STUDIA UNIV. BABEŞ-BOLYAI, MATHEMATICA, XXXIX, 3, 1994

MODELLING DISTRIBUTED EXECUTION IN THE PRESENCE OF FAILURES

Alexandru VANCEA[*]

Dedicated to Professor Sever Groze on his 65th anniversary

Received: February 10, 1995
AMS subject classification: 68Q10, 65Y05

REZUMAT. - Modelarea execuţiei distribuite în prezenţa căderilor. Pe măsura răspândirii tot mai accentuate a sistemelor distribuite şi a utilizării lor de către nespecialişti, atributul toleranţei la erori (*fault tolerance*) în prezenţa unor căderi ale unor servere devine o cerinţă absolut esenţială pentru un sistem de calcul distribuit viabil. Lucrarea prezintă o clasificare şi o modelare matematică a celor mai răspândite tipuri de căderi ale serverelor. O astfel de modelare constituie un prim pas pentru o abordare sistematică a posibilităţilor soft de transformare automată a unor căderi grave în căderi mult mai puţin grave la nivelul efectelor execuţiilor, activitate care ar asigura practic toleranţa la erori ale acelor sisteme de calcul distribuite.

1. Introduction. One of the original goals of building distributed systems was to make them more *reliable* than single processors systems, That is, if some machine goes down, some other machine takes over the job. There are various aspects regarding this concept.

Availability refers to the fraction of time in which the system is usable. Availability oan be enhanced by a design that does not require the simultaneous functioning of a substantial number of critical components. Another way for improving availability is redundancy: key pieces of hardware and software should be replicated, so that if one of them fails the others will be able to take up the task.

A main aspect related to reliability is *fault tolerance* [Tan92]. That is, what happens

* *"Babeş-Bolyai" University, Faculty of Mathematics and Computer Science, 3400 Cluj-Napoca, Romania*

when a server crashes and then quickly reboots ? In general, distributed systems can be designed to mask failures. If a system service is actually constructed from a group of closely cooperating servers, then it should be possible to construct it in such a way that users do not notice the loss of one or two servers, other than some performance degradation. The challenge is to arrange this cooperation such as not to add substantial overhead to the system in the normal case, when everything is functioning correctly.

2. Types of failures. Distributed computing systems give algorithm designers the ability to write fault-tolerant applications in which correctly functioning processors can complete a computation despite the failure of others. It is well established that th complexity of writing such applications depends upon the type of faulty behaviour that processors may exhibit. For example, while simple stopping failures are relatively easy to tolerate, tolerating completely arbitrary behaviour can be much more difficult. To assist the designers of such applications, *translations* were developed that automatically convert algorithms tolerant of relatively benign types of failure into ones that tolerate more severe faulty behaviour [Bazzi93]. We give below a hierarchy of the most commonly considered failures, from the most easy ones to the most severe:

i). *crash failures* - in which the incriminated processors fail by stopping prematurely. Before they stop they behave correctly and after they stop they take no further actions.

ii). *send-omission failures* - the processor fails by intermitently omitting to send some of the messages that it should send, but the sent messages are always correct. Because these processors can fail and yet continue to send messages, their failure is more difficult to detect and deal with than crash failures.

iii). *general omission failures* - the processor may stop or it may intermittently fail to send or receive messages [Perry86] and the sent messages are always correct. The identity of faulty processors is uncertain: did the sender or the receiver of an omitted message fail ?

iv). *arbitrary failures* - processors subject to arbitrary failures can take any action [Lamport82]. They can stop, omit to send messages, send spurious messages and falsely claim to have received messages they did not actually receive.

3. Protocols, histories and problem specifications.

Definition. A *distributed system* is a set D of n processors joined by bidirectional communication links. Processors do not share any memory, the communications being made through message passing. Each processor has a local *state* and we denote by Q the set of local states.

Processors communicate with each other in synchronous rounds. In each round, a processor first sends messages, then receives messages and then change its state. Let M be the set of messages that may be sent in the system and let $\square \notin M$ be the value that indicates "no message" and let $M'=M\cup\{\square\}$. Thus, if p sends no message to q in a round, we can say that p sends $\square$ to q, although no message is actually sent.

