An Algorithm Providing Fault-Tolerance for Layered Distributed Systems

M. Taghelit, S.Haddad and P. Sens

Université Pierre et Marie Curie
C.N.R.S. MASI
4, Place Jussieu
75252 Paris Cedex 05, France
e-mail: taghelit@masi.ibp.fr

Abstract
This paper presents a new approach for the fault-tolerance in layered distributed systems. We develop the dynamic regeneration method in which faulty software components are dynamically regenerated. In contrast to other techniques which duplicated critical components, this method does not increase the complexity of the system and tolerates an unlimited number of failures. To apply this technique, we develop a method for the design of reliable systems. We transform an initial software architecture in a model which encloses homogeneous elements. Our study relies on the OSI standard model defined within the ISO organization. So, from an initial layered distributed architecture, we exhibit a homogeneous communication chain. The building and the maintenance of this chain in an unreliable environment is achieved by an algorithm using dynamic regeneration.

Key words: fault-tolerance, dynamic regeneration, distributed systems, layered systems, end-to-end communication.

1. Introduction

Distributed systems provide new opportunities for developing high-performance applications; at the same time, because of dependency of the components, such systems are particularly fragile: one component failure may imply all the system failure. So it is essential to have a fault-tolerant management. Several methods based on the redundance techniques exist [7][1]. These methods prevent the failure of critical elements by duplicating them in several copies. There are two basic kinds of redundancy [6]: the passive one where copies run only if the element fails and the active one where all copies run in parallel with the element. These techniques imply a great overhead of the system and tolerates a limited number of failures (this number is proportional to the number of copies).

We propose a new solution that offers an optimum degree of tolerance for a low cost: the dynamic regeneration. Instead of duplicating the elements, they are regenerated in case of failure.

From the OSI standard model defined within the ISO organization, we generalize this kind of architecture to any communication chain. A communication chain allows a dialogue between an initial and a terminal entities through inner ones which relay messages.

The aim of the proposed algorithm is to build and preserve the communication chain in an unreliable environment. The entities of the chain are dynamically generated unlike the other techniques where elements have a static existence. When a failure of an entity is detected, we regenerate a new entity and integrate it into the chain.

In the first part, we describe how to obtain a communication chain from the OSI model. In the second part, we outline the different methods providing fault-tolerance and we introduce the principle of the dynamic regeneration. The third part informally presents the algorithm and the fourth one describes it. Finally, the last part shows the correctness of the algorithm.

2. Functional scheme for layered distributed systems

The ISO organization defines an architecture for communicating opened systems (the OSI standard) [3]. This hierarchical model is composed of seven layers. Each layer has a *specific functionality* and the communication between two layers of the same level is done through the lower layers. Although the OSI standard does not impose a specific localization of the layers, current systems locally implement all of them.

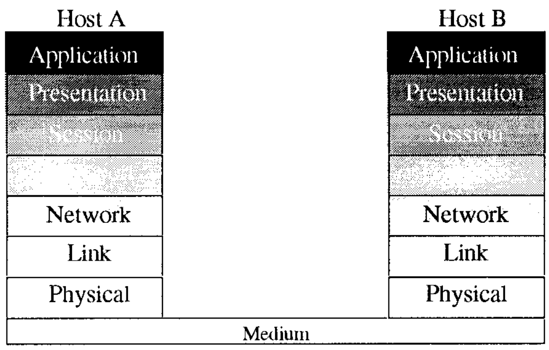


figure 1 : The OSI model

We would like to make this kind of architecture reliable. For that, we modify the basic functional scheme. The first step consists in unlayering the architecture. The notion of layer disappears and this new architecture, even though identical as the old one, makes a chain composed of communicating entities. This unlayering shows three kinds of entities: the initial entity at the beginning of the chain, the ending entity at the end of the chain and the inner entities which each has a predecessor and a successor. All functionalities of one layer are included in an entity but we generalize the initial model by making *no assumption about the localization.*

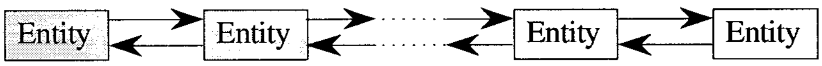


figure 2 : Unlayering of the model

The global architecture being defined, we will proceed to the functional division of entities. All the entities have three functionalities: the receipt and sending of messages *being homogeneous ,for all the entities* and the processing which is *heterogeneous.* We bring together these functions in two specialized modules: one module of communication that carries out the communication tasks and another one, processing module, that carries out the processing. The communication module becomes the communication interface between an entity and the others.

