Designing Modular Services in the Scattered Byzantine Failure Model

E. ANCEAUME* C. DELPORTE-GALLET† H. FAUCONNIER† M. HURFIN* G. LE LANN††
* {anceaume,hurfin}@irisa.fr, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
† {cd,hf}@liafa.jussieu.fr, LIAFA, 75251 Paris Cedex 05, France
†† Gerard.Le_Lann@inria.fr, INRIA, 78153 Le Chesnay, France

Abstract— **In this paper, we propose the scattered byzantine failure model. In this model processes alternate correct and faulty periods. Specifically, during its faulty periods, a process behaves arbitrarily (one cannot expect anything from it during these periods) whereas during its correct periods, it behaves according to its specification. In that sense, the scattered Byzantine failure model generalizes the classical Byzantine failure model. We characterize two reliable services guaranteeing timeliness properties in the presence of Byzantine failures, namely the Clock Synchronization and the Δ-Atomic Broadcast. We identify necessary and sufficient conditions to ensure the correctness of both services in the scattered byzantine failure model.**

I. INTRODUCTION

Fault tolerance is an important issue in distributed computing. To tolerate failures (from benign to malign ones) physical redundancy is mandatory. Replication of critical data and functionalities on a group of processors allows to increase the overall reliability of the system. To be able to coordinate the activities of the processors, a significant body of work on replication techniques and agreement problems has been done. Classically, the proposed solutions assume that all the processors involved in the computation are classified into two categories, and that this classification holds forever: the *correct* processors and the *faulty* ones. A processor is correct if it behaves according to its specification until the completion of the computation; otherwise it is faulty. Hence, to correctly design fault tolerant applications, one has to suppose that a maximal subset of the whole set of the processors may possibly fail, and once they have failed, no more failure can happen.

Although the above failure model fits the requirements of many common distributed applications, the specificity of the space domain makes it not adapted essentially because of the harshness of the environment. Quoting Shirvani et al. [22], "radiation (such as alpha particles and cosmic rays), electronic interference and power supply glitches (an example is the undesired bit-flip effect which is the change of state in the content of a storage element) may cause transient and frequent faults in electronic devices". Such faults cause single-event upsets which are a major concern in a space environment. Additionally, running times of the applications are extremely long (typically, the useful orbital life of a satellite is expected to last several years). Both reasons make unrealistic the assumption that only a subset of the processors can fail during the whole duration of a mission. Furthermore, for space, weight and cost savings, drastic limitations are imposed on the computer system: spare processors are limited, and in any case, human intervention to replace or repair hardware after the launch of the satellite is rare. Thus a processor cannot be excluded or precluded from participating to the forthcoming computations under the pretence that it commits a transient failure. Finally, most of the failures are recoverable, and more importantly, are accidental: physical phenomena such as heavy ion bombardments or alpha radiation can arbitrarily affect the behavior of a processor by altering the executed code, the data, the program counter, or the registers. But, checking procedures or reconfiguration mechanisms are usually available in the computer system [22]. They guarantee that faulty processors can recover some *operational* state. In particular, they guarantee that the operating system and the hardware (processor and physical clock) will operate correctly again. On the other hand, finding an operational state does not mean finding a *safe* state: an operational state can be reached through an arbitrary sequence of transitions which may have altered the local variables, the program counter, and/or the registers. Thus, an operational may not be semantically correct.

In this general context, we simultaneously address two different issues. The first one consists in providing algorithmic solutions that guarantee processes to converge progressively toward safe states with respect to the given service. This convergence is obtained through self-stabilization properties: starting from any arbitrary state, a process reaches a safe state in a bounded time. The second issue concerns the service availability. Beyond the properties of self-stabilization [12], we provide algorithmic solutions that ensure a complete availability of the system in spite of concurrent failures.

We are conducting this work supported by the French Space Agency (CNES - CENTRE NATIONAL D'ÉTUDES SPATIALES). The aim is to propose both a generic architecture and an universally applicable development method called TRDF [17] on which the next generation of satellites could be based.

A. Contributions of this Work

We start by formally defining the *scattered byzantine failure model*. In this model, all the processors involved in a computation may alternate good periods and bad ones. Specifically, during its bad periods, a processor behaves arbitrarily: one

Proceedings of the ISPDC/HeteroPar'04
0-7695-2210-6/04 $20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

cannot expect anything from it during this period. For example, a bit-flip in a location of memory that contains the instructions of a program may cause the program to produce incorrect results, and thus to commit a byzantine failure. [1] On the other hand, during its good periods, a processor behaves as prescribed by its specification. This model of failure does not limit the number of faulty processors. Rather, it frees the application designer from the traditional and recurrent question: "What happens if the quorum of processors that were supposed to fail is exceeded ?" With the scattered byzantine failure model, all the processors may possibly fail: the only constraint, as will be detailed later, is to bound the number of concurrent failures. In that sense, this model is a generalization of the traditional byzantine failure model.

Given this failure model, we then present a solution to two important problems that are inherent to distributed systems: the *Clock Synchronization* and the Δ-*Atomic Broadcast* problems. Both of them are essential building blocks in the design of fault-tolerant real-time applications. In the particular context of the aerospace case study, both problems have been tackled to find a solution of the very problem of real-time distributed scheduling. Both building blocks address different facets of our work: first, for each service s, a characterization of the minimal period of time that has to elapse after a process failure is given. This period of time, called in the following post-fault period and whose duration is denoted D_p^s, enables a faulty process to recover a safe state. Second, for non atomic services, the notion of fore-fault period is introduced. A fore fault period, whose duration is noted D_f^s, reflects the completion time of a given service s. It ensures that if service s has been invoked by some process D_f^s time units before some process failure then s is completed at all the processes.

