Dependable Ordering Policies for Distributed Consistent Systems

Matei Dobrescu, Manuela Stoian, Cosmin Leoveanu
General IT Directorate
Insurance Supervisory Commission
Bucharest, Romania
mdobrescu@csa-isc.ro

***Abstract*—A distributed system can be characterized by the fact that the global state is distributed and that a common time base does not exist. A linearly ordered structure of time is not always adequate for distributed systems and many authors have adopted a generalized non-standard model of time which consists of vectors of clocks. The paper present an improved algorithm where these clock-vectors are partially ordered and form a lattice. By using timestamps and a simple clock update mechanism the structure of causality is represented in an isomorphic way and the causal consistency is obtained. Finally, is presented the implementation of this new algorithm which allow to compute a consistent global snapshot of a distributed system for replicated services, where messages may be received out of order.**

Keywords- *temporal ordering; distributed systems; causal consistency; events structure; clock-vectors*

I. INTRODUCTION

An asynchronous distributed system consists of several processes without common memory which communicate solely via messages with unpredictable (but non-zero) transmission delays. In such a system the notions of global time and global state play an important role but are hard to realize. Since in general no process in the system has an immediate and complete view of all process states, a process can only approximate the global view of an idealized external observer having immediate access to all processes.

The fact that a priori no process has a consistent view of the global state and a common time base does not exist is the cause for most typical problems of distributed systems. Control tasks of operating systems and database systems like mutual exclusion, deadlock detection, and concurrency control are much more dificult to solve in a distributed environment than in a classical centralized environment. The great diversity of the solutions to these problems exemplifies many principles of distributed computing to cope with the absence of global state and time. To simplify the design and the validation of algorithms for asynchronous systems, one can try to simulate a synchronous distributed system on a given asynchronous systems, simulate global time (i.e., a common clock) and simulate global state (i.e., common memory), and then use these simulated properties to obtain the desired result. The first approach is realized by so-called synchronizers [1] which simulate clock pulses in such a way that a message is only generated at a clock pulse and will be received before the next pulse. The second approach does not need additional messages and the system remains asynchronous in the sense that messages have unpredictable transmission delays. This approach has been proposed by Lamport [2]. He shows how the use of virtual time implemented by logical clocks can simplify the design of a distributed mutual exclusion algorithm. The last approach was pursued by Chandy and Lamport in their snapshot algorithm [3], one of the fundamental paradigms of distributed computing. More recent approaches ([4], [5], [6], [7], [8], [9]) proved that to maintain the data consistency, the special synchronization operations are reduced to the minimum and are delivered using a global ordering algorithm. Almost all this algorithms assure a time complexity linear to network delays by utilizing timestamp estimations.

The organization of the informational flow as a linear sequence of discrete events is inappropriate for asynchronous distributed systems, where information is distributed and perception is delayed. Distributed environments require a distributed notion of time and a theory of distributed time provides a natural framework for solving problems in distributed environments.

While a synchronous distributed computing model provides processes with bounds on processing time and message transfer delay, which can be used to safely detect process crashes and allow consequently the non-crashed processes to progress with safe views of the system state, the asynchronous model is characterized by the absence of time bounds (this model is sometimes called *time-free* model). In these systems one can only assume an upper bound on the number of processes that can crash (let denote them by m) and consequently design protocols relying on the assumption that at least $(n - m)$ processes are alive, n being the total number of processes. In a distributed environment, the main drawback is the consensus problem, that has no deterministic solution when even a single process can crash. The *consensus* problem can be stated as follows: each process proposes a value, and has to decide a value, unless it crashes, such that there is a single decided value to be proposed for assuring validity. The impossibility of solving consensus has motivated researchers to find distributed computing models, weaker than the synchronous models but stronger than the asynchronous models, in which

 ISBN: 978-1-61208-149-6

consensus can be solved. In such a model we can describe the target in terms of distributed time, as a *timeslice* of logical simultaneity in the temporal relations expressed by a *time model*. The *timed asynchronous* model considers asynchronous processes equipped with physical clocks to ensure temporal ordering.

