Consensus in Asynchronous Distributed Systems

A. Coccoli*, A. Bondavalli, L.Simoncini***

***University of Pisa, Inform. Eng. Dep., via Diotisalvi 2, I-56126, Pisa, Italy**
****CNUCE-CNR, via Santa Maria 26, I-56126, Pisa, Italy**
e-mail: A.Coccoli@guest.cnuce.cnr.it, {A.Bondavalli, L.Simoncini}@cnuce.cnr.it

ABSTRACT

The distributed consensus problem arises when several processes need to reach a common decision despite failures. The importance of this problem is due to its omnipresence in distributed computation: we need consensus to implement reliable communications, atomic commitment, consistency checks, resources allocations etc. The solvability of this problem is strictly related to the nature of the system it is conceived in. When an asynchronous system is considered, a research result states the impossibility of deterministically reaching consensus when even one single fault occurs. In this paper we will focus our attention on the models proposed to overcome this result and the research originated from them.

Key-words: Fault-tolerance, distributed systems, distributed consensus, failure detectors, partial synchrony, quasi-synchronous systems, timed asynchronous systems.

1. INTRODUCTION

Information sharing, availability (even if some components are not properly working) and, above all, partial mode failure semantic are some of the many advantages offered by the distributed system model. One of the drawbacks is represented by the need for complex techniques to manage redundancy and to ensure consistency of the state. Fundamental in this sense is the role of the consensus paradigm.

The problem of consensus arises each time several processors (or processes) have to reach a common decision, as a function of their initial inputs, despite failures. The importance of this problem is related to its diffused presence in the distributed systems model: we need consensus to implement reliable communications, atomic commitment, consistency checks, resources allocations etc. From a theoretical point of view, its importance is related to its equivalence with other problems like membership or atomic broadcast. This can easily explain the central role it played in the study of fault-tolerant distributed computing (Turek and Shasha, 1992, Barborak et al. 1993) .

The solutions of the consensus problem are strictly related to the nature of the system it is considered in. If it is possible to define the temporal properties of all the events of the system (i.e. if the system is *synchronous*), there is a deterministic solution for the consensus problem. If it is not possible to make any temporal assumption (the system is *asynchronous*), then the impossibility to deterministically distinguish a slow process from a stopped (crashed) one makes any solution for this

The asynchronous system model offers several advantages in terms of portability and generality of its applications, but it does not allow to provide any useful result. The synchronous model can not be implemented in practice, because it is a theoretical abstraction. For these reasons the research tried (and is still trying) to define system models with intermediate degrees of synchrony.

Among the most interesting extensions of the asynchronous model we will focus our attention on the asynchronous with *failure detectors* (Chandra and Toueg, 1996), the *partially synchronous* (Dwork et al. 1988), the *quasi-synchronous* (Verissimo and Almeida, 1995), and the *timed asynchronous* models (Cristian and Fetzer, 1998).

The failure detector model is based on a distributed oracle that gives (possibly incorrect) hints about the behaviour of the component of the system so to help to achieve consensus. The *partial synchronous* system model relaxes the requisites of synchrony for processors and/or for communications considering the cases in which the bounds on messages deliveries or on the clock drift of the processors exist but are unknown or they are known but hold after an unknown time. In the *quasi-synchronous* model the properties that define a synchronous system are supposed to have a probability $\neq 0$ not to hold.

The consideration that existing fault-tolerant services for asynchronous distributed systems are timed is at the basis of the timed asynchronous model definition. This model is less general than the asynchronous one (it assumes, for example, that all the processes have access to local hardware clocks that run within a linear envelope of real-time) but it is not a practical restriction if we consider the high standards of quality offered by the current technology.

The rest of this paper is organised as follow: after a formal definition of synchronous and asynchronous systems and a classification of failure types (section 2), section 3 will define the consensus problem and its correlation with other important problems in distributed systems and reports the Fischer, Lynch and Paterson impossibility result for asynchronous systems, we will introduce the above mentioned models proposed to overcome it. We will describe the *failure detectors* model and the most important research results originated from a seminal paper of Chandra and Toueg (1996) in section 4, the *partially synchronous* in section 5, the *quasi-synchronous* in section 6, and the *timed asynchronous* system model in section 7. Section 8 will conclude this paper including remarks on the models considered and their comparison.

2. MODEL CLASSIFICATION

Distributed system models can be classified according to what they assume about synchrony, failure model, and network topology.

Synchronous and asynchronous systems

We say a system is *synchronous* when the following properties hold:

1. There is a known upper bound on the message delivery delays;
2. There is a known upper bound on the processing speeds;
3. There is a known upper bound on local clock rate drifts.

In this kind of systems, it is possible to implement failure detection mechanisms, e.g. using some timeouts, and to achieve high degrees of reliability. The drawback is represented by the difficulty of ensuring these properties in a large-scale system and for a long time.

We say a system is *asynchronous* (or *time-free*, if we adopt the definition given in (Cristian and Fetzer, 1998)) when there is no chance to make any temporal assumption (the previous mentioned bounds do not exist or are unknown). The interest towards this model can be explained because of the high portability, robustness and generality of an application suited for it, and if we consider as cause of asynchrony a variable or an unexpected workload then the limitation of applicability of the synchronous model emerges. These two models represent the extremes of a spectrum of possibilities in which we can insert all the currently available models for the implementation of distributed applications.