Finally we identify necessary and sufficient conditions to ensure the correctness of the algorithms we propose. As aforementioned, constraints have to be put on the number of concurrent failures —rather than on the maximum number of failures during the execution of a service. At any given time, the maximal number of processors that are in a faulty period is denoted by t. To keep the presentation of the algorithms as simple as possible, we assume in this paper that $t = 1$ and show that the number n of processors has to be at least equal to 4. This assumption ($n > 3t$) is a classical requirement when one considers byzantine failures with no authentication mechanism. The choice of $t = 1$ which is also imposed by the fact that the number of processors used during a space mission has to be as small as possible, can easily be generalized to $t \geq 1$.

In this paper, we only study two specific services, namely the Δ-atomic broadcast and the clock synchronization. However the same approach can be applied to any other distributed service insofar as the service manipulates evanescent data. Roughly speaking, data are evanescent if variables (control and critical ones) have a limited duration of validity (e.g., limited to the execution of a loop, or to the execution of a task activated by an external event), or are refreshed periodically (e.g., in a clock synchronization algorithm, the value of the clock is recomputed cyclically), and if logs (when they exist) are purged recurrently to keep only information related to the recent past or the immediate future, then our approach can be directly used.

Paper Road map Section II presents the model assumptions and the scattered byzantine failure model. Related works is discussed in Section III. Sections IV and V revisit the Clock Synchronization and Atomic Broadcast services according to the scattered byzantine failure model. Section VI concludes, and discusses the scope of our results.

[1] In the following, a program or activity running on a processor is called a process. Several processes can execute on the same processor. Both processes and processors can experience failures. For example in the above example, the bit-flip damages the execution of a program, leading to its failure.

II. Model of the System

A. Computational Model

The system consists of a finite set of processes $\Pi = \{p_1, \ldots, p_n\}$ communicating and synchronizing with each other by sending and receiving messages over a completely reliable connected point-to-point network. The system is synchronous, meaning that *i)* there exist known upper and lower bounds on the time required for a process to execute a computation step, *ii)* every process has access to a hardware clock with bounded drift rate with respect to real time ($\rho > 0$), and *iii)* there is a known upper bound on transmission delay of the messages (δ denotes the value of the maximal message transfer time).

Each process is modeled as an automaton with (possibly) an infinite number of states. A state comprises the execution context of the process (e.g. registers, stack, program counter, binary code). The transition function takes as input both the current state and the set of received messages, and encode the protocol the process has to follow.

B. The Scattered Byzantine Failure Model

We assume that all processes can alternate *correct* and *faulty* periods. At any time, at most t processes are in their faulty periods. The assumption $n > 3t$ is a classical requirement when one considers byzantine failures with no authentication mechanism. Regarding a faulty period, we decompose it into two parts:

- The first one, the *bad period*, encompasses the time interval during which processes commit byzantine failures, i.e., during which their behavior can deviate arbitrarily from the one prescribed by the given algorithm/service. Note that byzantine failures are the more severe kind of failures and encompass crash, omission, timing and values failures. Byzantine processes may omit to send messages, may change their content or may generate spurious messages. However masquerading is impossible. To prevent one process from masquerading as another, messages are signed; a process that receives a message must verify the signature on the message. If the signature on the message is not valid, then the process does not

0-7695-2210-6/04 $20.00 © 2004 **IEEE**

IEEE Computer Society

accept the message. Note that signed messages are necessary whenever one wants to solve distributed problems. Indeed, if a process can masquerading as another, then it can masquerading as all the other processes, and thus no distributed problems can be solved.

A process ends its bad period whenever it reaches an operational state. With respect to the computational model, an operational state is reached whenever a faulty process recovers the correct automaton, and makes steady progress in its computation, that is, is able to apply the right transition functions associated to its automaton; however its execution context may be altered. As previously said, such a state can be reached by resorting to automatic recovering procedures.

For layered services, bad periods are no more defined according to the automaton of a given service but according to the automata of all the layered services. Specifically, for a given processor P [2] and a set of K layered services, P is in a bad period for level k service, with $1 \leq k \leq K$, if there is some k', $1 \leq k' \leq k$, such that the automaton associated to level k' service is in a bad period. An operational state is reached at level k, if both the bad period at level k and the post-fault period at level $k-1$ are ended (see hereafter).

- The second period, called the *post-fault period*, defines the delay needed for a process to reach a safe state from an operational one. A process reaches a safe state whenever its execution context is consistent with the one of the processes in correct periods. An analysis of the distributed protocol that implements the given service s enables to determine the duration D_p^s of the post-fault period.

  For layered services, the post-fault period of level k service begins when both the bad period of level k service and the post-fault period of level $k-1$ are over. For $k = 1$, the post-fault period of the service starts at the end of its bad period. To summarize, for level k service, the faulty period starts at the beginning of its bad period and finishes at the end of its post-fault period. This is illustrated in Figure 1. In this figure, the different periods that encounter some processor P during the execution of two layered services, namely the Δ-atomic broadcast (level k), and the clock synchronization (level $k-1$) are shown.