Resuming, one can say that the principles for temporal ordering in asynchronous distributed systems are: 1) Each machine maintains its own time; 2) There is no global shared clock; 3) Each target has a list of files on which it depends; 4) At the target one compare the associated timestamps; 5) If the target is older than some file that it depends on, then target is re-built.

A simple algorithm that respect these principles should ensure the following steps: 1) A time server maintains global notion of time; 2) Each machine periodically contacts time server asking for current global time; 3) Machine updates local time with global time. For implementation, the problem to solve is to associate with each event a logical timestamp T such that if $A \Rightarrow B$ then $T(A) < T(B)$, where $\Rightarrow$ means that event A precedes event B. Then, the ordering algorithm keeps for each i-th process a non-negative integer counter T_i, initially 0; when i-th process performs computation event, $Ti \leftarrow Ti + 1$ and when i-th process sends a message m, it computes $Ti \leftarrow Ti + 1$ and appends $T(m) \leftarrow T_i$ to m. Finally, when i-th process receives message m, $T_i \leftarrow \max\{Ti, T(m)\} + 1$. For event A at i-th process, one define $T(A) = T_i$ computed during A. A scheme for such a process is shown in figure 1 a. A better solution of Mattern is based on clock vectors [10], i.e. the i-th process keeps a vector T_i with n elements (see figure 1b). Each element $T_i[j]$ is a non-negative integer counter, initially 0. The following statements work: when i-th process performs any event, $Ti[i] \leftarrow Ti[i] + 1$; when i-th process sends m, it also appends $T(m) \leftarrow T_i$ to m; when i-th process receives m, it also computes $T_i[j] \leftarrow max\{T_i[j], T(m)[j]\}$ for each $j \neq i$; for event A at i-th process, define $T(A) = T_i$ computed during A such that $T(A) < T(B) = [\forall j: T(A)[j] \leq T(B)[j] \vee \exists j: T(A)[j] < T(B)[j]]$.

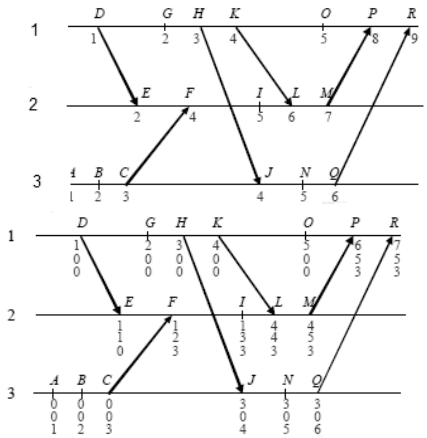

Figure 1. Ordered process using classical algorithms: a) Lamport; b) Mattern

While in some sense the snapshot algorithm computes the best possible attainable global state approximation, Lamport's virtual time algorithm is not that perfect. In fact, by mapping the partially ordered events of a distributed computation onto a linearly ordered set of integers it is losing information. Events which may happen simultaneously may get diferent timestamps as if they happen in some definite order. For some applications (in our case the objective was the ordering of events in the alerts flow of an emergency system) this defect is noticeable. In this paper, we aim at improving Lamport's virtual time concept, considering that a partially ordered system of vectors forming a lattice structure is a natural representation of time in a distributed system. In this non-standard model of time all events which are not causally related are considered simultaneous, thus representing causality in an isomorphic way without loss of information.

II. EVENT STRUCTURES

In an abstract setting, a process can be viewed as consisting of a sequence of events, where an event is an atomic transition of the local state which happens in no time. Hence, events are atomic actions which occur at processes. Usually, events are classified into three types: send events, receive events, and internal events. An internal event only causes a change of state. A send event causes a message to be sent, and a receive event causes a message to be received and the local state to be updated by the values of the message.

Events are related: Events occurring at a particular process are totally ordered by their local sequence of occurrence, and each receive event has a corresponding send event. Formally, an event structure [11] is a pair $(E;<)$, where E is a set of events, and „<" is a partial order on E called the causality relation.

