A Structural Approach to Design Reliable Distributed Applications

M. Taghelit, S.Haddad

Université Paris Dauphine
Laboratoire LAMSADE
Paris, France

P. Sens

Université Pierre et Marie Curie
Laboratoire MASI
Paris, France

Abstract

This paper presents a method to design reliable applications composed of communicating entities. This method is based on the separation of communicating and processing functions to create a software "network" independent of the computational processes. Common functionalities are isolated to reduce the overhead of replication based approach. In contrast to classical techniques only the processing part of entities is replicated. We exhibit a homogeneous communication structure: the R-CHAIN. We describe an algorithm to dynamically reconfigurate the R-CHAIN in presence of host failure. This algorithm provides an efficient way to maintain end-to-end communication.

Key words: fault tolerance, distributed systems, functional structuring, dynamic regeneration

1. Introduction

Distributed systems provide great opportunities for developing high-performance applications; at the same time, because of dependency between components, such systems are particularly fragile: one component failure may cause all the system failure. Replication of data and/or computation on different nodes is the only means by which a distributed system may continue to provide non-degraded service in presence of failed nodes [11]. However, such techniques imply a great overhead and tolerate a limited number of failures proportional to the number of copies.

We propose a new approach to design reliable distributed systems reducing the cost of replication. Our goal is to design a software "network" independent of the computational processes. This network maintains the initial communication structure in an unreliable environment and allows to unload applications from a part of fault tolerance management. Our method is based on two concepts: the functional structuring and the dynamic regeneration. The functional structuring exhibits an homogeneous communicating structure: the R-CHAIN.

Dynamic regeneration maintains the communication between entities. Two entities converse through a homogeneous communication chain issued from the functional structuring. Our algorithm is based on the dynamic regeneration technique: whenever a failure of an entity is detected, a new one is regenerated and integrated into the chain. Instead of replicating elements, they are regenerated in case of failure. However, the regeneration is only applied to homogeneous entities.

This paper is organized as follows. Section 2 describes system and application models. Section 3 presents our design method based on the functional structuring and the dynamic regeneration. Section 4 describes the algorithm that allows to maintain a communication chain in an unreliable environment. In Section 5, we show the correctness of the algorithm. Section 6 briefly presents an application of the R-CHAIN to large scale networks. We conclude in section 7.

2. Models and environment

We consider applications composed of a set of communicating *entities*. We do the following assumptions on communication between two entities:

- the same entity always initializes the sending,
- communication involves inner entities,
- each entity may perform local processing.

Such a description corresponds to the most distributed systems based on the client server architecture where several servers cooperate to achieve a request. An example is shown in the figure 1. The entities E1 and E6 communicate through the inner entities E2, E3, E4 and E5.

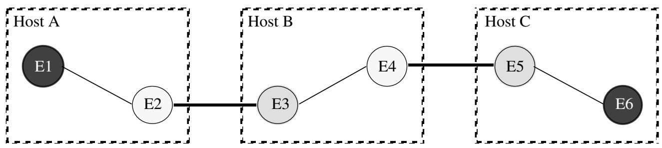

Figure 1: Communicating entities

The system is composed of *fail-silent* processors [10], where the failed node simply stops and all the processes on the node die. Viewed from the communication network a faulty processor remains silent and cannot receive or send message. Valid processors are not automatically informed of the failure.

3. Design of reliable application

We want to provide fault tolerance to applications composed of a chain of communicating entities. In most fault tolerance systems, software components are replicated [4, 11].

A replicated software component is defined as a software component that possesses a representation on two or more hosts [11]. There are three kinds of replication: the active and semi-active ones in which all replicas process concurrently all input messages, and the passive one in which only one of the replicas processes all input messages.

Active and semi-active replications lead to a high overhead of treatment. If the degree of replication is n (i.e., there are n replicas), there will be n treatments to produce one result [3, 4].

Passive replication economizes processor utilization by activating redundant replicas only in case of failures [13]. However, this technique needs a checkpoint management which is expensive in processing time and space [12].