Whenever a process reaches a safe state, it becomes correct, that is all the effects of its previous failures have disappeared. A process remains correct as long as it does not commit a new byzantine failure. During this period of time, the *correct period*, a process behaves according to the specification of the service it executes. Typically, the specification of a service expresses requirements on the program execution state at a given time, or on the desired (or undesired) evolution of this state during a long-lasting activity. In the latter case, a computing activity started during the correct period of a process may not be completed at the very moment of its failure. To exactly identify completed activities from uncompleted ones, a correct period is divided into two successive periods: the first one is called the *good period* while the second one is the *fore-fault period*. The fore-fault period reflects the worst case execution time of a given service s. In the following we determine the duration D_f of the fore-fault period of the Δ-atomic broadcast service (see Figure 1). This period ensures that if service s has been invoked by some process p D_f^s time units before some process q failure then s is completed at all the processes. Clearly nothing distinguishes a process in a good period from a process in a fore-fault period from a behavioral point of view; however the notion of fore-fault is needed to limit the number of concurrent failures.

Note that, for the clock synchronization service, such a period does not exist. This comes from the fact that the functionality provided by this service is atomic in the sense that a query to this service is immediately followed by a response (i.e., the current time). This contrast with the Δ-atomic broadcast in which a query to atomically broadcast some message m requires a distributed activity among all the processes to be effective.

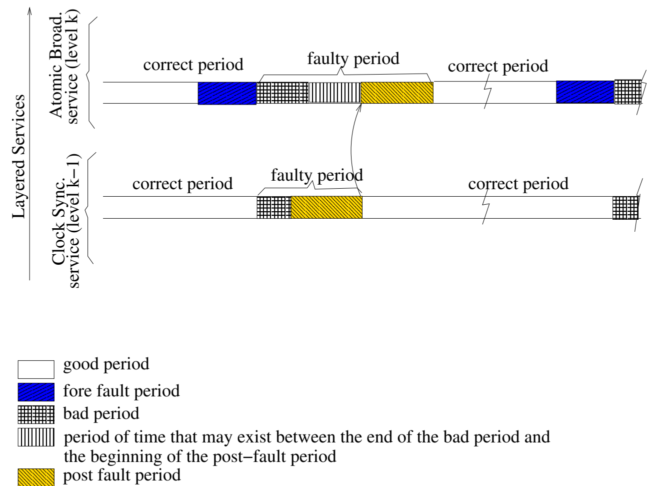

Fig. 1. Fore and Post-fault Periods of the Atomic Broadcast Service and the Clock Synchronization Services at Processor P.

III. Related Works

Many works consider that, after a failure, a process can recover a safe state that is consistent with the state in which it was at the time the failure occurred. Very few works consider an arbitrary failure model. In fact, in less complex failure models, solutions heavily rely on the fact that failure detection is much easier to handle. For example, in the crash failure model, the detection of a local failure is easy to observe. Detection of a remote crash can also be done using failure detectors. During its recovery, a process is aware that it is currently in a fore-fault period. This information can also be known and trusted by the other processes. Additionally, in a crash failure model, a running process always behaves

[2] When speaking of all the layered services, we use the term "processor" to refer to the processes involved in the services.

0-7695-2210-6/04 $20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

according to its specification. Consequently, any data saved in its permanent storage or logged by another process can be used to recover to a safe state [16], [7], [1], [4]. All these assumptions are no more true when one consider arbitrary failures.

Works done on proactive security [9], [10], [5] have strong connections with this paper. In a proactive security system, any processes can experience arbitrary failures but during a fixed period of time, no more than t processes can be faulty. To ensure security requirements, algorithms perform periodic computations of critical data (like for example secret keys [9]). In all these works (including ours), a byzantine failure is not necessarily detected: locally a process does not know whether it is correct or not.

In [21], Reischuk design an agreement protocol able to tolerate malicious failures as long as they remain stationary for a given interval of time. In [8], the concept of mobile faults is introduced. In this works, a malicious agent is able to corrupt a process and to move from one process to another one. Malicious agents are characterized by the fact that their moving speed is limited. More precisely, a parameter called the roaming pace is used to denote the minimal amount of time that has to elapse between the time at which an agent leaves an host and the time at which it starts to corrupt another process. In our model, a similar time interval exists. Its duration is equal to $D_p + D_f$. More precisely, if at a given time τ, one process enters a post-fault period while $t-1$ other processes remain in a bad period, then no other process can enter a bad period before time $\tau + D_p + D_f$. The proposed model refines the interval of time into two bounded phases of computation related respectively to the last process that has experienced a failure and the next process that will experience a failure. When considering stacks of protocols, this approach allows to establish in a more simple way the assumptions (values of D_f and D_p) that have to be made for each layer: assumptions related to the fore-fault period and the post-fault period are cumulated separately. Another difference between our solution and the above mentioned solutions lies in the fact that they proceed in rounds, where processes that are not corrupted approximately agree on the time at which a round start.

In [6], the authors describe a synchronization algorithm that relies on a similar model. Yet the algorithmic solution is based on a convergence function whereas our solution appears as an extension of the protocol proposed in [23] that achieves optimal accuracy.