Event structures represent distributed computations in an abstract way. For a given computation, $e < e'$ holds if one of the following conditions holds:

1) e and e' are events in the same process and e precedes e',

2) e is the sending event of a message and e' the corresponding receive event 3) $\exists e''$ such that $e < e''$ and $e'' < e'$.

The causality relation is the smallest relation satisfying these conditions.

A *consistency mechanism* guarantees that operations will appear to occur in some ordering that is consistent with some condition. Most of the research on this subject addressed *strong* consistency conditions like *sequential consistency* and *linearizability*. These conditions guarantee that operations appear to be executed in some sequential order that is consistent with the order seen at individual sites. Unfortunately, supporting either sequential consistency or linearizability requires a non-negligible cost. A way around this cost is to define conditions that provide weaker guarantees on the ordering of operations, and can be

efficiently implemented. These conditions can be roughly classified into two categories: *weak* and *hybrid* conditions. Weak conditions provide very little guarantee on the relative ordering of events at different processes. These conditions admit very efficient implementations, but they are too weak to support conventional methods for concurrency control. Hybrid conditions distinguish between two types of operations, *strong* and *weak*. Strong operations appear to be executed atomically, in some sequential order that is consistent with the order seen at individual processes. The only guarantees provided for weak operations are those implied by their interleaving with strong operations. When the *consistency mechanism* offers *hybrid* conditions, one can define the synchronization as hybrid too.

Let's now consider that a model of a distributed consistent system (DCS) system is composed of a finite set of sequential processes $P_1, P_2, \ldots P_n$, one for each node. The processes interact with the application program at the same node using *call* and *response* events. The processes $P_1, P_2, \ldots P_n$ interact through a finite set of $x \in X$ shared objects via *message-send* and *message-receive* events. The process P_i can be also modeled as an automaton with states and a transition function that takes as input the current state and a call or message-receive event, and produces a new state, a set of response events and a set of message-send events.

A *history* of a process describes what steps the process takes and times they occur; it must satisfy certain "consistency" conditions. An execution of a set of processes is a set of histories, one for each process.

An *execution* of a set of processes is a set of histories, one for each process, together with a one-to-one correspondence between the messages sent by P_i to P_j and the messages received by P_j from process P_i. We use the message correspondence to define the *delay* of any message in an execution to be the real time of receipt minus the real time of sending. The execution is admissible if the delay of every message is less than d, for fixed $d \geq 0$, and for every P_i, at any time at most one call at P_i is *pending*.

Every object is assumed to have a *serial specification*. The specification defines a set of *operations*, which are ordered pairs of call and response events, and a set of *operations sequences*, which are the allowable sequences of operations on that object. As an example, in the case of a read/write object, the ordered pair of events *[Read$_i$ (x), Return$_i$ (x,v)]* forms an *operation* for any process P_i, object x, and value v, i.e. (*v, (r(x,v))*) as does *[Write$_i$ (x,v), Ack$_i$ (x)] (w(x,v))*.

A. Legal Operations in Distributed Consistent Systems

An *execution history* of a DCS is a partial order $\widehat{H} = (H, \rightarrow_H)$, formally:

$H = \bigcup_i h_i$

$o_1 \rightarrow_H o_2$ if:

1) $\exists P_i : o_1 \rightarrow_i o_2$ (in that case $\rightarrow_H$ is called a *process-order* relation
2) $\exists w(x,v), r(x,v)$ such that $w(x,v) \in o_1$ and $r(x,v) \in o_2$ (in that case $\rightarrow_H$ is called a *read-from* relation)
3) $\exists o_3 : o_1 \rightarrow_H o_3$ and $o_3 \rightarrow_H o_2$ (transitivity)

Let's now consider a history $\hat{H}$. Informally, an operation $o \in H$ is legal if it does not read overwritten values, i.e. the legality of an operation (causal dependency) is defined as follows:

Definition 1. An operation *o* is legal if $\forall r(x,v) \in o : \exists o'$ such that:

$o' \rightarrow_H o$ (*o'* precedes *o*)

$w(x,v) \in o'$ (*o'* is the operation that wrote *v* into *x*)

$\forall o''$ such that $o' \rightarrow_H o'' \rightarrow_H o : w(x) \notin o''$ (there is no overwriting operation)

Definition 2 A history $\widehat{H} = (H, \rightarrow_H)$ is *causally consistent* if, for each process P_i there exists a linear extension of $\widehat{H}$ in which all operations issued by P_i are legal. In other words, the order of all operations of P_i maintains causal dependency of the operations .