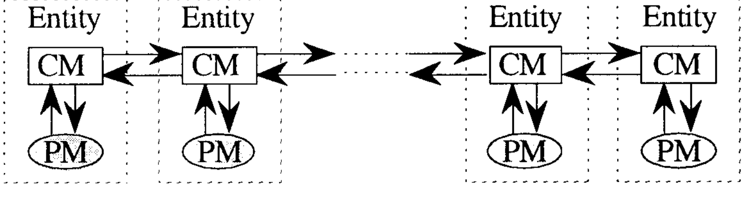


figure 3 : Functional division

Communication modules (CM) compose an *homogeneous communication chain,* since each module has the same functionality (sending/receipt). The processing modules remain heterogeneous and each of them locally communicates with the communication module of the same entity. Thus, afterwards we will pay attention to ensure communication between entities and this is done by preserving the communication chain in case of failures.

Note that the modularity of an entity implies a limited propagation of the failures. The failure of the processing module does not mean the breaking of communication. Moreover, the separation of functionalities allows diagnosis more precise of failures and thus a more efficient recovery.

figure 5: Communication chain

3. Tolerance by dynamic regeneration

Generally, techniques providing fault-tolerance use redundancy methods [7]. These methods duplicate a basic element in several copies. In this way, they prevent the failure of the basic element called the primary.
There are two kinds of redundancy techniques: the active redundancy where copies are actives in parallel with the primary element, and the passive redundancy where copies are active only if the primaly element fails.

3.1. Active redundancy

Each copy receives the same input and do the same tr eatment. Only one result is taken into account [8]. To achieve this, two different methods are currently used.

In the first one, a vote mechanism chooses one result from among all of those produced by the redundant elements. A disagreement between elements provides a failure detection. The simple majority is the most used vote technique [6].

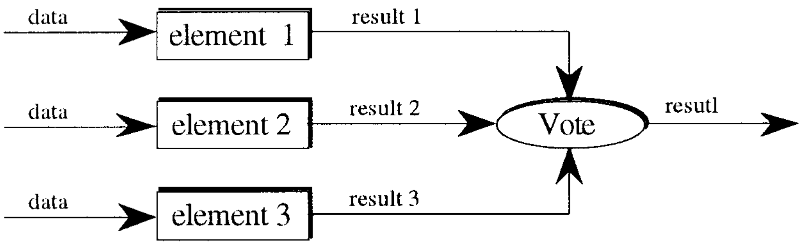


figure 6: The vote mechanism

In the second method, only the result of the primary element is considered. If the primary module fails, it is replaced by one of its copies. The recovery is very simple, it only consists in taking new results on the output of the chosen copy.

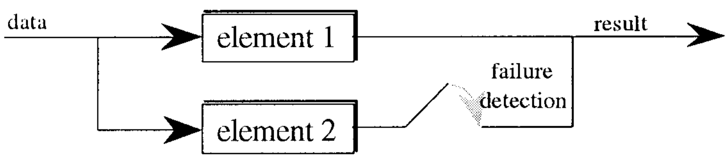


figure 7: ....

The vote technique allows a limited degree of tolerance (i.e. the number of element failures): for a number of n redundant elements, a majority vote tolerates only n/2 failures. Moreover, the vote component can itself fail. This kind of failure is catastrophic, and a replication of the vote component is recommended.

The second technique allows a greater degree of tolerance than the vote. For n redundant elements, it tolerate n-l failures. But this method does not prevent a faulty behaviour of the primary element. Contrary to the vote technique, erroneous results produced by the primary element will be taken into account.

These methods lead to a great overhead of treatment. If the degree of redundancy is n (i.e. there are n redundant elements) there must be n treatments for the production of one issue. Moreover, each redundant element has a failure probability and so the global failure probability is greater. But, active redundancy provides a fast recovery failure, because it asks no state's restoration (all the time, copies do the same process and are in the same state). This method is currently used for reliable real time systems where the recovery time is bounded [10].

3.2. Passive redundancy

A redundant element can only start running when the primary element is failing. All the time, only one element is active.

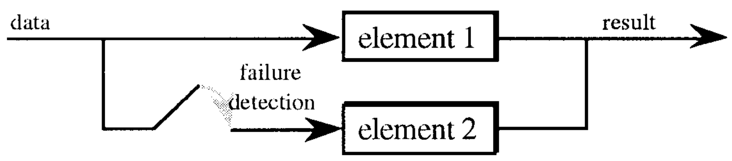


figure 8: passive redundancy

When a copy replaces the faulty primary element, a previous state of the primary element must be restored. Passive redundancy needs also a checkpoint management. This kind of technique is expensive in processing time and space. Moreover, like the active redundancy, the failure probability of redundant modules is non-null. Generally, this technique has a lower cost than active redundancy and it is often preferred to this one [9].

The main drawbacks of redundancy techniques is the increase of the system's complexity and the limited fault-tolerance degree. We propose a new technique that solves these problems while profiting of the characteristics of our architecture and of the tunctional division: the dynamic redundancy.