IV. Clock Synchronization Service

To be able to build control systems with real-time requirements, a clock synchronization algorithm is needed. It allows processes to update their clocks to overcome the effects of drifts and failures. In the space domain, some processes may have access to an external source of timing signal (GPS time for example). However, there exists periods of time during which this external service is unavailable essentially because of technical difficulties. [3]

Specifically, the goal of the clock synchronization algorithm is to guarantee that the maximum deviation between the logical clocks of all the processes (that are in a correct period) is bounded, and that they are within a linear envelope of real time. Clock synchronization protocols compute the logical clock according to *i)* the value of the local physical hardware clock and *ii)* messages exchanged with the other processes.

A. Definition of the problem

We consider that each process p_i has a physical hardware clock, denoted R_i, and computes its logical time, denoted C_i, by adding a locally determined adjustment to this physical clock [23]. The following two assumptions are made:
1) the rate of drift of physical clocks from real time is bounded by a known constant ρ. That is, if $R_i(\tau)$ is the reading of the physical clock of process p_i at time τ, then for all $\tau_2 \geq \tau_1$:

$$(1+\rho)^{-1}(\tau_2 - \tau_1) \leq R_i(\tau_2) - R_i(\tau_1) \leq (1+\rho)(\tau_2 - \tau_1)$$

2) there is an upper bound δ on the time required for a message to be prepared by a process, sent to a set of processes and processed by the recipients of the message.

Given both assumptions, a clock synchronization algorithm has to satisfy the following two properties: For all processes p_i and p_j in good period at time τ:
Agreement: There exists a constant ϵ such that:

$$| C_i(\tau) - C_j(\tau)| \leq \epsilon$$

For any process p_i such that p_i is in a good period at time τ, then:
Accuracy: There exists a constant γ such that, for any execution of the algorithm,

$$\tau/(1+\gamma) + a \leq C_i(\tau) \leq \tau(1+\gamma) + b$$

for some constants a and b that depend on the initial conditions of the execution.

The agreement property guarantees that the maximum deviation between the logical clocks of any two processes that are in a correct period is bounded. The accuracy property states that the logical clock of a process remains in a linear envelope of real time while this process is in a correct period.

B. The Algorithm

1) Principles of the Srikanth and Toueg Algorithm: We propose a clock synchronization protocol inspired from the one of Srikanth and Toueg [23]. Their solution relies on periodic resynchronizations. We have chosen the principles of their protocol mainly because it is simple, and efficient: it achieves optimal accuracy, that is, the accuracy of synchronized clocks (with respect to real time) is as good as that specified for the underlying hardware clocks. To simplify, we assume that initially the logical clocks of processes that

[3] As quoted from Barak et al [6], GPS receivers cannot be use as an attractive alternative to clock synchronization protocols mainly because the GPS signal is easily corrupted by radio transmitters, radio interference, or malfunctions in the satellites themselves.

0-7695-2210-6/04 $20.00 © 2004 IEEE

IEEE Computer Society

```
initialization:
1  k := 1
2  MRec := ∅        /* MRec: set of all received messages */
3  MAcc := ∅             /* MAcc set of accepted messages */

task 1 || task 2

task 1:                              /* Synchronization time */
4  when C(t) = kP
5    send (Sync, k, i) to all other processes

task 2:
6  on receipt of (Sync, m, j) from p_l at time T = C(t)
7    if (l = j)                              /* validity tests */
8      ∧(−(1 + ρ)ε ≤ T − mP(1 + ρ) ≤ (δ + ε)(1 + ρ)) then
9      send(Sync, m, j) to all
10   else
11     if j ≠ l then
12       add(Sync, m, j, l) to MRec
13       if ∃l' : l' ≠ l s.t. (Sync, m, j, l') ∈ MRec then
14         add (Sync, m, j) to MAcc
15         if ∃j' : j' ≠ j s.t.(Sync, m, j') ∈ MAcc
16             ∧k ≠ m + 1 then
17           C(t) := mP + α
18           MRec := MAcc := ∅                  /* cleaning*/
19           k := m + 1        /* clock resynchronization */
20         endif
21       endif
22     endif
23   endif
```

Fig. 2. Clock Synchronization Algorithm Run by Process p_i.

are in correct periods are synchronized. Principles of their protocol is as follows. Let P be the logical time between resynchronizations. The resynchronization protocol proceeds in rounds, a period of time during which processes exchange messages and reset their clocks: when the logical clock of some process p_i shows time kP, with $k \geq 1$, this process broadcasts a resynchronization message, indicating that it is ready to resynchronize. When a process in a correct period receives $t+1$ messages, the algorithm ensures that *i)* all the processes that are in correct periods receive them at least 2δ time units after their sending and *ii)* one of them has been sent by a process during a correct period. Upon receipt of a resynchronization message from $2t+1$ processes, process p_j knows that $t+1$ other processes are ready to synchronize. Thus, p_j accepts this message and resynchronizes its logical clock to $kP+\alpha$, with α some constant value guaranteeing that logical clocks are never set back. Then p_j relays this message to ensure that all the processes that are in correct periods will resynchronize too.

2) Principles of the Algorithm we Propose: We extend this clock synchronization algorithm to handle scattered byzantine failures, that is, to guarantee that *i)* the local structures of the processes that are in correct periods are never corrupted by the recovering processes, and *ii)* faulty processes recover by resynchronizing its local clock within a bounded delay (i.e., within $2P(1+\rho)$ time units).