As an example let see Figure 2, where appears the model of an execution that is only possible in a causally consistent system. This shows processes P_i, P_j and P_k modifying concurrently different objects. The operation $o_{i,1}$ updates object *y* at the same time that $o_{j,1}$ updates object *x*. The second concurrent update occurs when $o_{j,3}$ writes to object *x* and $o_{k,4}$ writes to object *y*. P_k is able to read the update of P_j in $o_{k,1}$ but the update *w(y,1)* from P_i is not seen until $o_{k,3}$. These executions are acceptable because the two objects are written concurrently and hence P_k makes no assumptions about which object will be updates first. The model in Figure 2 shows that the execution $\hat{H}2$ is not serializable since there does not exist a linear extension of $\hat{H}2$ in which all operations are legal. However, $\hat{H}2$ is causally consistent as there exists, for each process P_i, a linear extension including all write operations plus all read operations issued by P_i , in which all operations are legal.

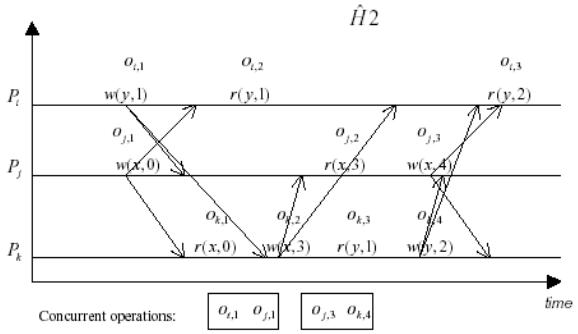

Figure 2. Causal consistency executions

The causal ordering of messages deals with the notion of maintaining the same causal relationship that holds among "message send" events with the corresponding "message receive" events. Events that occur at a single site are ordered in time in the normal way. Informally, an event *a* at a site *s* is ordered temporally after an event *b* at site *t* if, and only if, there is a sequence of messages, the first one originating from site *t* after the event *b*, the next message being sent from the destination site of the first message after the first message is received there, and so on, with the last message being received at site *s* before the event *a*. The following execution examples show how inconsistencies can appear if the system does not ensure causal synchronization.

A conflict-free run is depicted in Figure 3. This is normally the case, where due to the relatively low network roundtrip times are small compared to user interaction intervals. In this example, Process 1 modified and unselected the object (released the lock over the object) before Process 2 had sent a select message: Process 2 started without waiting for any synchronization or acknowledgement messages.

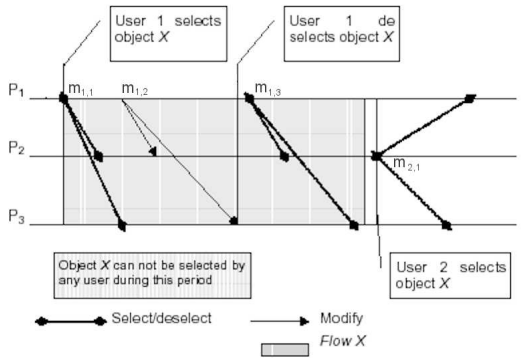

Figure 3. Normal execution with no conflict.

In Figure 4 it is presented a case when the system does not provide any causal consistency mechanism. P_2 received the deselect message from P_1 and immediately selected the same object (message $m_{2,1}$) before P_3 received the previous deselect message from P_1. This case may occur if packets travel between sites through different paths, and their roundtrip times vary noticeably. If P_2 modifies its local copy before $m_{1,3}$ arrives to P_3, the database becomes inconsistent. The last occurs because there is no causal synchronization.