3.3. Principle of the dynamic regeneration

The entities are no longer duplicated but generated dynamically after their failure. The generated element has no existence and is integrated in the system only after one failure. This element has the same tunctionality of the faulty element. This approach implies the following points:

- **.**The system has as much element as a fault-free system. So, it is less complex than a system using redundancy techniques.
- **.**The overhead is only related to the fault detection.
- The fault tolerant degree is not limited.
- The generated element have no localization constraint unlike the redundant elements which have a fixed localization. A new module may be dynamically generated on an unfaulty host.

More precisely, the treatment of a failure is composed of three steps: the detection, the regeneration of a new element and its insertion in the chain.

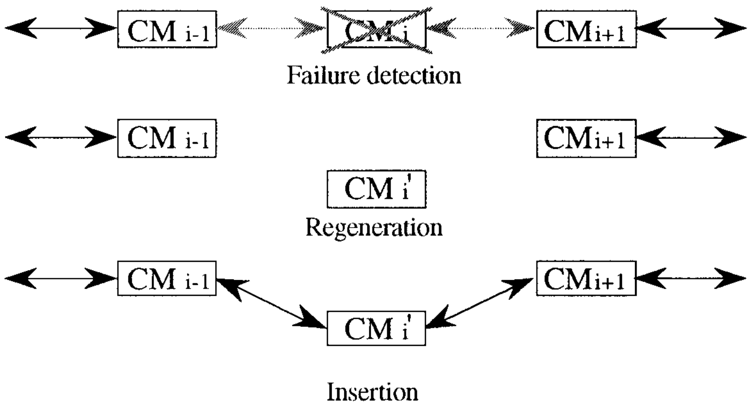


figure 9 : The dynamic regeneration

Each module has to supervise the one it has generated (its successor) and to regenerate it in case of failure. This ensures a decentralized control which answers our tolerance and performance requirements.

4. Implementation of the dynamic regeneration

4. 1. Targets of the algorithm

- The building of the communication chain

The building of the chain consists in generating a finite set of communication modules such that each inner module has a predecessor and a successor. Unlike the other systems where the elements are static, the modules are generated dynamically. On the other hand, an initialization step is required. This step consists to generate the first module which is on user's initiative.

- Maintenance of the communication chain in an unreliable environment

If one or several modules fail, the chain breaks and no communication between the initial and ending entities is possible. To keep end-to-end communication, it is necessary to replace the faulty modules. This is carried out by the dynamic regeneration of the faulty modules.

4.2. Principles of the algorithm

We want to construct and keep a chain of N modules, each module communicating with the adjacent modules. There are three principles :

- <u>Activation principle</u>: Each non-ending module carries on the building of the chain.

Each module with no successor and which is not the Nth one generates a successor. This allows to carry on the building of the chain if it is not complete and to rebuild it if modules fail.

- <u>Knowledge principle</u>: Each module knows all its successors

When a module fails, it is regenerated according to the activation principle. Then, this new module needs to be attached to the predecessor (the one who generated it) of the faulty module and to its successor. The regenerating module must know its two immediate successors so as to give to the regenerated module the informations necessary to attach itself to the chain.

But that is not enough in case of multiple failures of adjacent modules. In that case, each regenerated module has to regenerate a successor until a valid successor has been found. So, to restore the chain whatever the number of faulty adjacent modules, each module has to know all its successors.

If the regenerated module is the initial one, it cannot receive informations from its successor by its predecessor since the last one does not exist. In order to avoid the building of a new chain, the initial module must keep in stable memory informations of its successors. These informations will be communicated to it, if it is regenerated after a failure.

- <u>Purge principle</u>: Each non-initial module with no predecessor destroies itself after a finite time

When the predecessor of a module fails, this module must be eliminated if no new module replaced the predecessor after a finite time. We avoid, like that, the creation of "parasite" subchain.

- <u>Suspicion principle</u>: The validity of the chain is periodically tested

Periodically, each module supervises its successor. This mechanism allows to detect faulty modules.

More precisely, the detection is based on an acknowledgement mechanism. Each module periodically transmits a control message to its successor so as to verify that it is still valid. If after a while *t* no answer is given, the successor is considered faulty. This mechanism can bring false detections which rate is proportional to *t*.

4.3. Few executions

- The chain building

The *activation principle* allows the chain building. The chain building of N long is done in N steps. Each step corresponds to a generation of a new module and its joining to the chain already built. This joining is achieved when all its predecessors know its existence. The new module communicates its identity to its predecessor (regenerating module) which sends it back to its predecessor and so on until the initial module receives it. Then, the initial module transmits an acknowledgement which is retransmitted from successor to successor until it arrives to the new module. It is only at this moment that the new module is joined to the chain. This process is repeated until the joining of the N-th

- Chain maintenance in case of failures

The principles of *suspicion, activation* and *knowledge* allow the chain maintenance. Assume

that two modules, CMi and CMi+l, fail at the same time. The CMi-l module detects by the *suspicion* principle the CMi module failure and regenerates a substitute module CMi' by the *activation* principle. The substitute CMi' module detects the CMi+l failure and follow the same process. That is, regeneration of the substitute CMi+l' module to which it gives its own identity and those of CMi+2 to CMN modules according to the *knowledge* principle. Then, CMi+l' module connects itself to CMi+2 module and thus restores the chain. This example of situation is shown in figure 10.