Failure models

A process or a communication component is said to be *correct* if it behaves according to an agreed specification. If it is not the case, a *failure* is said to occur and the processor (or the communication component) is said to be *faulty*. A protocol is called *m-resilient* if it operates correctly despite up to *m* failures occurring during execution.

There are several widely used models of process failure. They can be classified in terms of 'severity' (Hadzilacos and Toueg, 1993). One model is less *severe* than another one if the faulty behaviour allowed by the former is a proper subset of that allowed by the latter. According to such a classification the first one is the *fail-silent* or *crash* failure model: in this model a process fails by crashing when it stops its computation and sending any message. A more severe type of failure is the *omission* failure model in which some messages from faulty processes may not reach their destinations. Then we have the "*by value*" failure model characterised by the transmission of a message different from the specified one, and, in the end, the *arbitrary* (or *Byzantine*) failure model, typical of a processor exhibiting an absolutely arbitrary behaviour (sending, for example, wrong and conflicting messages, or arbitrarily changing its state). The possibility to have an authentication mechanism (unforgeable signatures for authenticating messages like error detecting codes (Peterson and Weldon, 1972)) permits to distinguish arbitrary failures from *authenticated byzantine* failures.

When a precise temporal grid is available another failure model is introduced: the *timing* (or *performance*) failure, that is, missing the temporal bounds imposed either on execution speeds, on clock drift or on message delivery (we can distinguish between *early* or *late timing* failures according to the way these bounds are not respected).

By assuming the existence of a given failure model, we define some "boundaries" on the system behaviour. The validity of every assumption can be quantified by an associated hypothesis coverage (Powell, 1992). For example, if some error detection codes are used to authenticate messages, we can assume that no corruption of data occurs during a message transmission (a detected corrupted message can be discarded and considered as never received). The dependability estimation of applications based on this assumption has to take into account the coverage of the hypothesis made (i.e. the probability of the occurrence of a non-detectable error).

Network topology

Different assumptions can be made on the network topology: it can be either fully or partially connected, it can have point-to-point links or broadcast capabilities, the delivery of messages can have an arbitrary order or a FIFO order.

3. THE CONSENSUS PROBLEM

The consensus problem arises each time several processes have to reach a common decision (depending on their initial inputs) despite failures (Fischer et al, 1985).

Let's consider a system of n ($n \geq 2$) processes that communicate via messages. When a consensus is needed, each process proposes its value and the correct processes, following a deterministic protocol involving the receipt and sending of messages, have to irreversibly decide on a common value among the proposed ones.

In order to solve the consensus problem the protocol has to satisfy the following *safety* and *progress* properties:

Termination: every correct process eventually decides some value (*progress*);

Agreement: No two correct processes decide differently (*safety*);

Uniform integrity: every process decides at most once (*safety*);

Uniform validity: if a process decides v, then v was proposed by some process (*safety*).

Different forms of Consensus can be obtained modifying the above properties. A stronger form of Consensus can be defined imposing the **Uniform Agreement** property, that is assuming that no two processes (correct or not) decide differently (Neiger and Toueg, 1990). This is the so-called Uniform Consensus Problem.

In an asynchronous system the progress property requires the eventual reaching of consensus (*liveness* property), while in a synchronous system this property requires the consensus to be reached in a bounded time (*timeliness* property).

The concept of consensus as a support for the consistency of correct processes is at the basis of the imple-

mentation of *responsive*[1] systems, that combine fault-tolerant systems requirements with real-time systems ones (Malek, 1995). Another fundamental problem of computing systems, the *System Diagnosis* problem is closely related to Consensus: they both require the correct processes population to be handled in order to behave in a specified and consistent way (the former using a diagnosis mechanism, the latter masking the faulty processes) (Barborak et al, 1993). The generality of consensus can be made evident by its close relation (Fischer, 1983) with two other problems: the *Byzantine Generals* (Dolev, 1982; Lamport et al, 1982) and the *Interactive Consistency* problem (Fischer and Lynch, 1982). In the former, a transmitter process sends an initial value to the other processes, the receivers, which must agree on the value of the transmitter, in the latter, each process sends a private value to every other process, and then each correct process must infer a vector on values with an element for each of the processes (including itself). The equivalence with the *Atomic Broadcast* problem is another evidence of the importance of the consensus problem. Formally, the a*tomic broadcast* is a broadcast that satisfies the following conditions (Hadzilacos and Toueg, 1993):

1. **Validity**: if a correct process broadcasts a message *m*, then all correct processes eventually deliver *m* (progress);
2. **Agreement**: if a correct process delivers a message *m*, then all correct processes eventually deliver *m* (progress);
3. **Uniform integrity**: for any message *m*, every correct process delivers *m* at most once, and only if *m* was previously broadcast by some process (safety).
4. **Total order**: if correct processes *p* and *q* both deliver messages *m* and *m'*, then *p* delivers *m* before *m'* iff *q* delivers *m* before *m'*

A demonstration of the equivalence between the problem of Atomic Broadcast and the Consensus can be found in (Chandra and Toueg, 1996). Equivalence means that any solution to one problem can be used to solve the other problem and vice-versa (this is a process of reduction of one problem to the other). The specifications and some implementations of other broadcast model can be found in (Hadzilacos and Toueg, 1993). The close relationship between solutions to agreement problems and broadcast capabilities can be found in applications like the Advanced Automation System (Cristian et al., 1990), Amoeba (Tanenbaum et al., 1990), Delta-4 (Powell, 1991), Horus (van Renesse et al., 1996), Newtop (Macedo et al., 1993), Isis (Birman and van Renesse, 1994), Transis (Amir et al., 1992).

