latency of the communication protocol. Reservation of bandwidth can be done using *reserveBDW* and specifying the size and the periodicity of the information to send. The request to send information is done using *info_to_send*.

## 5. Group communication protocol

In this section we present a group communication protocol that aims to provide *total temporal order* in the type of environment represented by the described model. By *total temporal order* we mean a total order based on message timestamps obtained from synchronized clocks, with ties broken using participant identifiers. The protocol presented here is an extension of the early-delivery protocol presented in [1] and it also provides early-delivery. Instead of only using normal *ack* messages, here the services of TFDS are also used.

In order to simplify the explanation of the protocol, we will assume the existence of two basic multicast primitives – *disseminate()* and *accept()* – that are responsible for 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 not be addressed in this paper.

The basic primitives referred above (*disseminate() / accept()*), have the responsibility of dealing with omission and crash faults. We do not make any assumption about order, but, in the absence of timing faults, a message sent by a correct participant at time *T* is supposed to be accepted by all correct participants *before* $T + T_{D_{max}}$ on the recipients clock.

These basic primitives can easily be obtained from other published protocols like [11], 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 $\Delta$-protocols from Cristian *et al.* [4]. The latency $\Delta$ 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, we are interested in considering the case where the assumption about a given maximum message delivery time ($T_{D_{max}}$) might be violated. The value $T_{D_{max}}$ is determined assuming a given worst-case scenario, but, 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.

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? If there is only one participant that receives the message after the deadline, then maybe 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 and so, it can already be in an inconsistent state [9]. In that case it is also possible that it has already sent some other messages, causing a contamination problem [9]. 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.

A possible solution to this problem is 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 and can be obtained using the TFDS. Like this it is possible, not only to detect

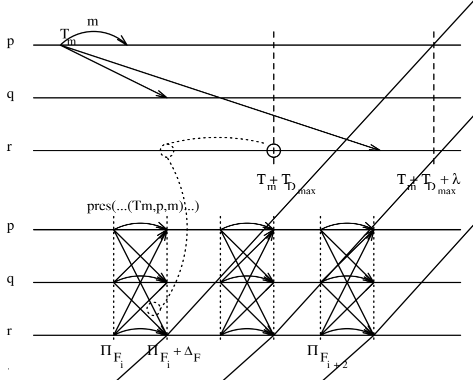


**Figure 4. Use of periodic messages to detect faults**

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 (figure 4 and figure 5). The new latency is $T_{D_{max}} + \lambda$, with $\lambda \geq \Pi_F + \Delta_F$. $\Pi_F$ and $\Delta_F$ are respectively the period and the latency of the TFDS. This increase in the latency is necessary in order to make sure that the information needed to make a decision is received.

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, $T_m + T_D$ ($T_D > T_{D_{max}}$))

An interface to obtain this information and specify configuration options, can be added to the TFDS, thus allowing to implement several different protocols with different QoSs.

## 6. Conclusions and future work

The work presented here is part of a more generic architecture that aims at providing the infrastructure to build fault-tolerant distributed real-time applications. Our target applications are mainly those with dynamic characteristics or those that run on environments that present some uncertainty (non-determinism).

**When** *p* wants to *rtcast (m)* **do**
  $T_m \leftarrow current\_time$
  insert $(T_m, p, m)$ in $SENT_p$
  $disseminate(T_m, p, m)$ **od**
**When** *p accepts* $(T_m, q, m)$
    [and $current\_time < T_m + T_{D_{max}}$] **do**
  insert $(T_m, q, m)$ in $ACCEPT_p$
  $hist_m \leftarrow SENT_p \cup ACCEPT_p$
  $send(q, ack_m, hist_m)$ **od**
Delivery:
  **if** *p received* $(ack_m, hist_m)$ from all *r*
    and $current\_time < T_m + T_{D_{max}} + \lambda$ **then do**
      obtain $sack_m$ and $shist_m$
      *send (all,* $sack_m$, $shist_m$*)*
  **od fi**
  **if** *p receives* $(sack_m, shist_m)$
    and $current\_time < T_m + T_{D_{max}} + \lambda$ **then do**
      **if** there is no $m' \prec m$ 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** $current\_time = T_m + T_{D_{max}} + \lambda$ **do**
    **for each** $(T_m, q, m)$ that *p* has *accepted*
      and not yet *delivered*
      in *lexicographic order* **do**
        **case** *check_delivery*$((T_m, q, m))$
          *TO_DELIVER: deliver* $(T_m, q, m)$
            check delivery of pending messages
          *FAULT: fault_handling*$(T_m, q, m)$
  **od fi od od**

**Figure 5. Sketch of the protocol (*send()* and *receive()* primitives, that deal with *acks*, are modified in order to also interact with TFDS)**

In this paper we propose a solution 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 allows for a graceful degradation of quality-of-service which is acceptable for many applications. The key component is a timing failure detection service implemented using a small bandwidth synchronous channel that is used to validate information sent using other more generic (not fully synchronous) communication channels.

As future work, we plan to pursue the development of the system referred above. We also plan to have other protocols with more relaxed qualities-of-service. 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, 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* group 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 timing 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 [5].

## References


[1] C. Almeida and P. Veríssimo. An adaptive real-time group communication protocol. In *Proceedings of the First IEEE Workshop on Factory Communication Systems*, Lausanne, Switzerland, Oct. 1995.

[2] C. Almeida and P. Veríssimo. Real-time group communication in *quasi-synchronous* systems. Technical Report RT/23-95, INESC, Lisboa, Portugal, Nov. 1995.

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

[4] F. Cristian, A. H., R. Strong, and D. Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *Digest of Papers, The 15th International Symposium on Fault-Tolerant Computing*, Ann Arbor-USA, June 1985. IEEE.

[5] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *24th Annual Symposium on Foundations of Computer Science*, Nov. 1983.

[6] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr. 1988.

[7] 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, Apr. 1985.

[8] A. Gopal, R. Strong, S. Toueg, and F. Cristian. Early-delivery atomic broadcast. In *Proc. of Ninth ACM Symposium on Principles of Distributed Computing*, Quebec City, Canada, August 1990.

[9] 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.

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

[11] 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, Oct. 1992. IEEE. INESC AR/66-92.