The algorithm is detailed in Figure 2. Processes locally maintain a resynchronization counter k, and two buffers of resynchronization messages. The first one, $MRec$ contains the set of all the relayed messages. Their pattern is as follows: $(sync, k, s, r)$, where $sync$ is the type of the message, k represents the resynchronization number, s the identifier of the process that sends the resynchronization message, and r the identifier of the process that relayed it. The second buffer $MAcc$ contains the set of resynchronization messages that have been relayed twice. Their pattern is similar to the ones contained in MRec, except that the identity of the process that has relayed the message does not appear.

Periodically (when the logical clock of some process p_j reaches kP, line 4), each process p_j sends to all the processes (except itself) a resynchronization message to inform all the other processes that it is ready to make its kth resynchronization. Upon receipt of such a message (line 6), a recipient p_i first checks whether the receipt of this kth resynchronization message makes sense (that is if both p_i and p_j are within the same round of resynchronization, line 8). In that case, p_i relays this message to all the processes, otherwise, it discards it. Upon receipt of a relayed message from two processes (different from the source of the resynchronization message), a recipient p_i keeps this relayed message (line 14): p_i knows that there are at least two processes that have received the resynchronization message from a source. However, p_i does not know whether the source of the message is correct or not. Indeed, nothing prevents a process in a bad period from relaying its own resynchronization message (which might lead to an incorrect resynchronization!). Thus, p_i makes its kth resynchronization only when it has received two other relayed resynchronization messages corresponding to a second source (line 15). We say that p_i *accepts the resynchronization message* k. In other words, p_i resynchronizes its logical clock whenever it receives a request to resynchronize from two different sources, each request being relayed twice. Similarly to the protocol of Srikanth and Toueg [23], p_i resynchronizes its logical clock with $kP+\alpha$, with α some constant that prevents from setting clocks backward. Specifically, $\alpha \geq ((1+\rho)\epsilon + 2\delta)(1+\rho)$, and $P \geq 2\delta(1+\rho) + 2\epsilon$.

The following proposition gives an intuition of how the algorithm handles scattered byzantine failures. Specifically, it shows that *i)* the local structures of the processes that are in correct periods are never corrupted by the recovering processes, and *ii)* faulty processes recover by resynchronizing its local clock within $2P(1+\rho)$ time units, i.e., the duration of the post-fault period equals to $2P(1+\rho)$ time units.

Proposition 1: Suppose that process p_i recovers an operational state at time τ (i.e., enters its post-fault period). Then by time $2((P-\alpha)(1+\rho)+2\delta)$, p_i is resynchronized with all the processes that are in correct periods.

We first show that the period between resynchronizations is bounded, and is equal to $(P-\alpha)(1+\rho)+2\delta$. From the

0-7695-2210-6/04 $20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

algorithm, every process p_i that is in a correct period and sends its $(k+1)$th resynchronization message does so before time $(k+1)P$ on its clock (i.e., $C_i(\tau) = (k+1)P$). Now, the time that elapses since p_i's kth resynchronization is no more than $(k+1)P - (kP+\alpha) = P - \alpha$ on p_i's clock. Thus no more than $(P-\alpha)(1+\rho)$ since the slowest process did its kth resynchronization. Thus every process in a correct period accepts the $(k+1)$th resynchronization within a further 2δ time units, which proves the assertion.

When process p_i recovers, both its logical clock and its buffer of messages are possibly corrupted. On the other hand, all the other processes are in correct periods and thus continue to resynchronize their logical clock. Thus, it is quite possible that upon receipt of a mth resynchronization message, p_i does not relay it because its logical clock is desynchronized. However, it stores all the relayed messages in its $MRec$ buffer, and since all the other processes are in correct periods (by assumption of the proposition, there is only one process in a bad period at a given time in the system), then p_i receives all the relayed messages for the mth resynchronization. Thus p_i stores at least two resynchronization messages sent by two different sources. However, since its local variables may be corrupted, the value of k may be equal to $m+1$. Thus the "if-then" test fails, thus p_i does not accept the resynchronization message m, and thus does not reset its local variables (including k). On the other hand, the next resynchronization will succeed, and will enable p_i to resynchronize its logical clock and reset its local variable k. Thus if p_i recovers at time τ, then two resynchronizations later p_i is resynchronized with all the processes that are in correct periods, that is by time $\tau + 2((P-\alpha)(1+\rho)+2\delta)$.

From this proposition, it follows that the duration of the post-fault period is equal to $2((P-\alpha)(1+\rho)+2\delta)$ time units, while the one of the fore-fault period is null (see Section II-B).

For space limitations, proofs of correctness of the algorithm cannot be given in this paper. The reader is invited to read the full paper [2].

V. Δ-ATOMIC BROADCAST

A. Definition of the problem

Atomic broadcast is a powerful communication paradigm for fault-tolerant computing. It allows processes to reliable broadcast messages, so that they agree on the set of messages they deliver and the order of messages deliveries. Atomic broadcast has been identified as a basic communication primitive used by many systems [20]. Atomic Broadcast is defined in terms of two primitives broadcast($\prec m, i \succ$) and deliver($\prec m, i \succ$), where m is a message drawn from a set $\mathcal{M}$ of possible messages, and i, the identifier of the message. In the scattered byzantine failure model, the Δ-Atomic Broadcast has to satisfy the following four properties:

- *Validity*: If process p_i broadcasts $\prec m, i \succ$ at time τ during a good period, then every process p_j in a good period at time τ delivers $\prec m, i \succ$ exactly once during this correct period.
- *Agreement*: If process p_i delivers $\prec m, j \succ$ at time τ during a good period, then any process in good period at time τ delivers $\prec m, j \succ$
- *Δ-timeliness*: If process p_i delivers $\prec m, j \succ$ at time τ during a good period, then p_j broadcast $\prec m, j \succ$ between time $\tau - \Delta$ and time τ.
- *Total Order*: If two processes p_i and p_j deliver two messages $\prec m_1, k_1 \succ$ and $\prec m_2, k_2 \succ$ during a good period then both messages are delivered in the same order by p_i and p_j.