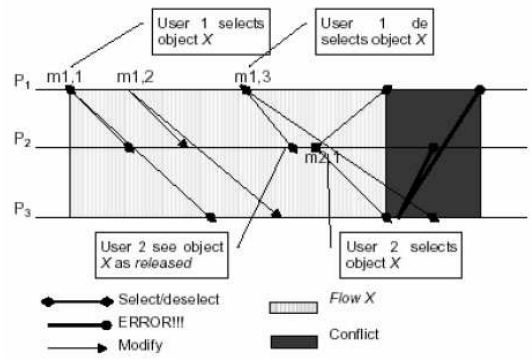

Figure 4. Execution with conflict and no causal synchronization.

The execution diagram depicted in Figure 5 shows the result of applying hybrid synchronization to the previous example. P_3 does not start a flow, it does not send any update message, until it receives the message sent by $m_{1,3}$. Therefore, P_2 cannot start any object processing until the select strong select operation is globally ordered at every site.

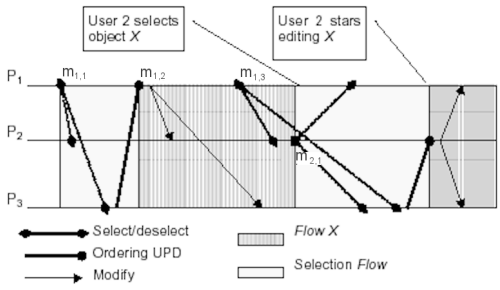

Figure 5. Causal synchronization

Causal consistency is attractive because not only it can meet the sharing needs of many applications but it can also be implemented efficiently. It is possible to complete send and receive accesses to causally consistent objects without synchronisation among processes (or sites) that store copies of the objects. This can lead to a scalable architecture because coordination among a large number of nodes is not necessary with causally consistent shared objects. Among these, service-oriented architectures (SOA) are typical for the necessity to assure a dependable global ordering.

III. AN ALGORITHM FOR DEPENDABLE GLOBAL ORDERING OPERATIONS IN SOA

This algorithm is an improvement of the classical ordering algorithms based on timestamps. As framework We considered a service-oriented architecture (SOA), which actually is a collection of services. A service is a function that is well-defined and does not depend on the context or state of other services. These services communicate with each other in the same way as interact processes in a distributed system. Services is becoming a platform for information interaction between applications.

Our approach can maintain the data consistency among multiple service replicas while we still guarantee the loose coupling and location transparency characteristics among the service replicas. In the informational flow, consistent operations are classified as either strong or weak. Informally, flows consistency guarantees two properties:

1) Strong operations appear to be executed in some sequential order.

2) If two operations are invoked by the same process and one of them is strong, then they appear to be executed in the order they were invoked.

Each replica of the editor holds a local copy of the entire memory, a local timestamp counter and an array that keeps conservative about the values of all other timestamp

 ISBN: 978-1-61208-149-6

counters in the system. A weak operation is executed instantly on the local copy of the object. In case of writes, update messages are sent to all other processes, which update their local copies of the memory upon receiving these messages. Timestamps are used to enforce global ordering on the strong operations. Strong operations are timestamped with the local timestamp counter, and a message is sent to all processes; the initiating process then increments its local timestamp counter by 1. The execution of any strong operation is postponed until the timestamp of that operation is smaller than all the estimated timestamp counters of the system. If more than one strong operation can be executed together, they are executed according to their timestamps in increasing order.

The algorithm guarantees that if process P_i estimates P_j counter as x, then the local timestamp of P_j is at least x (that is, the estimate is conservative). This implies that all strong operations ever invoked by P_j bearing timestamp smaller than x have arrived at P_i, and ensures that all strong operations are executed in the same order and that weak operations that were invoked later are also executed later.

We assume a system of n processes, connected by an interconnection network, each maintaining a local copy of the entire database. Each process P_i has a local timestamp counter, lts_i, initially 0, and an array ts_i such that ts_i[j] contains P_i's estimate of lts_i. Weak operations are executed locally and instantly. If a weak operation is a write of v to object x, then *update* messages are broadcast to all processes (an *update* message includes the new value v to object x to be updated). A process that receives an update message of v to object x, updates its copy of object x with v. For any strong operation (*select* or *deselect* messages), a strong-op message is sent to all other process; this message not only contains update information (path of the object to be selected) but also a timestamp *lts*. Process P_i suspends the execution of a strong operation with timestamp *ts*, until it knows that the counters are at least $ts+1$. When several pending strong operations may be executed, they are executed according to their timestamps and *ids* in increasing order.