The consensus problem is important because of a fundamental theoretical result based on the analysis of fault-tolerant distributed computation: the impossibility theorem of Fischer, Lynch and Paterson (FLP) (Fischer et al., 1985). This theorem states that no deterministic solution exists for the consensus problem in an asynchronous system where even a single crash failure may occur. This result holds also in the presence of a completely connected and reliable communication network where a sent message is eventually received. The core of this result is based on the intrinsic impossibility to distinguish, in an asynchronous system, a failed (crashed) process from a merely slow one. The demonstration is obtained by contradiction assuming that such a protocol exists and showing the existence of at least one initial configuration from which an infinite computation is not enough to make the protocol able to choose one value (contradicting its assumed correctness). The theoretical and practical implications of this result are very important. Whereas the synchronous model is a mere theoretical abstraction, violated by real systems, the FLP impossibility result tells that the asynchronous model is too general and does not allow to provide any useful property or result. This has led several researchers to consider intermediate models between these two extremes, either weakening the assumptions of the synchronous model, or constraining the asynchronous one. In the following sections we will see the most interesting models proposed in this direction.

5. FAILURE DETECTORS

In order to overcome Fisher et al's result, Chandra and Toueg proposed to extend the asynchronous system model introducing the concept of *Unreliable Failure Detector* (Chandra and Toueg, 1996). In their work, the system consists of a set of *n* processes, every pair of processes is connected by a reliable communication channel, and the processes can fail by *crashing*. A failure detector can be conceived as a distributed oracle that gives (possibly incorrect) hints about which process may have crashed so far. It is constituted by several modules local to each process periodically consulted by the corresponding process. Each module produces a list of processes suspected to be crashed, and this information is used by processes in order to achieve the consensus although it is important to note that a suspected process continues to behave according to its specification. The modules are intrinsically unreliable: they can make mistakes, so the lists dynamically change during the computation (and it is possible for two or more lists to be different at the same time).

The failure detectors can be classified according to their **accuracy** and **completeness** properties. The accuracy property restricts the mistakes a failure detector can make, while completeness represents the capacity of suspecting an actually crashed process. More precisely, we can define two completeness properties:

- **Strong Completeness**: eventually every crashed process is permanently suspected by every correct process;
- **Weak Completeness**: eventually every crashed process is permanently suspected by some correct process;

and four accuracy properties:

- **Strong Accuracy**: No process is suspected before it crashes;
- **Weak Accuracy**: some correct process is never suspected;

[1]Responsiveness is defined as the probability of a timely and correct execution under a given fault and load

- **Eventual Strong Accuracy**: there is a time after which correct processes are not suspected by any correct process;
- **Eventual Weak Accuracy**: there is a time after which some correct process is never suspected by any correct process;

Combining the above mentioned properties we obtain the eight classes of failure detectors depicted in table 1 (Chandra and Toueg, 1996).

The classes of failure detectors are not independent: using a reduction algorithm (roughly speaking: an emulator) it is possible to demonstrate the existence of equivalence relations. More precisely, a reduction algorithm $T_{D \to D'}$, is an algorithm that permits to solve any problem that can be solved with D', using D in its place. If an algorithm $T_{D \to D'}$ exists, we write $D \geq D'$ and say that D' is reducibile to D (or that D' is weaker than D). The reducibility attribute is a transitive relation, and if $D \geq D'$ and $D' \geq D$ we say that D and D' are equivalent. If $C \geq C'$ and C is not equivalent to C', then C' is said to be strictly weaker than C ($C > C'$). Chandra and Toueg (1996) showed the existence of reduction algorithms that transform any given failure detector that satisfies Weak Completeness, into a failure detector that satisfies Strong Completeness. These relations, as well as other reducibility relations between the classes of failure detectors, are described in figure 1.

Chandra and Toueg (1996) showed the solvability of the consensus problem in an asynchronous system where a failure detector is given: if eventual accuracy is provided ($\Diamond P$, $\Diamond Q$, $\Diamond S$, $\Diamond W$) a majority of correct processes is needed; if we provide perpetual accuracy (classes P, Q, S, W) then there are no limits on the tolerated failures. In the same work two algorithms for consensus are described: one based on a Strong Failure Detector and the other on an Eventually Strong Failure Detector. An interesting result deriving from the equivalence relations between the failure detectors classes is the solvability of consensus with an Eventually Weak failure detector (with a majority of correct processes) (Chandra et al., 1996). This result is important because, being the weakest failure detector that can be used to solve consensus, it states necessary and sufficient conditions for the solution of the problem.