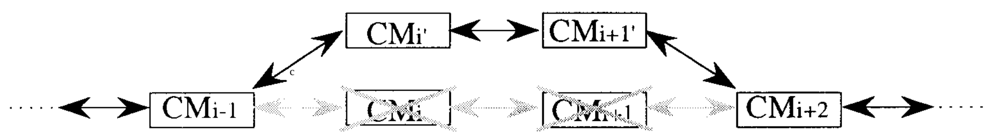


Figure 10: Recovery from multiple failures

With the *knowledge* principle, it is possible to restore a communication chain, whatever the number of faulty modules. The failure of the first communication module (CMl module) is particular, because its detection and regeneration are done by the user.

- creation and destruction of a parasite sub-chain

We saw that the probability of false detection is non-null. In case of false detection, a new module is regenerated and will be reconnected to the successor of the module supposed to be faulty. The supposed faulty module becomes isolated (with no successor and predecessor). It constitutes a sub-chain which can grow up by following the *activation* principle. Such a case is shown in figure 11.

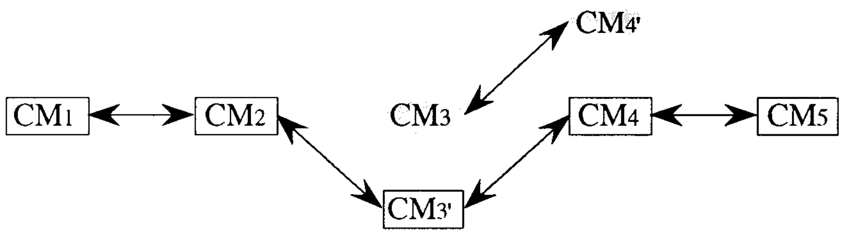


Figure 11: Creation of a sub-chain from an isolated module (CM3)

This situation is not the only one where a sub-chain is created. Several adjacent modules which cannot be accessible can also constitute a sub-chain. The destruction of parasite sub-chain is achieved by the *purge* principle.

4.4. Definition of the algorithm

4.4.1. Description of a module

The states transitions of a communication module are set up by the reduced graph of figure 12. For the sake of simplicity, only the main transitions are described. In this graph, we do not specify neither various recovery cases (during or after complete building of the chain) nor the fact that a module can fail at any moment and thus in any state.

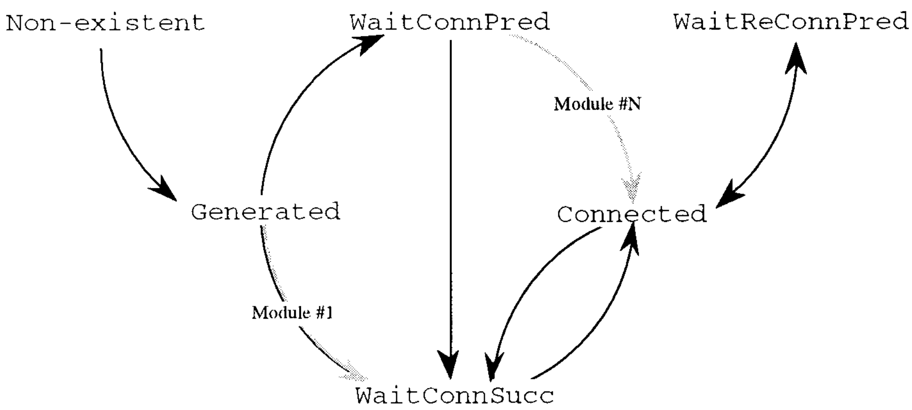


Figure 12: States transitions of a communication module

Non-existent: the module does not exist.
Generated: the CMi module has just been generated.
WaitConnpred: the CMi module gave its identity to the module which generated it (CMi-l module) and waits the acknowledgement message.
WaitConnSucc: the CMi module generated a CMi+l module to which it gave its identity, and waits for the reception of the CMi+l identity.

Connected: the CMi module received the identity f'orm the module it has generated (CMi+l module).
WaitReConnPred: the CMi module detected its disconnection with its predecessor, it waits a reconnection message.

Each module has a set of variables including its level in the chain and a **knowledge** vector which contains all its successors' identities.