Definition. Processors run a *protocol* P, which specifies the messages to be sent and the state transitions. A protocol consists of two functions, a *message function* and a *state-transition function*. The message function is defined as mf_p:N$\times D\times Q \rightarrow M$, where N is the set of positive integers. If processor p begins round i in state s, then P specifies that it send $mf_p(i,p,s)$ to all processors in that round. The state-transition function is st_p:N x D x $(M')^n$ –> Q. If in round i processor p receives the messages $m_1,...,m_n$ from processors $p_1,...,p_n$ respectively, then P specifies that it change its state to $st_p(i,p,m_1,...,m_n)$ at the end of round i.

The code below illustrates the execution of a protocol P:

```
state := initial state;
for i=1 to ∞ do
        message := mf_p(i,p,state);
        if message ≠ □ then send message to all processors;
        foreach q∈D
                if received some m from q then  get[q]:=m
                                                   else  get[q]:=□;
        state := st_p(i,p,get);
```

This definition of protocols appears restrictive in a sense. For example, every processor is required to broadcast a message in every round. A protocol's state transition function depends only on the messages that it just received and not on its previous state. Furthermore, processors are required to run forever and never halt. These restrictions were made for simplifying the presentation and they do not restrict the applicability of the results.

Histories describe the executions of a distributed system. Each history is a 4-tuple including the following elements: the protocol being run by the processors, the sequence of states through which the processors pass, the messages that the processors send and the messages that the processors receive. Formally, a history consists of a protocol and three functions. The functions define the states through which the processors pass and the messages sent and received by the processors in each round. A *state-sequence function* sseq:N×D–>Q identifies the states of processors at the beginning of each round. *sseq(i,p)* is the state in which processor p begins round i. A ***message-sending function*** msf:$\mathbf{N}\times D\times D$-->$M'$ identifies the messages sent in each round. ***msf(i,p,q)*** is the message that p sends to q in round i or □ if p sends no message to q in round i. A ***message-receiving function*** mrf:$\mathbf{N}\times D\times D$-->$M'$ identifies the messages received in each round. ***mrf(i,p,q)*** is the message that p receives from q in round i or □ if p does not receive a message from q in round i. Let mrf(i,p) be an abbreviation for the sequence mrf(i,p,1),..., mrf(i,p,n).

H=(P,sseq,msf,mrf) is then a *history of protocol P*.

A *system* is identified with the set of all histories (of all protocols) in that system. A system can also be defined by giving the properties that its histories must satisfy. If S is a system and H=(P,sseq,msf,mrf)$\in S$, then H is a history of P running in S.

Protocols are run to solve particular problems. Formally, such problems can be specified by predicates on histories. Such a predicate, called a *specification*, distinguishes histories that solve the problem from those that do not.

Protocol P *solves problem with specification* Σ *(or solves* Σ*) in system* S if all histories of P running in S satisfy Σ. That is $\forall H\in S$ [H is of the form (P,sseq,msf,mrf) $\Rightarrow$ H satisfies Σ].

4. Correctness and failures. A processor executes *correctly* if its actions are always those specified by its protocol. Considering a history H = (P,sseq,msf,mrf), processor *p sends correctly in round i of* H if

$$\forall q\in D\ [\mathrm{msf}(i,p,q) = \mathrm{mf}_P(i,p,\mathrm{sseq}(i,p))].$$

Processor *p receives correctly in round i of* H if

$$\forall q\in D\ [\mathrm{mrf}(i,p,q) = \mathrm{msf}(i,p,q)].$$

Processor *p makes a correct state transition in round i of* H if

$$sseq(i+1,p) = st_p(i,p,mrf(i,p)).$$

Processor *p is correct through round i of* H if it sends and receives correctly, and makes correct state transitions up to and including round *i* of H. Let

$$Correct(H,i) = \{p \in D \mid p \text{ is correct through round i of H}\}.$$

We assume that all processors are initially correct, so $Correct(H,0)=D$. Then let $Correct(H)$, the set of all processors *correct throughout history* H, be $\bigcap_{i \in N} Correct(H,i)$. If a processor is not correct, it is *faulty*. Formally,