Chandra and Toueg's work is essentially based on abstract definitions of the model of failure detectors, though this work triggered a lot of other studies: in (Aguilera and Toueg, 1996) an integration with a non-deterministic model of the system is proposed, while in (Dolev et al., 1997) the attention is focused on systems with omission failures. Guerraoui and Schiper (1996) redefine the failure detector formalism for systems where network partitions can occur. The recovery of crashed processes is considered in (Hurfin et al, 1997; Oliveira et al., 1997; Aguilera et al., 1998). Lo and Hadzilacos (1994) considered the use of failure detectors in shared-memory systems.

6. PARTIAL SYNCHRONOUS SYSTEM.

Analysing the FLP impossibility result, there appears to be three different types of asynchrony:

Process asynchrony: a process may "go to sleep" for arbitrarily long finite amounts of time while other processes continue to run;

Communication asynchrony: no a priori bounds exist on message delivery time;

Message order asynchrony: messages can be delivered in a different order from the one in which they were sent.

Investigating on the influences of the different types of asynchrony, Dolev et al. (1987) found that it is not necessary to have all the above types of asynchrony to obtain the impossibility result. More precisely, they considered five critical parameters and the way their attributes contribute to solve the consensus. They considered the synchrony/asynchrony of processors (1) and of communications (2), existence/absence of a correspondence on the order messages are sent and received (3), the transmission mechanism (point to point or broadcast) (4), and the possibility/impossibility to atomically execute send and receive (5). Combining all these five parameters, a space of 32 possible system models can be obtained. Between them, four cases have been identified in which *n*-crash resilient protocols exist. These cases are necessary in the sense that it is enough to change one parameter from favourable to unfavourable to lose the possibility of having a *m*-crash resilient protocol (*m* being 1 or 2). These minimal cases are:

1. synchronous processors and synchronous communication;
2. synchronous processors and synchronous message order;
3. broadcast transmission and synchronous message order;
4. synchronous communication, broadcast transmission and atomic send/receive.

In the case of bounded message delivery time, atomic send/receive and point-to-point transmission, we can get 1-resiliency but not 2: in a critical step, a process *p* sends a message to another process *q*, in order to hide this event it is necessary and sufficient that both *p* and *q* fail. Figure 2 shows the maximum resiliency for each setting of the five parameters (empty table entries represent the unsolvability of consensus even in the presence of one crash failure).

All the above considerations constituted the basis for the definition of the **partially synchronous** system model (Dwork et al., 1988). This model is originated relaxing the requisite of synchrony for processors and/or for communications. According to this work, calling Δ and Φ the upper bounds on message transmission and on relative clock speeds of processors respectively, a partial synchrony can be caused by two conditions:

-the bounds exists but are not known;
-the bounds are known, but they hold after some unknown time.

In the former case, the system is de facto synchronous, so the impossibility result does not hold. The problem is to manage the messages exchange without the knowledge of the real values of Δ and Φ: using non correct values for these bounds will obviously affect the protocol correctness or performance. In the latter case an instant of time, called Global Stabilisation Time (GST), is supposed to exist such that the bounds are valid from

GST on. The same situations can be observed considering only one bound at a time. For these models table 2 shows the maximum resiliency for different forms of synchrony and types of failures. It is interesting to observe that the usage of authentication mechanisms does not improve resiliency of systems with partially synchronous communications.

7. QUASI-SYNCHRONISM

The respect of all the timing constraints is mandatory when life-critical applications are considered, however, there are other real-time applications where, despite the need for dependability, it is acceptable to eventually miss some of them (assuming to achieve the most important ones). This is the consideration at the basis of the definition of the **quasi-synchronous** model proposed by Verissimo and Almeida (1996b).

It should be observed that, according to the authors' approach, the systems in which these applications are built are not soft real-time in the strict sense of the word: they are designed on the assumption of a non negligible probability of timing failures occurrences, for this reason, adequate timing fault-tolerance measures are needed.

Using Almeida and Verissimo wording, a system is synchronous if there are

P1. bounded and known processing speeds;
P2. bounded and known message delivery delays;
P3. bounded and known local clock rate drift;
P4. bounded and known load patterns;
P5. bounded and known difference among local clocks.

A system is quasi-synchronous if

D1. it can be defined by properties *Px*;
D2. There is at least one bound where there is a known probability ($\neq 0$) that the bound assumption does not hold (this probability is called *assumption uncoverage*);
D3. any property can be defined in terms of a series of pairs (bound, assumption uncoverage).

A Quasi-Synchronous system can be conceived as a synchronous system in which the absolute bounds on messages transmission delays, local clock drift rates and process execution times are so far away from those observed during the normal operative mode that it is more convenient to use different values, even if the coverage of such assumptions is not equal to one (Powell, 1992). This model allows to catch both the situation in which the bounds do not exist and that in which the bounds exist, but some or all of them are too far from the normal case, so it is preferable to use shorter artificial bounds, with an uncoverage probability. In (Almeida and Verissimo, 1998) the integration of a timing failure detector (implementable using a dedicated small bandwidth synchronous channel, or using highest priority messages) with an hierarchical structure for group management is used to efficiently handle timing failures and to obtain a real-time group communication in a Quasi-Synchronous system. A more detailed description of the above mentioned concepts can be found in (Almeida and Verissimo, 1995; Almeida and Verissimo, 1996b). An interesting application of this architecture [illegible]

time data-base, where the consistency of replicas has to be ensured without influencing timeliness and vice-versa (Almeida and Verissimo, 1996a). The conjunction of a failure detection service with the communication protocols can be related to the work done by the ISIS group to solve the consensus problem in asynchronous systems (Ricciardi and Birman, 1991).