Executing a strong *select* operation at process P_i is done by updating the list of selected objects in the local copy. If object x specified in the select message is marked as already selected by another operation, the operation is ignored and no action is taken. Otherwise, object x is added to the local selection list. Executing a strong *deselect* operation at process P_i is done by deleting the object x specified in the message from the selection list.

Process P_i increases its timestamp in each of the following cases:

1) After P_i sends a strong-op message to all processes.
2) After P_i receives a strong-op message with timestamp equal to lts_i and for all j, $ts_i[j] \geq lts_i$.
3) A strong operation *with* $ts=lts_i-1$ was executed in P_i, and there exists k such that $ts_i[k] \geq lts_i$.

In the last two cases, a *ts-update* message is sent to all other processes.

Let's now discuss how the proposed algorithm offers dependable solutions. A crucial issue encountered in distributed systems is the way each process perceives the state of the other processes. To that end, the proposed model provides each process p_i with three sets denoted $idle_i$, $active_i$ and $uncertain_i$. The only thing a process p_i can do with respect to these sets is to read the sets it is provided with; it cannot write them and has no access to the sets of the other processes. These sets, that can evolve dynamically, are made up of process identities. Intuitively, the fact that a given process p_j belongs to one of the three sets provides p_i with some hint on the current status of p_j . More operationally, if $p_j \in idle_i$, p_i can safely consider p_j as being crashed. If $p_j \notin idle_i$, the state of p_j is not known by p_i with certainty: more precisely, if $p_j \in active_i$, p_i is given a hint that it can currently consider p_j as not crashed; when $p_j \in uncertain_i$, p_i has no information on the current state (crashed or active) of p_j. The specification of the sets $idle_i$, $active_i$ and $uncertaini$, $1 \leq i \leq n$, is the following:

S1 - Initial global consistency. Initially, the sets $active_i$, $idle_i$ and $uncertain_i$ of all the processes p_i are identical. Namely, for $t = 0$, $\forall\ i, j$: $state_i(t) = state_j(t)$, where *state* is *active*, *idle* and *uncertain* respectively.

S2 - Internal consistency. The sets of each p_i define a partition $idle_i(t) \cup active_i(t) \cup uncertain_i(t) = \Pi$, $\forall\ i,t$. and any two sets in $idle_i(t)$, $active_i(t)$ and $uncertain_i(t)$ have an empty intersection.

S3 Consistency of the $idle_i$ sets: an $idle_i$ set is never decreasing, i.e. $idle_i(t)\ \forall\ idle_i(t+1)$, $\forall\ i,t$

S4 Consistent global transitions. The sets $idle_i$ and $uncertain_j$ of any pair of processes p_i and p_j evolve consistently. More precisely, $\forall\ i, j, k, t_0$ we have $(p_k \in active_i\ (t_0)) \cap (p_k \subseteq idle_i\ (t_0 + 1)) \Rightarrow \forall\ t_1 > t_0 : p_k \notin uncertain_j\ (t_1)$.

As we can see from these specifications, at any time t and for any pair of processes p_i and p_j, it is possible to have $active_i(t) = active_j(t)$ (and similarly for the other sets). Operationally, this means that distinct processes can have different views of the current state of each other process. The rules [S1-S4] define a distributed computing model that satisfies the strong consistency property. That property provides the processes with a mutually consistent view on the possibility to detect the crash of following a given process. More specifically, if the crash of a process p_k is never known by p_i (because p_k continuously belongs to $uncertain_i$), then no process p_j will detect the crash of p_k (because $p_k \in idle_j$). Conversely, if the crash of p_i is known by p_j, the other processes will also know it.