$$Faulty(H,i) = D - Correct(H,i) \text{ and}$$

$$Faulty(H) = D - Correct(H).$$

The following examples of formal specifications illustrate these definitions: Σ_1 specifies that "in round 7 processor p does not send correctly to q":

$$\Sigma_1 = \Sigma_1(P,sseq,msf,mrf) = msf(7,p,q) \neq mf_p(7,p,sseq(7,p)).$$

Σ_2 specifies that through round 10 at least 6 processors are correct:

$$\Sigma_2(H) = |Correct(H,10)| \geq 6.$$

Informally, a specification Σ is a state specification if it depends only on the state-sequence function and, in a certain way, on the set of correct processors. Formally, Σ is a state specification if

$$\forall H_1,H_2 \ [(\Sigma(H_1) \wedge sseq_1=sseq_2 \wedge Correct(H_2) \subseteq Correct(H_1)) \Rightarrow \Sigma(H_2)].$$

Informally, a state specification Σ is failure-insensitive if it does not depend on the states of the faulty processors. Formally,

this means that

$$\forall H_1,H_2 \ [(\Sigma(H_1) \wedge \forall i \in N \ \forall p \in Correct(H_2)[sseq_1(i,p)=sseq_2(i,p)]) \Rightarrow \Sigma(H_2)].$$

Individual processors may exhibit *failures*, that is to deviate from *correct* behaviour.

They may do so by failing to send or receive messages correctly or by otherwise not following their protocol. In the following we will formally define crash, send-omission, general omission, and arbitrary failures.

4.1. Crash Failures. A *crash failure* [Hadzilacos83] is the most simple type of failure that can appear. A processor commits a *crash failure* by prematurely halting in some round. Formally, p commits a crash failure in round $i_c \in \mathbf{N}$ of H=(P,sseq,msf,mrf) if i_c is the least i such that $p \in \mathit{Faulty}(H,i)$ and if:

- p sends to each processor q either what the protocol specifies, or nothing at all:

$$\forall q \in D\ [msf(i_c,p,q) = mf_p(i_c,p,sseq(i_c,p)) \lor msf(i_c,p,q) = \square];$$

and, afterwards,

- it sends no messages: $\forall i > i_c\ \forall q \in D\ [msf(i,p,q) = \square]$,
- it receives no messages: $\forall i \geq i_c\ \forall q \in D\ [mrf(i,p,q) = \square]$,
- it makes no state transitions: $\forall i > i_c\ [sseq(i,p) = sseq(i_c,p)]$.

BIBLIOTECA FACULTĂŢII CLUJ-NAPOCA DE MATEMATICĂ

The system $C(n,t)$ corresponds to the set of histories in which up to t processors commit only crash failures and all other processors are correct. That is, $H \in C(n,t)$ if and only if D can be partitioned into sets C and F such that $C = \mathit{Correct}(H)$, $|F| \leq t$, and

$$\forall p \in F\ \exists i_c \in \mathbf{N}\ [p \text{ commits a crash failure in round } i_c \text{ of } H].$$

4.2. Send-omission Failures. Another type of failure, called a *send-omission failure*, occurs if a processor omits to send messages [Hadzilacos84]. Processor p may commit such failures in history H=(P,sseq,msf,mrf) if it always makes correct state transitions, receives correctly, and sends to each processor what its protocol specifies or nothing at all:

$$\forall i \in \mathbf{N}\ \forall q \in D\ [msf(i,p,q) = mf_p(i,p,sseq(i,p)) \vee msf(i,p,q) = \Box];$$

The system $S(n,t)$ corresponds to the set of histories in which up to t processors are subject to send-omission failures and all other processors are correct.

While crash failures are relatively easy to tolerate, intermittent send-omission failures are more difficult to identify and compensate. If processors may omit to send messages and later function correctly, then the correct processors may have more difficulty agreeing on the identity and timing of failures than they would if only crash failures occured.