If we assume that faulty periods last forever, like in the classical model, the specification given above corresponds to the classical one [11], [14], [18].

B. Algorithm

The atomic broadcast algorithm is shown in Figure 3. It relies on the clock synchronization algorithm detailed in the previous section. The general structure of this algorithm is not original: there is a sending task (task 1), a delivering task (task 2), and a task in charge of reaching an agreement on the received messages and on their ordering (task 4). On the other hand, the innovative aspect of this algorithm is to cope with the scattered byzantine failures, that is, to guarantee that *i)* the local data structures of the processes that are in correct periods are never corrupted by the recovering processes, and *ii)* the recovering processes have again both a correct state and correct data structures in a bounded time (task 3). The main strategy looks like the one adopted in the clock synchronization algorithm: messages are initially sent by the source to all the processes except the source itself, and are relayed to all the processes. Furthermore, upon receipt of a message, the recipient checks whether the logical dates contained in the received message are mutually consistent (lines 19-20).

When process p_s executes broadcast($\prec m, s \succ$), it sends a message $M = (\prec m, s \succ, d_s, d_s)$ to all the processes except itself. Variable d_s is the local time at which p_s broadcast $\prec m, s \succ$. When a process p_j relays this message, it sends a message $M = (\prec m, s \succ, d_s, d_j)$ to all the processes. Variable d_j is the local time at which p_j relays $\prec m, s \succ$. Each process p_i maintains two different buffers of messages. The first one, $MRec$ contains information about the recent messages $(\prec m, s \succ, d_s, d_j)$ that have been relayed by a process p_j that seems to be correct. Information about a message directly received from the source is never logged. After a positive control of the value of d_j (line 19), $(\prec m, s \succ, d_s, j)$ is kept in $MRec$. Note that the origin of the relayed message is logged rather than the date at which the relayed message was issued. The second buffer $MDeliver$ is an array of sets of messages such that $MDeliver(d)$ contains all the messages that have to be delivered at local time $d + 2(1+\rho)\delta + 2\alpha + 2\epsilon$. The time at which a process delivers a message is determined by the time at which the source has broadcast the message. Messages broadcast at the same logical time are delivered in a predefined order based on the identities of sources and messages.

IEEE COMPUTER SOCIETY

A message ($\prec m, s \succ$) is said to be accepted by a process p_i when $\prec m, s \succ$ is inserted in the set $Mdeliver$ managed by p_i. If $\prec m, s \succ$ is inserted by p_i in the set $Mdeliver$ for the first time at time τ, the local time $C_i(\tau)$ is called the acceptation time of ($\prec m, s \succ$) by p_i.

For space limitations, we only give an intuition of correctness proofs. Furthermore, for legibility reasons, proof of Lemma 1 is given in the full paper [2].

C. Sketch of the Correctness Proof

Lemma 1: Let p_i and p_j be any two processes such that p_i is in a correct period during the time interval $[\tau, \tau + \delta]$ and p_j is not in a bad period during the same time interval. If message m is sent by p_j at time τ, then m is received by p_i and the validity test (lines 19-20) holds.

Theorem 1: Consider any synchronous system subject to scattered byzantine failures and in which physical clocks are ϵ-synchronized clocks. For $\Delta = 2(1 + \rho)\delta + 2\alpha + 3\epsilon$, the algorithm given in Figure 3 ensures all the properties of Δ-Atomic Broadcast, assuming that the fore-fault period lasts at least Δ time units and the post-fault period $(1 + \rho)\Delta$ time units.
Proof of the theorem follows from the following four properties:

Lemma 2 (Validity property): Suppose that p_s broadcasts $\prec m, s \succ$ at time $d_s = C_s(\tau)$ during a good period. Because $D_f > 2\delta$, any process p_j in a good period at time τ is in a correct period during the time interval $[\tau, \tau + \delta]$. All messages sent by p_s at time τ are received by all processes p_j. By Lemma 1, any process that receives a message from p_s relays it (execution of line 22). By assumption, there are at least two such processes. Again, by Lemma 1, any process p_i in a correct period during $[\tau, \tau + 2\delta]$ receives at least two relayed messages. Consequently, the validity test (lines 19-20) holds at least twice at p_i. Therefore, p_i accepts $\prec m, s \succ$ before local time $d_s + 2(1 + \rho)\delta + 2\alpha + 2\epsilon$ (line 28). Moreover, $\prec m, s \succ$ is inserted in a single entry of $MDeliver$, namely $MDeliver(d_s)$. Hence, this message is delivered only once by process p_i during a correct period.

