Notes On Real-Time Distributed Database Systems Stability

Fabio A. Schreiber

Dipartimento di Elettronica - Politecnico di Milano - Italy

Abstract

Issues on the effect of routing and load balancing algorithms on the stability of a fault-tolerant distributed database system are examined. Degradation indexes are defined for the system dependability and some open question are risen.

1. Introduction

This paper deals with concepts related to algorithms as an important issue in the evaluation of the Dependability of Information and Control Systems. Such systems can be seen as very complex "machines" whose components belong to three different technological species:

1. hardware;
2. algorithms;
3. software (both system and application).

Mentioning algorithms as components in the evaluation of systems dependability, I want to separate algorithms themselves – even if in the broad meaning of problem resolution procedures – from their materialization by means of programs which can be buggy, in which case they would fall into species 3 [SCH 86].

To be more precise, I mainly refer to control algorithms, i.e. to those algorithms which can be found in the heart of Operating Systems, Database Management Systems, industrial process control systems, etc.. These algorithms, even if "correct" with respect to functional specifications, can influence the availability of the system to the end user "physiologically" - in the sense that their very function is to deny a user the access to a particular resource owing to instantaneous workload conditions (e.g. concurrency control). However algorithms can influence the overall system dependability "pathologically" since they can show a faulty behaviour under particular dynamic workload conditions and system constraints (e.g. load balancing, routing, etc. under tight response time constraints).

This research is partially supported by MURST research funds

The very simple algorithms we deal with in this paper are but examples of how algorithms can and must be accounted for in evaluating the overall system dependability. A more detailed description can be found in the references at the end of this paper.

1.1 Preliminary concepts

First of all, let us review some concepts which are closely related to "good behaved" algorithms:

Correctness: it is the fundamental property of whatever algorithm. An algorithm is correct if for whatever legal input it produces an output in accordance with the functional specifications of the problem [BOY 81, GHE 87].

Efficiency: it is the property which deals with the practical implementability and with the overhead the algorithm introduce in the system. Generally it is measured in terms of number of computation steps (*time complexity*) or of memory occupation (*space complexity*) as asymptotycal functions of the dimension of the problem [HAR 87].

Besides these classical features, two more properties became of interest in the recent years:

Robustness: this property deals with the practical operation of a system. An algorithm is robust if it produces correct results or it allows on-the-way adjustments even in presence of (small) errors in the inputs or in intermediate results. It is very important to avoid back-up/restart operations for error conditions which can be easily dealt with, and to provide a friendly user interface for possible errors made by the human operator.

Safety: an algorithm is safe if for whatever input it will not produce "hazardous" results. This is a very important property for algorithms used in industrial process control, where wrong actions from the controlling computer are directly applied to the physical world, possibly with disastrous consequences, and cannot be recovered. In Information Systems, Safety takes a slightly different meaning and it is generally called *Security* [LEV 86, COM 84].

As to the system the algorithms are part of, two properties are of relevance:

Availability: a system is available if it correctly delivers its services to the user, whatever happens to its internal operation. Availability is measured as the probability that within a time interval, called the mission time, the system works [BAR 65].

Reliability: a system is reliable if it does not break at all. Reliability is measured as the probability that, at some time t after having been started, the system is still working, without any maintenance action being performed [BAR 65].

Two more properties are derived from the above definitions:

Fault-tolerance and *Fail-soft behaviour*. The former is related to the possibility that the system remains available even in the presence of faults; the latter that, in presence of faults, the system still partially works, but with reduced performance.

Availability and reliability are generically referred to as *Dependability*. We can observe that, while efficiency is related to the physiological behaviour, correctness, robustness, and safety are all involved with possible faults which affect system dependability.

In evaluating the dependability of a system, dynamical aspects (i.e. those features which depend on workload variations in time) are of great relevance [MEY 88]. In fact, the behaviour of many control algorithms is heavily affected by the workload feeded to the system, both as to its quantitative and qualitative features.

In fact, in a real-time system (e. g. for process control or for counter operation), the response time constitutes not only a parameter for performance evaluation, but it becomes a fundamental factor for the system reliability since, if a threshold T is exceeded, the system must be considered faulty [KOP 89]. Therefore, the behaviour of the response time t_r vs. the utilization factor (i. e. the workload) ρ of a computer resource becomes discontinuous, as shown in figure 1 [SCH 84]. The system therefore shows an unstable behaviour, since a *small* increase in ρ results in an *infinite* growth in t_r.

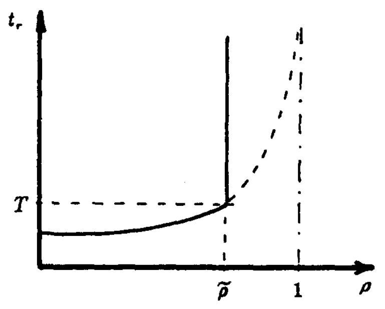

Fig. 1

In distributed computing systems, resource availability can be enhanced by replicating identical copies of the same resource at several nodes and by designing scheduling algorithms which can allocate requests to any available copy [STA 85]; this is a feature common to systems with a very different technological nature, as shown in [SCH 88]. In [KIN 89] procedures are described to manage such a routing process.

Usually the allocation is made with the aim of balancing the workload among the nodes. In presence of failures of any node, the workload is rerouted, if possible, to other available copies of the resources which were active on the faulty node (this model can be refined to consider failure of specific resources on the node instead of the node as a whole) [SCH 86].

While several papers address the issues of defining algorithms for load sharing with the aim of optimising the system behaviour [ELZ 86, TFG 88], in this paper we introduce a quantitative study of system degradation and of the possible arising of instability conditions.