8. TIMED ASYNCHRONOUS SYSTEM

At the basis of the timed asynchronous model definition there is the consideration that existing fault-tolerant services for asynchronous distributed systems are timed: the specification of the services not only describes the states transitions and the outputs in response to invocations of operations, but also the time interval within which these transitions have to complete (Cristian and Fetzer, 1998).

The timed asynchronous system model is characterised by a set of assumptions on the behaviour of processes, of communications and of hardware clocks:

1. all the services are timed (the temporal characteristics of the events are specified), so it is possible to associate some time-outs whose expiration produces a performance failure
2. communication between processes is obtained via an unreliable datagram service with omission and/or performance failure semantics;
3. processes have crash/performance failure semantics (Cristian, 1991);
4. all processes have access to private hardware clocks that run within a linear envelope of real-time;
5. no bound exists on the rate of communication and process failures.

As we can see, the timed asynchronous system model is asynchronous in the sense that it does not require the existence of upper bounds for message transmissions and scheduling delays. However, the access to local hardware clocks and the definition of time-outs allow us to define the performance failure as that failure which occurs when an experienced delay is greater than the associated time-out delay. It could be argued that the timed model is less general than the asynchronous one, but this is not true from a practical point of view. In fact, each time an higher level of abstraction depends on a service, it makes the service to become "de facto" timed. Also the presence of clocks is not a practical restriction if we consider that the currently technology makes available high-precision quartz clocks. Differently from the asynchronous model, the timed model allows to implement fundamental services like clock synchronisation, membership, consensus, election and atomic broadcast (Cristian, 1989; Cristian and Schmuck, 1995; Fetzer and Cristian, 1995; Cristian, 1996).

Let's analyse more in detail the model, starting from the support for the communication.

The datagram service

The following assumptions hold:

- there are no assumptions on the physical network topology;
- messages are transmitted either via unicast or broadcast;

[illegible]

- there is no upper bound on the message transmission delay;
- it permits the definition of a time-out on the message transmission (one-way time-out delay δ) whose value influences the failures rate and, consequently, the system stability;
- it transmits the messages with a time proportional to their dimension;
- it has a crash/omission failure semantics (the possibility of message corruption is negligible).

The one-way time-out delay associated to the communication services permits to say that a message is received in a *timely manner* if its transmission delay is not greater than δ.

The processes

All the processes have access to a private stable storage so it is possible, for them, to be recovered after a crash occurred. In particular, a process can be in one of the following three states: *up*, *crashed*, and *recovering*. A process is in the *crashed* state when a stops executing its code and has lost all its previous state. It is in the *recovering* state when executes its state 'initialisation code', i.e. after its creation, or in the restarting phase after a crash occurred. When it executes its 'standard' code, it is in the *up* state.

The non-crashed processes change their state in correspondence of triggering events like time-outs or message delivery. The time between the occurrence of an event and the moment the process finishes the processing of this event is called *process scheduling delay*. For each process *p* a time-out σ for scheduling delays is given. When σ expires the process is said to have suffered a *performance failure*.

Local Clocks

Processes have access to a local hardware clock whose drift rate is bounded by a calibration mechanism. The local clocks are subjected to crashes and this event produces a crash in the processes that rely on them (the opposite is not true: a process crash does not affect its local clock).

The Progress Assumptions

The timed asynchronous system model is extended by the so-called *progress assumptions*. In short, the progress assumptions are introduced to characterise the synchronous behaviour exhibited by a system, i.e. to model the additional synchronism into the timed asynchronous system model (e.g. progress assumptions can be used to characterise the behaviour of distributed systems based on LANs which alternates between long stability periods and comparatively short instability intervals). Progress assumptions can be expressed in the following way: infinitely often a majority of processes will be stable for a bounded amount of time. In order to formally define the progress assumptions we need some definitions. A process *p* is *timely* in an interval [*s*, *t*], if it is non-crashed and it does not suffer any performance failure in [*s*, *t*]. Two processes are *connected* in [*s*, *t*] if they are timely during [*s*, *t*] and each message exchanged between them in [*s*, *t*-δ] is received in a *timely* manner, i.e. the message transmission delays of connected processes are at most δ. A *stable majority* in an interval [*s*, *t*] is composed by a majority of up, timely and pair-wise connected processes in [*s*, *t*]. A process is *majority-stable* when it belongs to a stable majority. A system is said *majority-stable* during an interval if there exists a stable majority during that interval, otherwise it is said *unstable* in that interval.

In the specifications of the protocols for timed asynchronous system some stability predicates are defined in order to verify the favourable conditions of the system (i.e. its synchronous behaviour). Here we will consider the *always eventually majority-stable* predicate:

1. after each unstability period, a system eventually becomes majority-stable for at least D clock time units, where D is an a priori given constant;
2. each process always eventually becomes majority-stable for at least D clock-time units or it crashes.