4. 4 .2. Description of message types

The various message types that are exchanged between communication modules are:

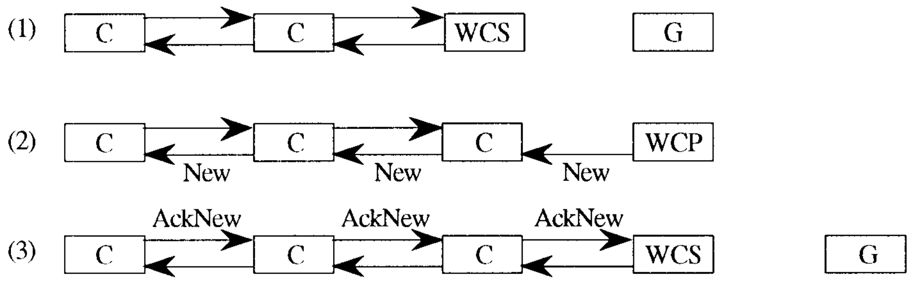
New: allows a newly generated module to be known by all its predecessors. Each predecessor which receives this message memorizes the identity of the newly generated module and transfers it to its own predecessor When this message arrives to the first module of the chain, it memorizes the identity in the stable memory and sends an acknowledgement message to its successor.
AckNew: is the acknowledgement message for the New message sent by the newly generated module. It is used to confirm to this module that it is actually known by all its predecessors.

Update: in case of recovery, this message is sent by the regenerated module in order to inform the successor that it is its new predecessor. This message is always taken into account and acknowledged.

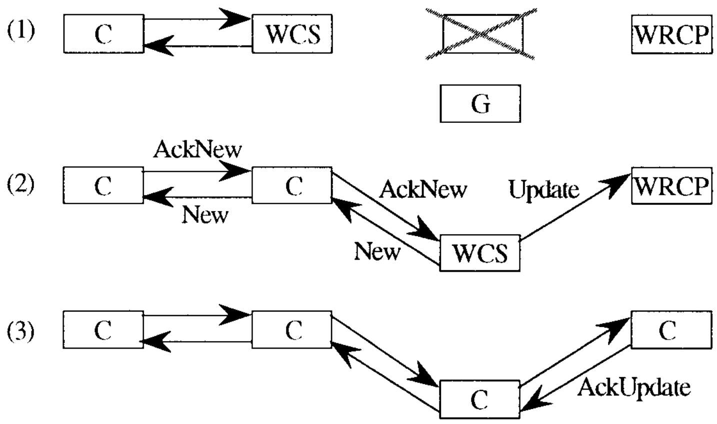
AckUpdate: confirms to a regenerated module that its successor considers it as its new predecessor.

When a module is generated, or regenerated, it receives the identity and the knowledge vector from its predecessor (the generating module) When the initial module is generated, or regenerated, it receives the knowledge vector saved in stable memory

This example shows the generation and the insertion of the fourth module In the first step, the third module generates a new module and changes to the WaitConnSucc (WCS) state The new module is in Generated *(G)* state. In the second step, the new module sends its identity (New message) to its predecessor and changes to the WaitConnPred (WCP) state The New message is retransmitted from predecessor to predecessor. In the last step, the initial module sends the acknowledgement of the New message (AckNew message) which is retransmitted from successor to successor. When the new module receives the AckNew message, it generates a successor and changes to the WaitConnSucc state This process is repeated until the generation and insertion of the Nth module.



This second example shows the recovery of a faulty module. In the first step, the second module detects the failure of the third one. It regenerates a substitute module and changes to the WaitConnSucc (WCS) state. The fourth module detects its disconnection with the faulty module and changes to the WaitReConnPred (WRCP) state. The regenerated module is in the Generated state. In the second step, the substitute module is recognized by its predecessors, like in the above example, sends an Update message to its successor and changes to the WaitConnSucc state. In the last step, the fourth module receives the Update message, acknowledges it by sending an AckUpdate message and changes to the Connected state. When the substitute module receives the AckUpdate message it changes to the Connected state.



4.4.3. Definition of rules

We give the various rules for the building of the chain, its rebuilding after the failure of particular modules and the elimination of sub-chains.

4.4.3. 1. Rules of building

Initialization rule: The original chain creation demand comes from the user who generates the first module (CMl module). When this first module is generated (generated state) it then also generates the second module and so on.

Connection rule: When a module is generated, it simultaneously receives the identity of the module which generated it (except for the first module). A module that is in state Generated sends its identity to its predecessors (the generating module) through the emission of a New message, and waits for an acknowledgement (WaitConnpred state).

The identity of a communication module enclose localization, localization of the associated processing module etc...

Knowledge rule: A module which receives its successor's identity from a newly generated module memorizes it and sends it back to its predecessor. If the receiver is the initial one of the chain, it saves it in stable memory and sends an acknowledgement message (AckNew message) to its successor. Moreover, if the receiver is in state WaitConnSucc (it has no successor) then the message sender is considered as its successor and changes to Connected state (it generated the sender of message).