4.3. General Omission Failures. A more complex type of failure, called a *general omission failure* [Perry86], occurs if a processor intermittently fails to send and receive messages. Processor p may commit such failures in history H=(P,sseq,msf,mrf) if it always makes correct state transitions, always sends to each processor what its protocol specifies or nothing at all, and always receives what was sent to it or nothing at all:

$$\forall i \in \mathbf{N}\ \forall q \in D\ [msf(i,p,q) = mf_p(i,p,sseq(i,p)) \vee msf(i,p,q) = \Box];$$

$$\forall i \in \mathbf{N}\ \forall q \in D\ [mrf(i,p,q) = msf(i,q,p) \vee mrf(i,p,q) = \Box].$$

The system G(n,t) corresponds to the set of histories in which up to t processors are subject to general omission failures and all other processors are correct.

General omission failures are more difficult to tolerate than send-omission failures. In addition to the uncertainty regarding the timing of failures, there may also be uncertainty as to the identify of the faulty processors: if an omitted message is detected, it may be difficult to tell whether it is the sender or the receiver that is at fault. Furthermore, faulty processors may be sending incomplete information, as they may have omitted to receive message from correct processors in previous rounds.

4.4. Arbitrary Failures. Crash failures considerably restrict the behaviour of faulty processors. Omission failures place fewer restrictions on this behaviour. In the worst case, faulty behaviour may be completely arbitrary. Processors may fail by sending incorrect messages and by making arbitrary state transitions [Lamport82]. Processor p *is subject to arbitrary failures* in history H=(P,sseq,msf,mrf) if it may deviate from P in any way. It may do one or more of the following:

- fail to send correctly: $\exists i \in \mathbf{N}\ \exists q \in D\ [msf(i,p,q) \neq mf_p(i,p,sseq(i,p))$,
- fail to receive correctly: $\exists i \in \mathbf{N}\ \exists q \in D\ [mrf(i,p,q) \neq ms(i,q,p)]$, or
- make an incorrect state transition: $\exists i \in \mathbf{N}\ [sseq(i+1,p) \neq st_p(i,p,mrf(i,p))]$.

The system $A(n,t)$ corresponds to the set of histories in which up to t processors commit arbitrary failures and all other processors are correct. It is clear that arbitrary failures are more difficult to tolerate than the other kinds. Faulty processors may actively try to confuse the correct ones, they being able even to "cooperate" to make fault-tolerance even more difficult to achieve.

5. Conclusions. As distributed systems become more and more widespread, the demand for fault tolerance is one of the main request from a distributed computing system. Such systems will need considerable redundancy in hardware and the communication infrastructure, but they will also need it in software and data.

To achieve the goal of a fault tolerant distributed system we have first to distinguish between the types of failures a system may exhibit and try to model these failures and the behaviour of the system in the presence of these failures. The ultimate goal, based on the ideas from [Bazzi93], will be to develop translations from one type of failure to another, translations destinated to ease the system tolerance to failures. The model presented here may be a basis for developing such a scheme of translations.

References

[Bazzi93] R.Bazzi, G.Neiger - Simplifying Fault-Tolerance: Providing the Abstraction of Crash Failures, *Technical Report GIT-CC-93/12*, Georgia Institute of Technology, 1993.

[Hadzilacos83] V.Hadzilacos - Byzantine agreement under restricted types of failures (not telling the truth is different from telling lies), *Technical Report 18-83, Aiken Computation Laboratory, Harvard University, 1983, Ph.D. disertation.*

[Hadzilacos84] V.Hadzilacos - Issues of Fault Tolerance in Concurrent Computations , *Technical Report 11-84, Aiken Computation Laboratory, Harvard University, 1984, Ph.D. disertation.*

[Lamport82] L.Lamport, R.Shostak, M.Pease - The Byzantine generals problem, *ACM Transactions on Programming Languages and Systems*, 4(3), pp.382-401, July 1982.

[Perry86] K.J.Perry, S.Toueg - Distributed agreement in the presence of processor and communication faults, *IEEE Transactions on Software Engineering*, 12(3), pp.477-482, March 1986.