In (Fetzer and Cristian, 1995), a majority stability predicate is defined such that, if the system is majority stable for sufficiently long time, a solution to the leader election and consensus are deterministically implementable. It should be noted that the termination conditions for asynchronous systems require an algorithm to complete in a finite number of steps, while for synchronous systems this condition is time-bounded, i.e. the algorithm has to complete in a bounded amount of time. In timed asynchronous systems the termination conditions are conditionally-timed: in an always eventually majority stable system, if a process makes part of a majority of timely processes in an interval of time, then, each operation started at the beginning of that interval has to complete within it. Supposing to have an eventually majority system, the protocol described in (Fetzer and Cristian, 1995) solves the consensus problem using a rotating leadership service which gives each timely process the chance to be leader for a limited amount of time before it is demoted and the leadership is passed on to another process.

In this way, even in the case the leader suffers a crash, if the system is majority stable, another process can act as a leader in a bounded amount of time. This protocol solves the consensus problem despite the failures and recoveries that may affect the processes that do not belong to a stable majority. The solvability of the consensus problem in the timed asynchronous system model is strictly related to the possibility of associating an information to the time: it allows the timed asynchronous system model to be the support for the definition of the so called *Fail-Awareness*, (Fetzer and Cristian, 1996b; Fetzer and Cristian, 1997), that is, a design concept, based on the translation of performance failures in exception occurrences, for weakening the specification of a synchronous service S to a new specification FS, so that it becomes implementable in timed asynchronous systems. An interesting application of the timed asynchronous model can be found in (Essame', 1998; Essame' et al. 1999), where it is applied to the definition of a protocol (called PADRE) for the management of an asymmetric duplex redundancy in a fully automated train control system.

9. A COMPARISON OF MODELS

All the models we considered tried to overcome the FLP impossibility result strengthening the asynchro-

amount of "synchrony" to the system in order to allow the solvability of the consensus problem. Obviously, there are several differences between all the models, because the assumptions they rely upon are different.

A first comparison can be made between the partially synchronous system and the timed asynchronous system.

As we saw before, the former model assumes the existence of a bound on processor speeds and on the clock drift rate with a coverage equal to 1. On the contrary, the timed asynchronous system model only assumes a bound on clock drift rate (with a coverage equal to 1) and makes no assumptions on load patterns and on messages transmissions delays. The timed asynchronous system model has some points in common with the partially synchronous system model: the *Global Stabilisation Time* of the partially synchronous system model reminds the concept of *stability* of the timed asynchronous system, but, while the Global Stabilisation requires that the system is not affected by any (timing or crash) failure after a certain instant, the stability of a timed asynchronous system is valid only for bounded time intervals. Moreover, it is interesting to compare the partial synchrony system models with the notion of an *unreliable failure detector*. For every partial synchrony model we considered, it is easy to implement an eventual perfect failure detector (a failure detector that satisfies strong completeness and eventual strong accuracy). In fact one could implement such a failure detector with an even weaker model of partial synchrony: one in which the bounds on message transmission delays and on process speeds exist, but they are not known and they hold only after some unknown time. More difficult is to compare the failure detectors model with the timed asynchronous system one. In (Fetzer and Cristian, 1996a) the impossibility to implement a perfect failure detector in a timed asynchronous system has been proved. The main difference between the two models is in the philosophy of the design of the system. Failure detectors hide to higher abstraction levels all the aspects related to the time of the fault-tolerant distributed computation. This can constitute a problem if the abstraction levels are more than two. In such a situation all the time-outs are used in each level because a level that depend on another one (Cristian, 1991) has to be able to detect its failures. For this reason the meaning of time-outs change in correspondence of the levels they are associated to (usually the higher is the level, the greater is the time-out and the more severe is its violation). Let's consider now the differences between the Partial Synchronous and the Quasi-Synchronous model: in the latter model it is necessary to define an a priori given bound for message delay (although the bound does not hold with probability 1), while in the former, this bound is not known or it holds after an unknown instant. The last comparison is between the Timed Asynchronous and the Quasi-Synchronous model. These two models have several points in common. First of all they are both based on a careful observation of existing systems. They both are concerned on the realisation of a support for the development of real-time applications on not fully synchronous environments. Moreover, they both allow to build applications with a fail-safe shutdown. At a first glance the degree of synchrony of the Quasi-Synchronous model seems higher than the degree of synchrony of the Timed Asynchronous one, but we have to remind how the latter depends on the definition of the progress assumptions. Though, as stated in (Almeida and Verissimo, 1998), "the Quasi-Synchronous model provides more flexibility for system reconfiguration because, thanks to the timely control information provided by the Timing Failure Detector Service, an agreement can be reached before making a decision about the way that reconfiguration should be done".

REFERENCES

Aguilera, M. K., et al. (1998). "Failure Detection and Consensus in the Crash-Recovery Model". *Proc. of 12th Int. Symp. on Distributed Computing*, Andros, Greece.

Aguilera, M. K. and S. Toueg (1996). "Randomization and Failure Detection: a Hybrid Approach to Solve Consensus". *Proc of 10th Int. Workshop on Distr. Algorithms*, Bologna, Italy, pp. 29-39.

Almeida, C. and P. Verissimo (1995). "An Adaptive Real-Time Group Communication Protocol". *Proc. of First IEEE Workshop on Factory Communication Systems*, Leysin, Switzerland.

Almeida, C. and P. Verissimo (1996a). "The Quasi-Synchronous Approach to Distributed Real-Time Data-Bases", INESC tech rep RT/02-96. Lisboa, Portugal.