Lemma 3 (Agreement property): Suppose that at time τ', p_i is in a good period and delivers message $\prec m, s \succ$. By task 2, $C_i(\tau') = d_s + 2(1 + \rho)\delta + 2\alpha + 2\epsilon$. First, we show that $\prec m, s \succ$ delivered at time τ' has not been erroneously inserted in $MDeliver$ while p_i was in a bad period. Proof is by contradiction. Suppose that p_i inserted message $\prec m, s \succ$ in $MDeliver(d_s)$ while it was in a bad period. Assume that p_i started the subsequent post-fault period at time τ''. If message $\prec m, s \succ$ was not removed from $MDeliver$ when p_i executed task 3 at time τ'', condition $(d_s - \epsilon) \leq C_i(\tau'')$ had to be true. As p_i is in a good period at time τ', we have $\tau' - \tau'' > D_p$. Thus, $C_i(\tau') - C_i(\tau'') > D_p/(1 + \rho)$. As $D_p = (1+\rho)(2(1+\rho)\delta + 2\alpha + 3\epsilon)$ and $C_i(\tau') = d_s + 2(1 + \rho)\delta + 2\alpha + 2\epsilon$, we have: $ds > C_i(\tau'') + \epsilon$. This contradicts the fact that $d_s \leq C_i(\tau'') + \epsilon$.

In the first stage of the proof, we have shown that message $\prec m, s \succ$ is accepted and delivered by p_i during the same correct period. Now, suppose that p_i accepts $\prec m, s \succ$ at some time $\tau \leq \tau'$. To insert $\prec m, s \succ$ in $MDeliver(d_s)$, p_i must have received at least two relayed messages by time τ: $(\prec m, s \succ, d_s, d_j)$ from p_j and $(\prec m, s \succ, d_s, d_k)$ from p_k (with $k \neq j$ and $s \neq j$ and $s \neq k$). By definition, at least one of these processes (p_j or p_k) is in a good period at time τ and was not in a bad period when it relayed the message from p_s. Let p_j be this process. As the validity test has been evaluated to true by p_j before relaying the message, we have: $d_j \leq (d_s + (1 + \rho)\delta + \alpha + \epsilon)$.

Two cases have to be considered:

i) p_k was in a good period when it relayed p_s message. In

```
initialization:
1  MRec := ∅            /* Mrec: set of all received messages */
2  ∀d : MDeliver(d) := ∅
        /* MDeliver(d): messages to be delivered at time d */

task 1 || task 2 || task 3 || task 4

task 1:
3  on ABcast(m) do
4     d_i := date();              /* query of the local clock */
5     send (≺ m, i ≻, d_i, d_i) to all other
6  done

task 2:
7  at time d do
8     Deliver each ≺ m, s ≻∈ MDeliver(d') with
9        d' = d − 2(1 + ρ)δ − 2α − 2ε in some predefined order
10 done

task 3:
11 while true do
12    forall (≺ m, s ≻, d_s, j) ∈ MRec s.t.
13       ¬((d_s − ε) ≤ date() ≤ (d_s + 2(1 + ρ)δ + 2α + 2ε)) do
14       remove (≺ m, s ≻, d_s, j) from MRec;
15       MDeliver(d_s) := ∅
16 done

task 4:
17 on receive (≺ m, s ≻, d_s, d_j) from p_j do
18    d_i := date();
      /* validity test:*/
19    if ((d_s − ε) ≤ d_j ≤ (d_s + (1 + ρ)δ + α + ε))
20       and ((d_j − ε) ≤ d_i ≤ (d_j + (1 + ρ)δ + α + ε)) then
21       if (s = j) and (d_s = d_j) then
22          send (≺ m, s ≻, d_s, d_i) to all
23       else
24          if (s ≠ j) then
25             add (≺ m, s ≻, d_s, j) to MRec
26             if ∃k, k' : k ≠ k' ∧ (≺ m, s ≻, d_s, k) ∈ MRec
27                   ∧(≺ m, s ≻, d_s, k') ∈ MRec then
28                MDeliver(d_s) := MDeliver(d_s) ∪ {≺ m, s ≻}
29          endif endif endif endif done
```

Fig. 3. Atomic Broadcast Algorithm Run by Process p_i.

0-7695-2210-6/04 $20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

that case, any process p_l such that p_l is in a good period at time τ also receives a message from p_k. Consequently, the message $\prec m, s \succ$ is accepted by p_l before time $d_s + 2(1+\rho)\delta + 2\alpha + 2\epsilon$ and delivered by p_l.

ii) p_k was not in a good period when it relayed p_s message. Thus, p_s was in a correct period when it broadcast its message $\prec m, s \succ$ at time d_s. End of the proof follows by Lemma 2.

Lemma 4 (Δ-timeliness property): Straightforward from the fact that if $\prec m, j \succ$ is delivered at time d then the message was broadcast at time $d_s = d - 2(1+\rho)\delta - 2\alpha - 2\epsilon)$ and by Lemma 3.

Lemma 5 (Total order property): Messages are delivered according to their sending time which guarantees a total order delivery (identifier of the sender can be used to break symmetry if needed).

Theorem 2: Consider any synchronous system subject to scattered byzantine failures in which physical clocks are not ϵ-synchronized clocks. For $\Delta = 2(1+\rho)\delta + 2\alpha + 3\epsilon$, the algorithm given in Figure 3 altogether with the one in Figure 2 ensure all the properties of Δ-Atomic Broadcast, assuming that the fore-fault period lasts at least Δ time units and the post-fault period $(1+\rho)\Delta + 2((P-\alpha)(1+\rho) + 2\delta)$ time units. Proof directly follows from Theorem 1 and Proposition 1.

