# The Timed Asynchronous model

- At the basis of the Timed Asynchronous model (proposed by Cristian and Fetzer) there is the observation that existing fault -tolerant services for asynchronous systems are anyway TIMED.

- The specification of the services offered by these systems describes not only the state transitions and the outputs in response to requests for operations but ALSO the time interval in which such transitions must happen!!.

- *F. Cristian, and C. Fetzer, "The timed Asynchronous Distributed System Model," 28th Intern. Symp. On Fault-tolerant Computing (FCTS-28), (Munich, Germany), pp. 140-149, IEEE Computer Society Press, 1998.*

# General Description

- The model makes a set of assumptions on the behavior of the processes of the communications and on the hw clocks and is characterized as follows: :
  - 1. All services are TIMED (all the timing characteristics of the events are specified). It is therefore possible for them to define time-outs whose passing determines a time failure;
  - 2. communications between processes is realized through a DATAGRAM service – non reliable and subject to crash and timing failures;
  - 3. processes are subject to crash o timing failures;
  - 4. processes have access to local clocks which stay within a linear envelope of real time; (means clock drift are limited)
  - 5. There is no limit on the failure rate of communications and of processes

# Comparison with the time-free model

- Applications realized for an asynchronous system consider a complete absence of a time reference and the existence of a reliable service for communications.
- Considering the all hw available today have highly precise quartz clocks it is easy to understand that the postulated existence of a local clock it is not a practical restriction.
- Moreover, despite many services available in practice (such as UDP or UNIX/LINUX processes) do not offer timing guarantees, it is true that all these service become TIMED when an higher abstraction layer depending on them defines a time-out to determine their failure.

# Il datagram service

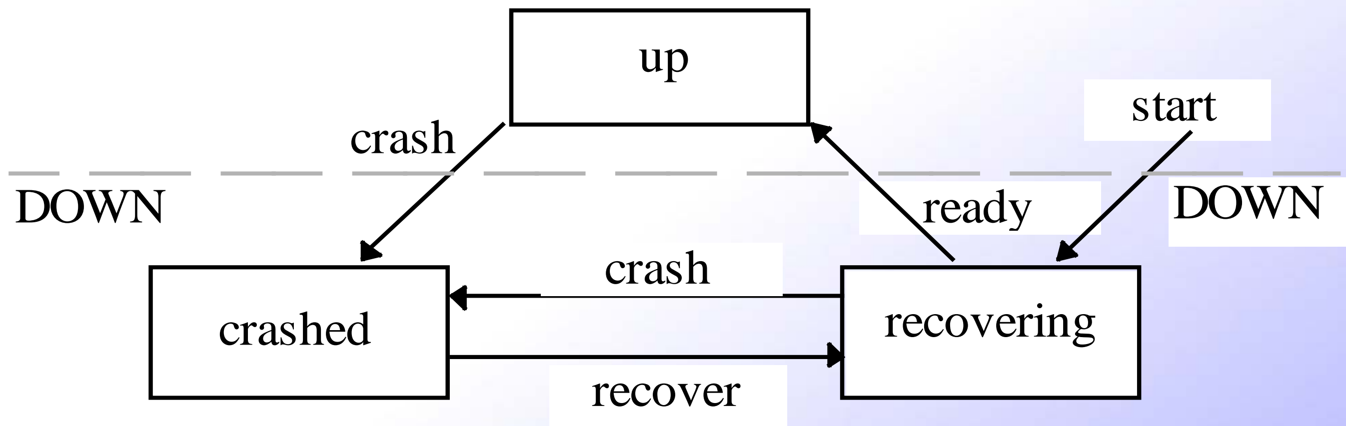
- The datagram service is characterized as follows:
  - 1. It allows unicast and broadcast;
  - 2. It identifies messages in an univocal way;
  - 3. it does not ensure the existence of an upper bound on message delivery delay (it is asynchronous!);
  - 4. It allows to define a time-out $\delta$ on message transmission (one-way time-out delay) whose choice has an impact on failure rates and on system stability;
  - 5. transmission time of messages is proportional to their size;
  - 6. It is subject to omission and timing failures only as the probability that a corrupt message is delivered is considered negligible.