Almeida, C. and P. Verissimo (1996b). "Timing Failure Detection and Real-time Group Communication in Quasi-Synchronous Systems". *Proc. of 8th Euromicro Workshop on Real-Time Systems*, L'Aquila, Italy.

Almeida, C. and P. Verissimo (1998). "Using Light-Weight Groups to handle Timing Failures in *Quasi-Synchronous* systems". *Proc. of 19th IEEE Real-Time System Symposium*, Madrid, Spain.

Amir, Y., et al. (1992). "Transis: a Communication Sub-system for High Availability". *Proc. of 22nd Annual Int. Symp. on Fault-Tolerant Computing*. pp. 76-84.

Barborak, M., et al. (1993). "The Consensus Problem in Fault-Tolerant Computing." *ACM Computing Surveys* Vol. 25, pp. 171-220.

Birman, K. P. and R. van Renesse (1994). "Reliable Distributed Computing with the Isis Toolkit". IEEE Computer Society Press.

Chandra, T. et al. (1996). "The Weakest Failure Detector for Solving Consensus." *Journal of the ACM*, Vol. 43, pp. 685-722.

Chandra, T. and S. Toueg (1996). "Unreliable Failure Detectors for Reliable Distributed Systems." *Journal of the ACM*, Vol. 43(2), pp. 225-267.

Cristian, F. (1989). "Probabilistic Clock Synchronisation." *Distributed Computing*, Vol. 3, pp. 146-158.

Cristian, F. (1991). "Understanding Fault-tolerant Distributed System." *Comm. of ACM*,Vol.34, pp.56-78.

Cristian, F. (1996). "Group, Majority, and Strict Agreement in Timed Asynchronous Distributed Systems". *Proc. of 26th Int. Symposium on Fault-Tolerant Computing*, Sendai, Japan.

Cristian, F. et al. (1990). "Faut-Tolerance in the Advanced Automation System". *Proc of 20th Int. Conf. on*

Cristian, F. and C. Fetzer (1998). "The Timed Asynchronous Distributed System Model". *Proc. of 28th Int. Symp. On Fault-Tolerant Computing* (FTCS-28), Munich, Germany. pp. 140-149.

Cristian, F. and F. Schmuck (1995). "Agreeing on Processor-Group Membership in Asynchronous Distributed Systems", UCSD technical report.

Dolev, D. (1982). "The Byzantine Generals Strike Again." *Journal of Algorithms*, Vol. 3, pp. 14-30.

Dolev, D., et al. (1987). "On the Minimal Synchronism Needed for Distributed Consensus." *Journal of the ACM*, Vol. 34, pp. 77-97.

Dolev, D. et al. (1997). "Failure Detectors in Omission Failure Environments". 16th ACM Symp. on Principles of Distributed Computing (Short Paper).

Dwork, C., et al. (1988). "Consensus in the Presence of Partial Synchrony." *J. of ACM*, Vol.35,pp. 288-323.

Essame', D. (1998). "Fault Tolerance in Critical Systems: Application to Automatic Subway Control". Tolouse, Doctoral thesis, LAAS-CNRS, Tolouse.

Essame', D., et al. (1999). "PADRE: A Protocol for Asymmetric Duplex Redundancy". *Proc, of 7th Int. Working Conf. On Dependable Computing for Critical Applications*, San Jose, CA. pp. 213-232.

Fetzer, C. and F. Cristian (1995). "On the Possibility of Consensus in Asynchronous Systems". Pacific Rim Int. Symp. on Fault-tolerant Syst., Newport Beach CA.

Fetzer, C. and F. Cristian (1996a). "Fail-aware Failure Detectors". *Proc. of 15th Symposium on Reliable Distributed Systems*, Niagara on the Lake, Canada.

Fetzer, C. and F. Cristian (1996b). "Fail-awareness in Timed Asynchronous Systems", T.Rep.CS95-45, UCSD

Fetzer, C. and F. Cristian (1997). "A Fail-aware Datagram Service". 2nd Annual Workshop on Fault-tolerant Parallel and Distributed Systems, Geneva, Switzerland.

Fischer, M., et al. (1985). "Impossibility of Distributed Consensus with One Faulty Process." *Journal of ACM*, Vol. 32, pp. 374-382.

Fischer, M. J. (1983). "The Consensus Problem in Unreliable Distributed Systems (A Brief Survey)". International Conference on Foundations of Computations Theory. Borgholm, Sweden. pp. 127-140.

Fischer, M. J. and N. A. Lynch (1982). "A Lower Bound on the Time to Assure Interactive Consistency." *Information Processing Letters*, Vol. 14, pp. 183-186.

Guerraoui, R. and A. Schiper (1996). ""Gamma-Accurate" Failure Detectors". *Proc. of 10th Intern. Workshop on Distr. Algorithms*, Bologna, Italy. pp. 269-286.

FIGURES AND TABLES

Table 1 - Failure Detectors Classes

Completeness	Accuracy			
	Strong	**Weak**	**Eventual Strong**	**Eventual Weak**
Strong	Perfect P	Strong S	Eventually Perfect ◊P	Eventually Strong ◊S
Weak	Q	Weak	◊Q	Eventually

Hadzilacos, V. and S. Toueg (1993). "Fault-tolerant Broadcasts and Related Problems". *Distributed Systems*. S. J. Mullender. Reading, Addison-Wesley. pp. 97-145.