Generation rule: When a module CMi receives an AckNew message from its predecessor (CMi-l), it sends it to its successor (CMi+l). If CMi is the newly generated one (WaitConnpred state) two cases are possible:

- **.CMi** is the Nth module of the chain (i.e. the last one) then it changes to the Connected state and the building is over;
- **.CMi** is not the last of the chain (i≠N), two other cases can occur: either it received the identity of CMi+ 1 (rebuilding step) and then it can connect itself to it and the rebuilding is over; either the CMi did not receive the identity of CMi+l, then it generates a successor and changes to the WaitConnSucc state (building step).

  When a module detects its successor's failure it only generates an other module when it is connected to a predecessor.

4.4.3.2. Rules of maintenance

The rebuilding rules of the chain consist in starting the recovery process as soon as a failure is detected. Two cases must be taken into account: either the chain is already built either it is in building or rebuilding step.

Regeneration rule: When a CMi-l detects its successor failure, the CMi module, it replaces it with the generation of CMi' substitute module and waits in the WaitConnSucc state. At the same time, it gives it its own coordinates and those of CMi+l to CMN modules, if they exist.

Update rule: An CMi module receiving a Update message from a CMj module always acknowledges it, and considers the CMj module as its new predecessor.

Purge rule: When a module, that is disconnected to its predecessor, does not receive reconnection demand (Update message) during a particular while, it destroys itself.

This rule allows the gradual elimination of modules in a sub-chain wrongly created.

4.5. Verification of the algorithm

4.5.1. Model

The specification of a distributed application always requires the choice of a model. Two possibilities are mainly considered: either the programming language supposed to build the application; either any f'ormal model such as C.C.S. [5] or the Petri nets [2]. Though the first solution is more attractive for reduced cost reasons, it does not include any proving method, which is irrelevant for complex systems that require a validity proof. With the second solution, some validation tools are available which allow the formal proving of the system correction.

We choose a formal model, though more general then the one mentioned above and which is highly inspired from [4]: the so called event model. It is worth our while to choose a general model than others which are more specific.

4.5.1.1. Definition of transition systems

The event model describes a system with the totality of its states and the set of events that make it change from one state to another.

Definition 1: A transition system S is a pair (E,R) where:

- E is the set of system states.
- R is the set of system rules.

where $r \in R$, .r.p: $E \to \{\text{true, false}\}$ is a predicate which defines the guard

.r.a: $E \to E$ is a function which defines the action

More precisely, r.p is a predicate on a system state which takes the true value if the r.a action is possible in that state, and the false one if the action is not possible in that state. Thus, R defines the system behaviour and a step of the system is defined by :

$$e \to_r e' \text{ if and only if } r.p(e) \text{ and } e' = r.a(e)$$

Such a model is called event model for the event is the occurrence of an action. Then, an event makes the system change from a state to another.

4.5.1.2. Proving techniques on the model

System states and properties may be expressed as predicates. The proof method based on the capture of system states and properties by predicates, and which is adapted to our model, is called assertions-oriented method.

We are often interested to ensure that if a system verifies a property at a moment or at a given state, so this property is retained whatever the system evolution. Predicates which express this kind of properties are called invariants.

Given a transition system, we generally study the behaviour of the system when started in certain initial states. It is, of course, unlikely that we are interested in considering all elements of E as possible initial states. When a system starts in e_0, the only interesting states are those reached by the system from e_0.

Definition 2: Let $S= (E,R)$ be a transition system and $e_0 \in E$. We define the t'unction .*Acc* over E as follows :

(i) $e_0 \in Acc(e_0)$

(ii) if $e \in Acc(e_0)$ and $e \rightarrow_r e'$ then $e' \in Acc(e_0)$

Acc(e) is the set of all reachable states from e.

Definition 3: Let $S= (E,R)$ be a transition system and $e_0 \in E$. A predicate P is said to be e_0-*invariant* if for each $e \in Acc(e_0)$ $P(e)$ is true.

To prove that a predicate P is e_0-*invariant* requires to prove that P is true for all states of $Acc(e_0)$. This is, tedious for complex systems and impossible when $Acc(e_0)$ is infinite. Following Keller [4], we propose a more restrictive concept namely *induction.*

Definition 4: Let $S= (E,R)$ be a transition system and $e_0 \in E$. A predicate P is said to be e_0-*inductive* provided that:

(i) $P(e_0)$ and

(ii) $\forall$ e, e' $\in E$, $P(e)$ AND $e \rightarrow_r e'$ implies $P(e')$

The power which lies in the induction principle is that it does not require a complete characterization of reachability in order to demonstrate that a predicate is invariant. Proving that a predicate remains true after every action which makes the system change from a state to another is enough to conclude that this predicate is invariant. Thus, we have just to study the initial state and the rules.