# processes (1)

- Processes that did not suffer from a crach have state transitions in response to events such as message receipt or time-out expiry.
- The time interval between the occurrence of an event and the termination of its processing is called ***process scheduling delay***.
- Let $\sigma$ be the time-out for the scheduling delays. If a process p reacts to each event witin $\sigma$ time unit then the process is said to be ***timely*** otherwise it suffered from a timing failure.
- The choice of $\delta$ is (normally) such that we can neglect $\sigma$.

# processes (2)

- Each process can be in one of the following states:
  - Up: while it is executing standard program code;
  - Crashed: when it stops to execute its code and looses all its state;
  - Recovering: when it executes state initialization code following a) its creation b) restart after crash.



# Hardware clocks

- Each process has access to a local hardware clock which can deviate from real time.
- The drift of a correct hardware clock corretto is limited by ρ (*maximum hardware clock drift rate*).
- The quartz clock granularity offered by the current technology typically ranges between1 ns and 1 μs while the clock drift rate ρ is in the range $10^{-4} \div 10^{-6}$.
- It is assumed that through a calibration mechanism of local clocks they stay within a linear envelope of real time.
- Local clocks are subject to crash with determine the crash of the related process, on the contrary the crash of s process does not imply the crash of the clock.

# progress assumptions (1)

- The *Progress Assumptions* constitute a fundamental of the model and can be syntethisedby the following statement :
  - Infinitely often a majority of the processes will be stable for a limited time interval.

- Analyzing distributed systems based on LANs it has been observed that their activity is characterized by long periods in which there exists a majority of stable processes alternating with short periods of instability.

- The intuition which derives from this observation is that as long as the system remains stable (i.e. failures are below a givem threshold) it is able to proceed in its computation in a limited time.

# Progress assumptions (2)

- The validity of the progress assumptions is confirmed by current hw and sw technologies and by the availability of OSs able to support soft real-time applications. Therefore per it is reasonable to assume that operations and communications offered by distributed systems are *timely* for most of their life.
- The introduction of the progress assumptions is important because it allows to solve consensus (when a system is stable it behaves exactly as if it was a synnchronous systems).
- In addition the progress assumptions are separate from the system model so it is possible to have different algorithms based on different progress assumptions for the same underlying system model.

# Stability predicates (1)

- In the specifications of the protocols defined for timed asynchronous systems we often resort to the use of stability predicates that verify system favorable conditions.
- Several different definitions have been used for stability predicates :
  - - stable predicates ,
  - - the Δ-F-partition e
  - - majority stable predicates.
- Two processes are **connected** in the interval [s, t] if they are *timely* in [s, t] and every message exchenged suffers a maximum delay of $\delta$ (*one-way time-out delay*).
- If the majority Sof the processes are pairwise connected in an interval [s, t] we say that S is a ***stable majority***.

# Stability predicates (2)

- A system is *majority-stable* in an interval if a stable majority exists.
- Clearly, in a given time interval there may be different stable majorities because not all the couples of processes are connected.

- A process p is majority-stable in an interval if it belongs to a stable majority in the interval.
- At this point we say that a system is *always eventually majority stable* if:

1. After each instability period the system becomes eventually majority-stable for at least $\Delta$ clock-time units;
2. Each process eventually becomes majority-stable for at least $\Delta$ clock-time units or suffers from a a crash.

# Stability predicates (3)

- Termination conditions for asynchronous systems (time-free) require the termination of an algorithm in a finite number of steps.
- In the case of synchronous systems these conditions are time-bounded that is they impose termination in a finite quantity of time.
- In the case of timed asynchronous systems we talk odf conditionally-timed conditions:
  - in a system which is always eventually majority stable if a process p is majority stable in an interval [t, t+E], then, an operation initiated at instant t must terminate by t + E.