Hurfin, M., et al. (1997). "Consensus in Asynchronous Systems where Processes can Crash and Recover", IRISA Internal report PI-1144.

Lamport, L., et al. (1982). "The Byzantyne Generals Problem." *ACM Transactions on Programming Languages and Systems*, Vol. 4, pp. 382-401.

Lo, W. K. and V. Hadzilacos (1994). "Using Failure Detectors to Solve Consensus in Asynchronous Shared-Memory Systems". *Proc of 8th International Workshop on Distributed Algorithms*. pp. 280-295.

Macedo, R. A., et al. (1993). "Newtop: a Total Order Multicast Protocol Using Causal Blocks", BROADCAST Project Technical Report n.10

Malek, M. (1995). "Omniscence, Consensus, Autonomy: Three Tempting Roads to Responsiveness". *Proc of 14th IEEE Symposium on Reliable Distributed Systems*, Bad Neuenahr, Germany. pp. 12-14.

Neiger, G. and S. Toueg (1990). "Automatically Increasing the Fault-tolerance of Distributed Algorithms." *Journal of Algorithms*, Vol. 11, pp. 374-419.

Oliveira, R., et al. (1997). "Consensus in the Crash-Recover Model", EPFL, Dept. d'Informatique, Tech. rep. 97-239, Lousanne (Switzerland).

Peterson, W. and E. Weldon (1972). "Error Correction Codes". MIT Press, Massachusetts.

Powell, D. (1991). "Delta-4: A Generic Architecture for Dependable Distributed Computing" New York.

Powell, D. (1992). "Failure Mode Assumptions and Assumption Coverage". *Proc. of 22nd Int. Conf. on Fault-Tolerant Computing*., Boston, pp. 386-395.

Ricciardi, A. M. and K. P. Birman (1991). "Using Process Groups to Implement Failure Detection in Asynchronous Environments", Tech. Rep., Cornell University, Dep. of Computer Science.

Tanenbaum, A. S., et al. (1990). "Experiences with the Amoeba Distributed Operating System." *Communication of the ACM*, Vol. 33, pp. 46-63.

Turek, J. and D. Shasha (1992). "The many Faces of Consensus in Distributed Systems." IEEE Computer Vol. 25, pp. 8-17.

Van Renesse, R., et al. (1996). "Horus: a Flexible Group Communication System." *Comm. of the ACM*, Vol. 39.

Verissimo, P. and C. Almeida (1995). "Quasi-Synchronism: a Step Away from the Traditional Fault-Tolerant Real-Time System Models." IEEE TCOS Bulletin Vol. 7, pp. 35-39.

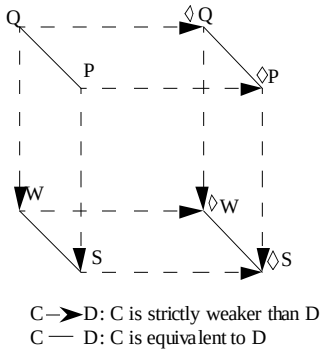

Fig. 1 Relations between Failure Detector classes (Chandra and Toueg, 1996)

Table 2. Smallest number of processors N_{min} for which a *t*-resilient Consensus protocol exists (Dwork et al. 1988) .

Failure type	System Model				
	Syn-chronous	Asyin-chronous	Partially synchr. communication and synchr. processors	Partially synchr. communication and processors	Partially synchr. processors and synchr. communication
Crash	t	∞	$2t + 1$	$2t + 1$	$t+1$
Omission	t	∞	$2t + 1$	$2t + 1$	$[2t, 2t + 1]$
Byzantine (with authentication)	t	∞	$3t + 1$	$3t + 1$	$2t + 1$
Byzantine (without authentication)	$3t + 1$	∞	$3t + 1$	$3t + 1$	$3t + 1$

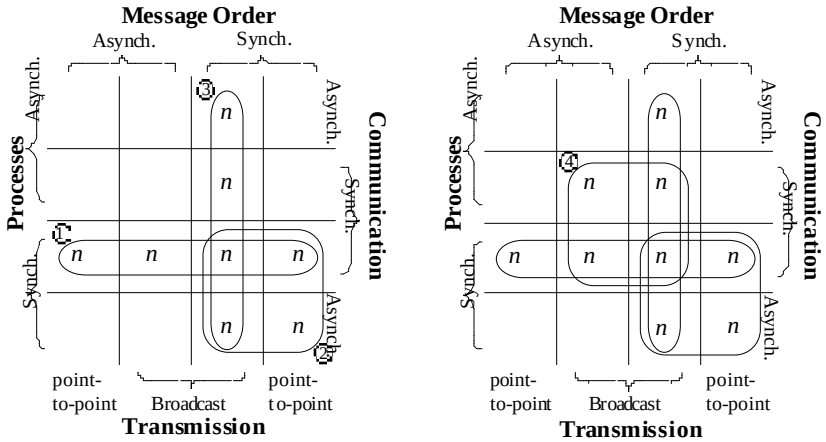

Fig. 2 Maximum crash resiliencies for consensus protocols