IV. THE IMPLEMENTATION OF THE APPLICATION

We will present an application that uses the proposed ordering algorithm in a distributed system for emergency management. The main objective is the consistent synchronization of alerts. That implies to have complete information about the temporal dimension of alerts, compatibility with the alert standards and with the software and hardware resources running the application. The

 ISBN: 978-1-61208-149-6

participants in the alert process are computers acting as nodes in a network which communicate using standard ISO-OSI protocols. The application is realized in Java, in order to be supported on a large set of hardware platforms.

The application is composed from several classes, as follows:

AlertNode – is the class for instantiation of the matricial logical clock of the node that contains the function *main()* which launch the client and server execution threads. In the initial state one must specify the node ID, the number of active nodes in the whole network and the port of the server which take the alert.

AlertServerThread – is the class that implements the several able to receive alerts. For each connection a dedicated thread is created, so many clients can be simultaneously serviced.

AlertProtocol – is the class that implements the communication protocol between the server and the alert client. This class contains the function *readAlert*, that initiate the class *CAPHandler*, which parses the client alert in the Common Alerting Protocol (CAP) XML format. When the alert is received, one launch the method *receiveAction* of the matricial clock that implements the clock logic.

MatrixClock – is the class that implements the matricial logical clock.

AlertClientThread – is the class which allows to transmit alerts from client to server, only in CAP format.

As an example, let now consider the following scenario, as shown in figure 6:

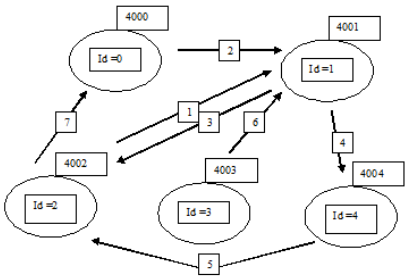

Figure 6. An alert secenario

The ellipses represent the network nodes, each node having an unique identifier. In the rectangle above the node appears the number of the listening port. The arrows represent the direction of the transmitted alert, and the associated numbers represent the sequential order for alerts transmission. On each node is running a software agent with double functionality (client and alert server). The alert is connection oriented, using TCP stream sockets. A socket is unique identified by an IP (node address) and a port (which directs the data to destination).

When a node has to transmit an alert to other node, the server try to connect the destination node through a separate execution thread), but it maintains the idle state in order to accept other connections also. The client is addressed by a command line, on the associated port. But it is noticeable that the client can interrogate periodically a data base were are registered the out of limits parameters, without a special command of the server, and can decide himself is another node must be alerted. Figure 7 shows the values of the timestamps at the matricial clocks, for the first steps of the scenario depicted in figure 6. At the end of the process the clocks have the value of the arrows end.

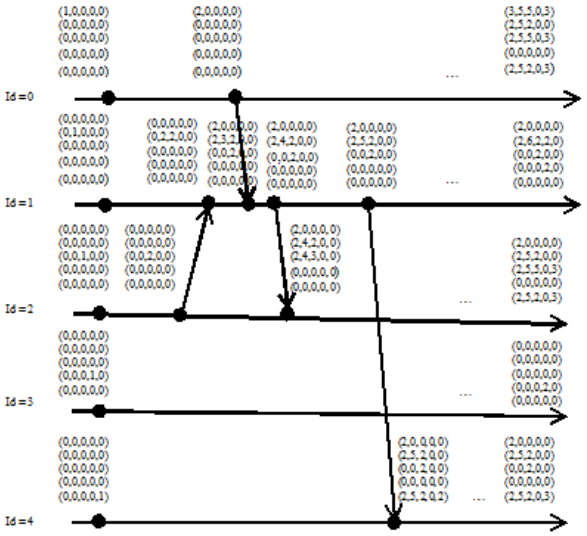

Figure 7. Alerts flow and the matricial clocks of the nodes

The main contribution of the proposed scheme is the correlation of alerts in emergency systems, introducing as a new element in the classical Lamport algorithm a matriceal clock which acts as a component of the advertising protocols structure. The efficiency of this mechanism is improved by adding a fault detection component of the timestamp assignment that verifies if each secondary vector of the matrix is smaller than the principal vector of the current node.