Usually, fault tolerance is achieved by means of replication of whole software entities [11, 1]. The main drawback of such techniques is the increase of the system complexity and the limited fault-tolerance degree. Moreover, each replica has its own fault probability and so the global fault probability of the application is bigger. These techniques are the only way to provide fault tolerance to hardware architecture but in a software context components do not need to pre-exist. From this principle, we propose to reduce replication overhead by functional structuring.

3.1. Functional structuring

The functional structuring is composed of the two following steps.

First, we assume that the location of entities is no more fixed. Thus, an entity can transparently change location in case of failure. It does not depend of a specific host. An entity can equally run on any hosts. We assume that the underlying system provides a migration facility. Each entity is designed by a *global name* independent of its location.

In the second step, we apply a functional division of entities. Each cooperating entity is composed of a common part which encloses the receiving, sending and storage of messages, and of a specific part which achieves the local processing. All common communication functions of an entity are gathered in a same module and processing functions are gathered in an other one. An entity is now composed of two specialized modules: the communication module and the processing one (figure 2).

After this structuring, the communication between entities is achieved through *an homogeneous communication chain* of communication modules: the R-CHAIN (Reliable Chain).

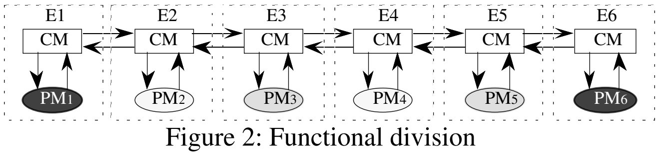

Figure 2: Functional division

The communication module is in charge of the communication with the environment (other entities) and with the processing module. The functional division has the following advantages:

- the diagnosis is more precise and allows a more efficient recovery,
- the communication is independent of processing and can be achieved even if the processing part fails,
- dynamic regeneration can be applied to the set of communication module.

3.2. Dynamic Regeneration

Homogeneous elements (i.e., the communication modules) are no longer replicated but dynamically generated from the valid ones in case of failure. The generated element is created and integrated in the system only after one failure. This element has the same function of the faulty one.

The dynamic regeneration is only applied to homogenous set of entities (i.e., communications modules). The fault tolerance of heterogeneous processing part is achieved by classical replication techniques.

This approach has the following advantages:

- Since communication modules are not replicated, the system is less complex than a system using only replication techniques.
- The overhead under normal functioning is only related to the fault detection, fault management of processing part, and storage of reconstruction information in stable memory.
- The fault tolerance degree of the communication module is not limited.

Since new modules are generated from the valid ones, the regeneration only deals with homogeneous software components.

The failure of an entity may come either from the communication module or from the processing module. Since modules are independent, the failure of one module does not directly affect the other.

The failure recovery of the communication module (CM) is composed of the following steps:

- generation of a new communication module (CM') by the environment (i.e., the other communication modules),
- CM' connection to the environment,
- CM' connection to the associated processing module (PM).

If PM fails, CM selects a new processing module from the set of replicas and connects itself to this one. This mechanism is totally transparent the other module of the chain.

4. The dynamic regeneration algorithm

Now, we present, more precisely, the algorithm providing fault tolerance of the R-CHAIN. This algorithm fulfills two main goals:

- **Building of the communication chain**. The building of the chain consists in generating a finite set of communication modules such that each inner one has a predecessor and a successor. Modules are generated dynamically. Only the first module is created 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. The treatment of a failure is composed of three steps: the detection, the regeneration of a new element and its insertion in the chain.

4.1. Algorithm principles

The algorithm must build and maintain a chain of N modules. It relies on the four following principles:

• Activation principle: Each non-ending module carries on the building of the chain.

Each module without successor and which is not the Nth one generates a successor. This principle allows to carry on the building of the chain if it is not complete and to rebuild it if some modules fail.

• Knowledge principle: Each module knows all its successors

In case of failure, a new module is regenerated according to the activation principle. Then, this module must 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 to give to the new module the necessary information to insert itself in the chain.
In case of multiple failures, 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 information from its predecessor then it keeps this information in stable memory [2].