It is not a restriction to consider an inductive predicate instead of an invariant predicate because the first one is obtained by strrengthening and including the second one.

Proposition 1 [4]: Any invariant is not an inductive invariant, but any invariant has an inductive invariant which implies it.

The discussion above has involved the use of the induction principle to show that certain properties always hold. However, there are other conditions which might be required before a system can be considered correct.
In some cases, it is desirable that a system always terminates for certain initial values. This is the case with a system designed to complete a specific task. On the other hand, many systems are designed not to terminate, or to terminate only in abnormal situations.

Our algorithm matches the first case. The chain building must terminate when N modules are generated and each of them is in its "Connected" state. This constitutes a home state, noted e_H, for the system. However, it is not obvious to show that starting from an initial state the system will inevitably reach its home state. Hence we use another method for proving termination.

Definition 5: Let $S = (E,R)$ be a transition system and $e_0 \in E$. We say that a function $\eta : E \rightarrow \omega$ (any well-ordered set will do in place of ω) is a *R'-norm* ($R' \subset R$) *with minimal state* e_H provided that

(i) $\eta(e)$ is minimal iff $e = e_H$

(ii) $\forall\ e \in Acc(e_0)$, $\eta(e)$ is not minimal $\Rightarrow \forall\ e \in E, \forall\ r \in R'$, $e \rightarrow_r e'\ \eta(e') < \eta(e)$

R' is a *well-founded subset* of R such that each rule of R' decreases η. To prove the termination, we associate a *R'-norm* function η with the system such that η decreases each time an action is executed by the system and is minimal for the state we wish the system reach. This means that no matter what state the system goes to, it must always get to its home state.

4.5.2. Modelling

4.5.2.1. Problems related to the model

The time does not exist in the event model. Thus, it is tricky to model events whose occurrences obey to temporal constraints. Scheduling may also be tediously achieved. The nonexistence of time implies sometimes uncontrolled event occurrences which express conditions easily avoidable in the real system. The time abstraction can not be an inconvenience since the model allows to describe a behaviour which includes the real system behaviour.

4.5.2.2. Modelling choice

To model our system we choose an approach based on the characterization of its states. The communication chain is composed of a set of modules and a communication channel. The module state is defined by the values of its variables and the channel state is defined by the set of messages sent but not yet received. The system state is then defined by a combination of the states of the whole modules and the channel. In this way, we can study the system by means of its states and the actions which make it change from one state to another. An instance of an action is called an event and each sequence of events expresses the system behaviour.

We assume that the system is composed of an infinite array (mod[]) of modules. Each module is indexed with its identity that designates it in an unique manner. We give the structure components of a module which are used to prove the correctness of the algorithm in the next section.

<u>Array of modules indexed with Id</u>

mod[Id] Id: Integer $\cup$ {-1} is the module identity

MLev: [1,..., n] $\cup$ {-1} expresses the module level in the chain

Pred: Id $\cup$ {-1} is its predecessor's identity

Succ: Id $\cup$ {-1} is its successor'sidentity

```
State: [Non-existent, Generated, WaitConnPred,
       WaitConnSucc, Connected, WaitReConnPred]
```

mod[Id] is the module indexed by its identity Id. When a component is not defined, we give it the -1 value. We use a global variable Next to allocate an unique identity for each module.

Initially, all the modules have their State component equal to Non-existent and Next is equal to one (1).

The channel is modelled by a multi-set "Channel". The different types and contents of the exchanged messages are given in the following. We give the only necessary messages that are used to prove the correctness of the algorithm in the next section.

New = (type = New, Sender, receiver, source-sender, Level)
AckNew = (type = AckNew, Sender, receiver, final-receiver)

Two actions are defined on the messages:

- Channel := Channel-m, expresses the reception of the message m.
- Channel := Channel + (type, compl, comp2, ...), expresses the emission of the message (type, compl, comp2, ...).

Moreover,

- Channel $\geq$ m, expresses that there is at least one message m in the channel.
- Channel = $\emptyset$, expresses that the channel is empty.

4.5.2.3. Proof

The assertions-oriented method lies on predicates that often express system global variables. In this way, the user has a general view of the system and can then better master its evolution in case of event instances.

For sake of simplicity and lack of place, we give the correctness of the algorithm for the chain building in a reliable environment.