The algorithm imposes to send dedicated messages for the matriceal clock refresh, at the same frequency as that of the information messages, if in a specified interval a process does not succed to perform a send-receive operation.

V. CONCLUSIONS

This paper proposed an improved global ordering algorithm for dependable distributed computing, that encompasses both the synchronous model and the asynchronous model. The algorithm guarantees the order of messages delivery to the application and respect temporal and causal relationships. In this aim the strong operations are timestamped with a local timestamp counter, and a message is sent to all processes. If more than one strong operation can be executed together, they are executed according to their timestamps and in increasing order. We have chose to focus on the distinction between performing a data operation locally at a process, based on its local state, and performing an operation that requires communication between processes before the control can be returned to the

 ISBN: 978-1-61208-149-6

application. When collaboration involves communicating via a single or multiple flows, causal relationships among messages sent over the flows must be maintained to preserve the context in which a message is sent.

Other contributions which derive from the conceptual framework can be summarized as it follows: the implementation and testing of a general protocol for data replication in a distributed architecture; a scheduler for operations of a collaborative process; the definition of a formal consistency criteria of the flows framework; the classification of strong and weak operations that allows the implementation of this consistency criteria; the definition of the form that a process state perceives each other's states by accessing the contents of three local non-intersecting sets, (*uncertain*, *active*, and *idle*). The proposed system has been implemented in JAVA and tested over a set networked LINUX workstations, equipped with QoS capabilities

Future work will be oriented on: strong operations' generalization for different type of operations, specially those operations that modify the topology of a scene tree, i.e. addition or deletion of nodes; the implementation of a policy that allows to support latecomers and early leaving in to the distributed system; the implementation of a multicast protocol for supporting many users simultaneously; the evaluation of the benefits of the admission control policies with respect to the media quality of the serviced clients, the average latency time, and the throughput of the system.

REFERENCES

[1] C. J. Fidge, „Timestamps in Message-Passing Systems that Preserve Partial Ordering". In *Proceedings of 11th Australian Computer Science Conference*, pp. 56-66, 1988

[2] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Comm. of the ACM*, 21(7), pp. 558-565, 1978.

[3] K. M. Chandy and L. Lamport, „Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Transactions on Computer Systems*, 3(1), pp.63-75,1985.

[4] H. Kopetz, A. Ademaj and A. Hanzlik, „Combination of clock-state and clock-rate correction in fault tolerant distributed systems", *Real-Time Systems*, Vol. 33, pp.139-173, 2006

[5] Yang, J., Q. Zhang and N. Gu (2006) A Consistency Maintenance Approach in Replicated Services, *Proc. of the Sixth IEEE Int. Conf. on Computer and Information Technology*, pp. 248 – 258

[6] A. Hanzlik, „SIDERA - A Simulation Model for Time-Triggered Distributed Real-Time Systems*", Int. Review on Computers and Software* (IRECOS), Vol. 1, N. 3, pp. 181-193, 2006

[7] R. Dobrescu and M. Dobrescu, A "flows consistency" model for message ordering in collaborative distributed systems, *13th IFAC Symposium on Information Control Problems in Manufacturing*, 2009

[8] V. Cholvi, A. Fernández Anta, E. Jimenez, P. Manzano, M. Raynal. "A Methodological Construction of an Efficient Sequentially Consistent Distributed Shared Memory". *The Computer Journal, 53(9),* pp.1523-1534, 2010

[9] R. Jimenez-Peris, M. Patiño-Martinez, D. Serrano, J. Milán and B. Kemme, „Leveraging the Scalability and Availability of Replicated Databases with Autonomic Capabilities", *3rd Int. Conf. on Autonomic Computing and Communication Systems*, 2009.

[10] F. Mattern, " Virtual Time and Global States of Distributed Systems", *Proceedings of the Parallel and Distributed Algorithms*, pp.215-226, 1989

[11] S. Gorender, R. Macedo and M. Raynal, "A Hybrid and Adaptive Model for Fault-Tolerant Distributed Computing", *Proceedings of the Int. Conf. on Dependable Systems and Networks*, pp.412-421, 2005

 ISBN: 978-1-61208-149-6