# Rotating Leadership (1)

- The solutione of consensus uses the solution of another problem: the leader election, in the variant known as  the Rotating Leadership.

- Assumptions:
- 1. at any time instant there exists at most one Leader;
- 2. only a majority-stable process is elected as leader;
- 3. a process remains  leader for a limited time;
- 4. a process knows that it is a leader (it is not required that other processes know who the leader is);
- 5. the clocks of the processes are synchronized (the deviation between the clocks is limited by some constant)

# Rotating Leadership (2)

- More formally the second hypothesis can be expressed as follows:
  - 2. if a system is majority stable in an interval I, then for every process p belonging to a stable majority of I there exists an interval [s, s + LD] contained in I where p is leader (LD indicates the time when a process remains a leader);

- Assumption 4 allows to define a global time grid in which to allocate for each process a time slot in which to become a leader.

# Rotating Leadership (3)

- The algorithm:
- At the beginning of each time-slot each process is a candidate to be elected leader.
- Each process is associated with a priority and the election protocol ensures that only the highest priority process is elected.
- For a process to become a leader, however, it needs to receive a majority of replies to its candidacy and that these replications come in time.
- After sending its application, in fact, each process waits for a certain period of time to receive the candidacies of the other processes before responding to the application with the highest priority.
- After becoming the leader, a process remains as such for LD clock-time units, after which it is "dismissed".

# La Rotating Leadership (4)

- This protocol guarantees that when the system is majority stable every majority-stable process will have the highest priority in one of the elections, so everyone will eventually become leaders.
- The main reason this problem is solved in timed systems is the presence of local hardware clocks that evolve into a linear real-time envelope.
- If these were not available, it would not be possible to communicate by-time (ie the association of information content over time) and then to determine an upper limit on the delay of messages or to ensure that a process is no longer leading in an instant known to all other processes.

# From leader election to consensus

- When a process p becomes a leader, it first performs a broadcast to know if any other process has already reached a decision or is aware of a previous proposal.
- Only a process in the UP state will respond to this request for information.
- The leader then sends his proposal indicating his priority with it.
- Each process stores the value and priority of the proposal most recently received in a protocol state (since each leader has a higher priority than all of its predecessors it is easy to establish the most recent proposal).

# Consensus

- The current leader waits for 2 $\delta$ clock-time units to receive replicas after which:
- if it learns that a process has already decided for w, then he too will decide for w and inform all the other processes via broadcast;
- if none of the processes from which it receives an answer is aware of a previous proposal, then proposes its initial value, otherwise it proposes the previously proposed value;
- if he does not know of any decision or does not receive a sufficient number of answers, he will not take any action.

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

# Consensus (2)

- Each process p that receives a proposal with an upper limit on the transmission delay at most $\Lambda$ (maximum error allowed by the clock synchronization algorithm) and with a higher priority than the last proposal, stores the value and the priority of the proposal and responds to the leader by sending an ack to confirm the reception.
- On the other hand, when p receives the leader's decision, he too decides for that value.
- For both the leader, and for any other process that is in the restarting state, the receipt of a decision or of a timely proposal of a value determines the transition to the UP state.

# Consensus (3)

- A very important invariant of the protocol is that a majority of processes know the proposed value v when the leader decides for it.
- A process p that performs restart must re-initialize its protocol state before moving to the UP state.
- In this way the invariant is respected even when some process that knows the proposed value is "replaced" by other processes after having suffered a crash.

# Timed Asynchronous vs. Failure Detectors (1)

- The expressive capacity of the two models is different: the impossibility to implement a Perfect Failure Detector in a timed asynchronous system has been demonstrated.

- There is a different design philosophy of the system.

- Failure detectors hide aspects related to time at higher levels of abstraction. This represents a limit when the levels of abstraction are more than two.

# Timed Asynchronous vs. Failure Detectors (2)

- IIn this type of application time-outs are used at each level because a level that depends on another must be able to identify errors and mask them.
- In general, time-outs vary with the levels to which they are applied: the higher the level, the greater the time-out and the meaning and impact of its violation is different.