At each time, in the chain building step we have:
$\mathbf{I_0}$: $\exists\ 0 \leq k \leq N$ modules, $\forall$ i, $i \leq k \Leftrightarrow$ mod[i].state $\neq$ Non-existent
which expresses that k modules have been generated. N is the length of the chain we want to build.
The **I** invariant which expresses all the states stemming from the algorithm execution, i.e. the states of the k modules and the channel, is defined as follow:

$$\mathbf{I} = \mathbf{I_0} \text{ AND } (\mathbf{I_{deb}} \text{ OR } \mathbf{I_1} \text{ OR } \mathbf{I_2} \text{ OR } \mathbf{I_3} \text{ OR } \mathbf{I_4} \text{ OR } \mathbf{I_5})$$

where

Ideb:			k = 0,	The building is not started
$\mathbf{I_1}$:	$\forall$ i	$\in$	[1,k-2],	mod[i] .state = Connected mod[k-l] .state = WaitConnSucc mod[k] .state = Generated
$\mathbf{I_2}$:	$\forall$ i	$\in$	[1,k-2],	mod[i] .state = Connected mod[k-l] .state = WaitConnSucc mod[k] .state = WaitConnPred Channel = m, m.type = New
$\mathbf{I_3}$:	$\forall$ i	$\in$	[1,k-1],	mod[i] .state = Connected mod[k] .state = WaitConnPred Channel = m, m.type = New
$\mathbf{I_4}$:	$\forall$ i	$\in$	[1,k-1],	mod[i] .state = Connected mod[k] .state = WaitConnPred Channel = m, m.type = AckNew
$\mathbf{I_4}$:	$\forall$ i	$\in$	[1,k],	mod[i] .state = Connected Channel = $\emptyset$ k = N

Informally, the $\mathbf{I_{deb}}$ predicate takes into account the system states for which the chain building has not started. This predicate becomes and remains true as soon as the initialization starts. The last predicate $\mathbf{I_5}$ allows to express the termination condition of the building. The other predicates take into account the situations where the building is still in progress.

We show now that the previously defined invariant remains true whatever the action that can change state to the system. For that, we systematically study all the actions which can alter **I**. We consider one by one each predicate of **I** and establish that if an action of the system alters the considered predicate, then one of the others predicates of **I** becomes true.

Let $\mathbf{I_l}$ be true. The only one possible event is the sending of a New message by the k-th module. Two cases are possible. If the k-th module is the initial one (k=l), then it generates a successor and the $\mathbf{I_l}$ predicate remains true. Otherwise ($k \neq 1$), it sends its identity (included in a New message) to its predecessor and changes to the WaitConnpred state. This alters $\mathbf{I_l}$ but make true $\mathbf{I_2}$.

We proceed in the same manner with the others predicates to demonstrate that **I** is an inductive invariant since we take into account all the possible rules of the system.

5. Conclusion

We have presented an algorithm providing fault-tolerance for layered distributed systems. From the OSI model, we have considered a communication chain. Our algorithm ensures the building and the preserving of the chain in a unreliable environment. This is achieved by introducing the dynamic regenera tion of faulty elements. In contrast to other methods, the dynamic regeneration method tolerates an unlimited number of failures with a smaller overhead. Naturally this technique is only applicable for software architectures. The correctness of the algorithm is formally proved.

At the prospect, the generalization of the dynamic regeneration to any architectures where each component has one predecessor and one or more successors. This kind of architectures includes ring, tree and so on which are most often operated.

References

[1] A. Avizienis, The *N-version Approach to Fault- Tolerant,* IEEE Transactions on Software Engineering vol n° 12, December 1985.

[2] G.W. BRAMS (collective name), *Réseaux de Petri: Theorie et pratique,* Edited by Masson, Vol. 1 and 2, Paris, 1982 and 1983.

[3] J. Henshall, S. Shaw, *OSI EXPLAINED End-to-End Computer Standards,* Second Edition, Ellis Horwood limited, 1990.

[4] R.M. Keller, *Formal Verification of Parallel Programs,* Communications of the ACM, July 1976, Vol. 19, No.7, pp. 371-384.

[5] R. Milner, *Communication and Concurrency,* Edited by Prentice Hall, 1989.

[6] V.P. Nelson, *Fault-Tolerant Computing: Fundamental Concepts,* COMPUTER, July 1990.

[7] B. Randell, *Design Fault Tolerance,* The evolution of Fault-Tolerant Computing, A. Avizienis, H. Kopetz, J-C. Laprie, Edited by Springer-Verlag 1987, Vol. 1, pp 251-270.

[8] S.S.B. Shi, G.G. Belford, *Consistent Replicated Transactions, A highly Reliable Program execution Environment*, Eighth Symposium on Reliable Distributed Systems, Seattle, Washington, October 1989, pp 30-41.

[9] N.A. Speirs , P.A. Barett, *Using passive Replicates in DELTA-4 to provide dependable distributed computing*, 19-th Fault-Tolerant on Computing Systems, 1989.

[10] P. Thambidurai, K.S. Trivedi, *Transient Overloads in Fault-Tolerant Real-Time Systems*, Real Time Systems Symposium, Santa Monica, Californie, 1989, pp 126-133.