• Purge principle: Each non-initial module with no predecessor destroys itself after a finite time

When predecessor of a module fails, the module must be eliminated if no new module replaced the predecessor after a finite time. In this way, we avoid the creation of "parasite" sub-chains (see next section).

• Suspicion principle: The validity of the chain is periodically tested

To detect faulty module, each module periodically checks the validity of its successor. The detection is based on a watch dog mechanism. Each module periodically transmits a control message to its successor 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`.

4.2. Definition of the algorithm

The states transitions of a communication module are set up by the reduced graph of figure 3. For the sake of simplicity, only the main transitions are described. We do not specify neither various recovery cases nor the fact that a module can fail at any moment.

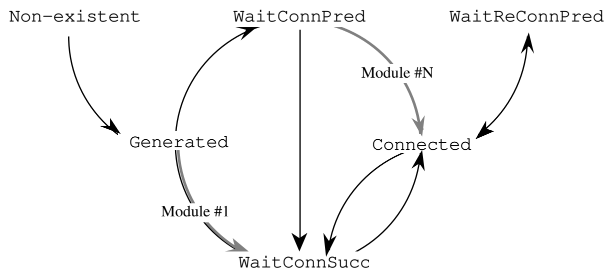

Figure 3 : States transitions of a communication module

`Non-existent`: the module does not exist.

`Generated`: the CMi module has just been generated.

`WaitConnPred`: CMi gave its identity to the module which generated it (CMi-1) and it waits for an acknowledgment message.

`WaitConnSucc`: CMi generated a CMi+1 module to which it gave its identity, and it waits for the reception of the CMi+1 identity.

`Connected`: CMi received the identity from CMi+1.

`WaitReConnPred`: CMi detected its disconnection with its predecessor, it waits for 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.

The various message types exchanged between communication modules are:

`New`: allows a newly generated module to be known by all its predecessors. When a predecessor receives this message, it memorizes the identity of the new module and forwards 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 acknowledgment message to its successor.

`AckNew`: is the acknowledgment message for the `New` message. It is used to confirm to the new module that it is known by all its predecessors.

`Update`: in case of recovery, this message is sent by the regenerated module 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 `knowledge` vector from its predecessor (the generating module). When the initial module is generated, or regenerated, it uses the `knowledge` vector saved in stable memory.

Building of the chain
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

insertion in the chain already built. This insertion of a module is over when all its predecessors know its presence. The example of the figure 4 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 acknowledgment 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.

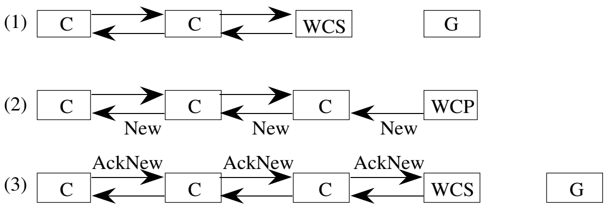

Figure 4 : Building of the chain

Chain maintenance in case of failures

The figure 5 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 according to the activation principle 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, all predecessors register the substitute module. Then the module 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 and the recovery is over.

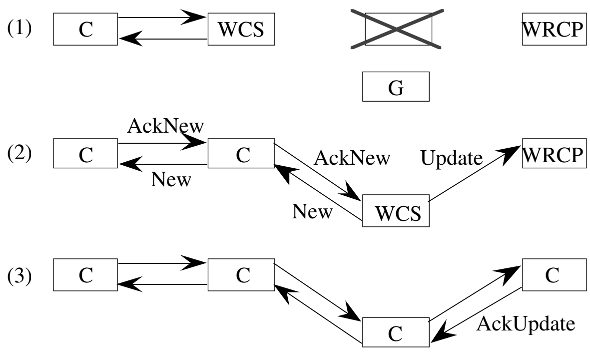

Figure 5 : Failure recovery

In case of false detection, a new module is regenerated and is connected 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 according to the activation principle. Such a case is shown in figure 6.