VI. Conclusion

In this paper, we have considered the byzantine recovery problem. We have formalized the scattered byzantine failure model. This model enables a process to alternate correct and faulty periods. Such model has been mainly inspired from the space model. We have considered two fundamental problems within this model: the clock synchronization problem and the atomic broadcast problem. We have revisited their specifications and designed simple and efficient solutions for these problems. Notice that, when failures last forever, the specifications (and the solutions) we propose come down to the ones proposed in the classical Byzantine model. Finally, we have evaluated the time needed for a process in a faulty period to recover a correct state and thus to resume correct operation with all the other processes.

We are currently working on the real-time distributed scheduling problem. Solving this problem in our model of computation amounts in ensuring that all the processors deliver the same sequence of messages in an order approximating EDF (Earliest Deadline First) ordering system wide in the presence of scattered byzantine failures. To the best of our knowledge, this problem has never been investigated so far.

References

[1] Aguilera M.K., Chen W., and Toueg S., Failure Detection and Consensus in the Crash-recovery Model. *Distributed Computing*, 13(2):99-125, 2000.

[2] Anceaume E., Delporte-Gallet C., Fauconnier H., Hurfi n M., and Le Lann G., Modular Reliable Services with Byzantine Recovery. *Technical Report IRISA*, 2004.

[3] Aguilera M.K., Delporte-Gallet C., Fauconnier H., and Toueg S., Thrifty Generic Broadcast. *In Proc of the 14th Int. Symposium on Distributed Computing*, LNCS 1914, pages 268-283, 2000.

[4] Backes M. and Cachin C., Reliable Broadcast in a Computational Hybrid Model with Byzantine Faults, Crashes, and Recoveries. *In Proc. of the Int. Conference on Dependable Systems and Networks (DSN'03)*, pages 37-46, 2003.

[5] Backes M, Cachin C., and Strobl R., Proactive secure message transmission in asynchronous networks. *In Proc. of the 22nd ACM Symposium on Principles of Distributed Computing (PODC 2003)*, pages 223-232, 2003.

[6] Barak B., Halevi S., Herzberg A., and Naor D., Clock Synchronization with Faults and Recoveries. *In Proc of the 19th ACM Symposium on Principles of Distributed Computing (PODC 2000)*, 2000.

[7] Boichat R. and Guerraoui R., Reliable Broadcast in the Crash-Recovery Model. *In Proc of the 19th Symposium on Reliable Distributed Systems (SRDS'00)*, pages 32-41, 2000.

[8] Buhrman H., Garay J., and Hoepman J., Optimal Resiliency against Mobile Faults. *In Proc. of the 25th Int. Symposium on Fault-Tolerant Computing*, pages 83-88, 1995.

[9] Canetti R., Gennaro R., Herzberg A. and Naor D., Proactive security: Long-term protection against break-ins. *RSA Laboratories' CryptoBytes*, 3(1):1-8, 1997.

[10] Castro M. and Liskov B., Proactive Recovery in a Byzantine-Fault-Tolerant System. *In Proc of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 273-287, 2000.

[11] Delporte-Gallet C. and Fauconnier H., Real-time Fault Tolerant Atomic Broadcast. *In Proc of the 18th Symposium on Reliable Distributed Systems*, pages 48-55, 1999.

[12] Dolev S., Self-Stabilization *MIT Press*, 2000.

[13] Gopal A. and Toueg S., Inconsistency and Contamination. *In Proc. of the 10th ACM Symposium on Principles of Distributed Computing*, pages 257-272, 1991.

[14] Hadzilacos V., and Toueg S., Fault-tolerant broadcasts and related problems. *In Sape J. Mullender, editor, Distributed Systems. Addison-Wesley*, chapter 5, pages 97-145, 1993.

[15] Halpern J.Y., Simons B., Strong R. and Dolev D., Fault-tolerant Clock Synchronization. *In Proc. of the 3rd ACM symposium on Principles of Distributed Computing*, pages 89-102, 1984.

[16] Hurfi n M., Mostefaoui A., and Raynal M., Consensus in Asynchronous Systems Where Processes Can Crash and Recover. *In Proc of the 17th Symposium on Reliable Distributed Systems*, pages 280-286, 1998.

[17] Le Lann, G., Proof-based system engeneering and embedded systems *Proceedings of the european School on embedded Systems*, Springer LNCS 1494, 1998.

[18] Lynch N., Distributed Algorithms. *Morgan Kaufmann*, 1996

[19] Pease M., Shostak R., and Lamport L., Reaching Agreement in the Presence of faults. *Journal of the ACM*, 27(2):228-234, 1980.

[20] Powell D., Group commuincation, *Communications of the ACM*, 39(4):50-53, 1996.

[21] Reischuk R., A New Solution for the Byzantine Generals Problem. *Information and Control*, Vol. 64, pages 23-42, 1985.

[22] Shirvani P., Saxena N., and McCluskey E., Software-Implemented EDAC Protection Against SEUs. *IEEE Transactions on Reliability, Special Section on Fault-Tolerant VLSI Systems*, Vol. 49, No. 3, pp. 273-284, 2000.

[23] Srikanth T.K. and Toueg S., Optimal Clock Synchronization *Journal of the Association for Computing Machinery*, 34(3), pages 626-645, 1987.