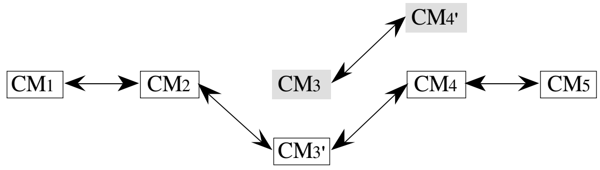

Figure 6 : Creation of a sub-chain from an isolated module (CM3)

In case of network partitioning, several adjacent modules can also constitute a sub-chain.

The destruction of parasite sub-chain is achieved by the *purge* principle.

5. Verification of the algorithm

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 formal model such as C.C.S. [7] or the Petri nets [8]. 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 [5]: the so called event model.

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 behavior.

We assume that the system is composed of an infinite array (mod[]) of modules. Each module is indexed with its unique Id. We define a module has follow:

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's identity
State: [Non-existent, Generated, WaitConnPred, WaitConnSucc, Connected, WaitReConnPred]

`mod[Id]` is the module indexed by its identity Id. When a component is not defined, Id is set to -1. We use a global variable `Next` to allocate a unique identity for each module. Initially, all the modules set `State` to Non-existent and `Next` to 1.

The channel is modeled by a multi-set "Channel". We give only the messages used to prove the correctness of the algorithm.

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, comp1, comp2, ...), expresses the emission of the message (type, comp1, comp2, ...).

Moreover,

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

The assertions-oriented method relies on predicates that often express system global variables. In this way, the user has a global 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:

Io: $\exists\ 0 \leq k \leq N$ modules,
$\forall$ i, i $\leq$ k <=> mod[i].state $\neq$ Non-existent

which expresses that k modules have been generated. N is the long of the chain we want to build. The **I** invariant expresses all the states stemming from the algorithm execution (i.e., the states of the k modules and the channel). **I** is defined as follow:

I = **Io** $\wedge$ (**Ins** $\vee$ **I1** $\vee$ **I2** $\vee$ **I3** $\vee$ **I4** $\vee$ **I5**)

where

Ins: k = 0, The building is not started

I1: $\forall$ i $\in$ [1, k-2],
mod[i].state = Connected
mod[k-1].state = WaitConnSucc
mod[k].state = Generated

I2: $\forall$ i $\in$ [1, k-2],
mod[i].state = Connected
mod[k-1].state = WaitConnSucc
mod[k].state = WaitConnPred
Channel = m, m.type = New

I3: $\forall$ i $\in$ [1, k-1],
mod[i].state = Connected
mod[k].state = WaitConnPred
Channel = m, m.type = New

I4: $\forall$ i $\in$ [1, k-1],
mod[i].state = Connected
mod[k].state = WaitConnPred
Channel = m, m.type = AckNew

I5: $\forall$ i $\in$ [1, k],
mod[i].state = Connected
Channel = ϕ
k = N

Informally, the **Ins** 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 **I5** 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. 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 **I1** 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=1), then it generates a successor and the **I1** predicate remains true. Otherwise (k≠1), it sends its identity (included in a `New` message) to its predecessor and changes to the `WaitConnPred` state. This alters **I1** but makes true **I2**.

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.

6. Application to large scale system

Large scale communication networks provide new opportunity to access to a high number of servers. The probability of a server failure or temporal disconnection is fairly high. The R-CHAIN can be used to ensure a reliable access to ending server.

Mobile agent is an attractive model to efficiently implement the R-CHAIN. Some models such as MAP (Mobile Assistant Programming) [9] or TeleScript [15] have already been defined to develop and execute distributed application in large scale networks of heterogeneous computers.

Several servers (for instance WWW servers) are often involved to achieve a client request. R-CHAIN can be used to access resources efficiently and in a reliable way. The first module of the chain in created on the first server. If the request requires an other server a new module of the R-CHAIN is created on the new server and so on. Thus, the client request creates an R-CHAIN among the set of servers. Each element of the R-CHAIN will be implement as a mobile agent. At the building step, the predecessor uses facility of agent programming language to create a clone on the following server. When a server failure is detected, the predecessor finds an equivalent server and creates a new clone on this later. The mobility is used to reconfigurate the R-CHAIN when the effective bandwidth of network becomes to low. This scheme has the following advantage: the natural redundancy of servers and inherent parallelism of large scale network are used.

7. Conclusion

We have presented a new method to design reliable distributed applications. This approach reduces the cost of replication by isolating common functionalities. The application is based on a software network providing reliable communication, the R-CHAIN. Thus, only processing parts of application are concerned by classical replication technique. We propose an algorithm to maintain the end-to-end communication by means of the regeneration concept. The communication structure is maintain with a low failure-free overhead.

Our method can easily be applied to provide fault tolerance to layered distributed systems [14]. We study the adaptation of the R-CHAIN to provide reliable accesses to WWW servers. It may be applied not only to maintain the structure communication of systems but to manage messages flows as well.

References

[1] O. Babaoglu, L. Alvisi, A. Amoroso, R. Davoli. Paralex: An Environment for Parallel Programming in Distributed Systems. *Proc. of International Conference on Supercomputing*, Washington D.C., pp. 178-187, July 1992.

[2] M. Banâtre, G. Muller, B. Rochat, P. Sanchez. Design Decisions for FTM: a General Purpose Fault Tolrant Machine. *Proc. of the 21st International Symposium on Fault-Tolerant Computing* (FTCS-21), pp. 71-78, Montréal, Quebec, Canada, 1991.

[3] M. Chérèque, D. Powell, P. Reynier, J. L. Richier, J. Voiron. Active Replication in Delta-4, *Proc. of the 22th International Symposium on Fault-Tolerant Computing Systems* (FTCS 22), Boston, Massachusetts (USA), pp.28-37, July 1992.

[4] E.N. Elnozahy, W. Zwaenepoel. Replicated Distributed Processes in Manetho. *Proc of the 22nd International Symposium on Fault-Tolerant Computing Systems* (FTCS 22), pp.18-27, Boston, MA, USA, 1992.

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

[6] M. Litzkow, M. Livny & M.Mutka. Condor - A Hunter of Idle Workstations. *Proc. of the 8th International Conferance on Distributed Computing Systems*, San Jose, Calif. June 1988.

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

[8] T. Murata. Petri Nets: Propertie, Analysis and Applications. Proceedings of the IEEE, vol. 7, n° 4, April 1989.

[9] S. Perret, A. Duba. MAP: Mobile Assitant programming for Large Scale Communication Networks. BROADCAST Third Year Report. Esprit Basic Research Projetct 6360, July 1995.

[10] D. Powell, G. Bonn, D. Seaton, P. Verissimo, F. Waeselynck, The Delta-4 Approach to Dependability in Open Distributed Computing Systems, *Proc. of the 18th International Symposium on Fault-Tolerant Computing Systems* (FTCS 18), pp. 246-251, Tokyo, Japon, (IEEE), 1988.

[11] D. Powell (Ed.). Delta 4: A Generic Architecture for Dependable Distributed Computing. Research Reports ESPRITS, Berlin, Germany, Springer-Verlag, 1991.

[12] P. Sens. The Performance of Independant Checkpointing in Distributed Systems. *Proc. of the 28th Hawaii International Conference on System Science* (HICSS-28), pp. 525-533, Maui, Hawaii, January 1995.

[13] N. A. Speirs, P. A. Barett. Using passive Replicates in DELTA-4 to provide Dependable Distributed Computing. *Proc. of the 19th International Symposium on Fault-Tolerant Computing Systems* (FTCS 19), Chicago, MI, USA, (IEEE), pp. 184-190, 1989.

[14] M. Taghelit, S. Haddad, P. Sens, *An algorithm providing fault-tolerance for layerd distributed systems*, Proc. of the IMACS-IFACS International Symposium on parallel and distributed computing in Engineering systems, Corfou, Greece, June 1991.

[15] J.E. White. Telescript technology: The foundation for the electronic marketplace. *Technical white paper, General Magic Inc.,* 1994

[16] J. Xu, R.H.B Netzer. Adaptive Independent Checkpointing for Reducing Rollback Propagation. *Proc. of the 5th IEEE Symposium on Parallel and Distributed Processing,* pp. 754-761, December 1993.