2. A Model

Henceforth, we consider a distributed database on a real time system. Let us consider the following architectural variables:

$N = \{n_1, \ldots, n_n\}$ a set of *nodes* (processors) of a computer network;

$F = \{f_1, \ldots, f_m\}$ a set of *files* constituting the distributed database;

f_i^k the *copy* of file f_i on node k;

T_k the *threshold* response time of node k.

Each node can be in just one of two states: *working* or *failed*. The state of the system is determined by the set W of the working nodes:

$$W = \{n_i | n_i \text{ is working}\}$$

$$S_i = \{n_i | n_i \in W\}$$

A system state S_i is *compatible* if $S_i \neq \{\Phi\}$, where $\{\Phi\}$ is the empty set (absorbing state).

A system is *fully working* if

$$\forall f_i \ : \quad \exists f_i^k \wedge n_k \in W$$

i. e. at least a working copy of each file exists.

A system is *partially working* if

$$\exists f_i \ : \quad \{f_i^k\} = \{\Phi\}$$

i. e. some file disappeared.

A system is *failed* if

$$\forall f_i \;:\; \{f_i^k\} = \{\Phi\}$$

i. e. no file of the database survives.

In the case of a partially working system, we can define a number of *degradation indexes* which measure the amount of unavailability of the system. As an example we can define the

Brute Degradation Index =

$$BDI = \left(1 - \frac{\#\{working\ f_i\}}{\#F}\right)\%$$

i. e. the percentage of the number of working files with respect to the total number of files in the database. Another index could be the

Mass Weighted Degradation Index =

$$MWDI = \left(1 - \frac{\sum_{i\in W} \#f_i}{\sum_{i=1}^{F} \#f_i}\right)\%$$

when the larger files are given more importance than the smaller ones. Other such indexes can be defined at will. In particular, we could define weighted indexes for each transaction, where the weight is *1* for the files interested in the transaction and *0* for the others;

Transaction(j) Weighted Degradation Index =

$$TWDI(j) = \left(1 - \frac{\sum_{i\in W} p_{ij} * \#f_i}{\sum_{i=1}^{F} p_{ij} * \#f_i}\right)\%$$

$$\text{where } p_{ij} = \begin{cases} 1, & \text{if } f_i \in \text{transaction } j; \\ 0, & \text{otherwise;} \end{cases}$$

the choice of such an index allows the evaluation of the availability of the system with respect to a specific transaction, as shown in [SCH 84].

More specifically, we can consider a *data-compatible* state as a state in which the system is at least partially working, since it is conceivable that the database occupies only a subset of the network nodes. A state in which the system is failed can be considered therefore as *data-absorbing.*

Let us give the following definitions:

- the transition $S_i \longrightarrow S_j$ is *stable* if S_j is data-compatible;
- the transition $S_i \longrightarrow S_j$ is *transitively stable* if there is a sequence of transitions $S_i \longrightarrow S_m \longrightarrow \cdots \longrightarrow S_j$ and S_j is data-compatible;

```
NAIVE_ASSIGNLOAD (L, W)
begin
    while $L \neq \{\Phi\} \wedge W \neq \{\Phi\}$
    do begin
        assign $L = \{l_1, \ldots, l_w\}$ to the available
        physical resources $n_1, \ldots, n_w$;
        evaluate $\rho_i,\ i = 1, \ldots, w$;
        if $\exists i,\ \bar{\rho}_i - \rho_i \leq 0$
        then begin
            $W = W - n_i$;
            NAIVE_ASSIGNLOAD (L, W);
            end;
        else exit;
        fi;
        end;
    od;
end;
```

Fig. 2

- the transition $S_i \longrightarrow S_j$ is *unstable* if S_j is data-absorbing.

Transitions between states can be induced by the failure or by the timing-out of a node. We suppose that the main reason for the timing-out is overloading and that no hot recovery/restart procedures exist on the single nodes (no special hardware). This means that the time constants of the recovery actions lie in the range of some minutes, while the time constants of the rerouting actions lie in the range of some hundreds of milliseconds; therefore the system, as to the effect we want to study, can be considered without repair. When there is a transition between two states and the workload can be switched to the copies of the files available at other nodes of the system, there is an increase of the utilization factors ρ_i of the latters, possibly causing some T_i to be exceeded. In this case another state transition is triggered and so on until either the system reaches a stable state, possibly a partially working one, or it fails. In the latter case we say the system is *unstable* [SCH 88].

The repartiton strategy of the workload among the equivalent resources can be represented by a *Dependence Graph DG* = $\{V,\ A,\ L\}$ [SCH 84] where:

- each vertex V represents a processing node and the files stored on it;
- each arc A connects two vertices iff the resources they represent are equivalent;
- labels L placed on arcs represent the amount of load which can be transferred between the vertices.

Figure 2 shows a naive algorithm implementing such a load routing policy. Wiser policies make information on the loading state of each node available to all the others and use this information to do adaptive load assignment [ELZ 86, LiM 82], as shown in figure 3. Activation conditions for both algorithms could be as follows, where $XX = \{NAIVE,$

```
WISE}:
    on new workload
    do $L = L + l_i$; XX_ASSIGNLOAD (L,W);
    on failure of a loaded resource
    do $W = W - w_i$; XX_ASSIGNLOAD (L,W);
```

A description in terms of a feedback loop of such a behaviour is shown in figure 4. It has been shown in [SCH 88] that this model applies to very different engineering systems.

```
WISE_ASSIGNLOAD (L, W)
begin
    while $L \neq \{\Phi\} \wedge W \neq \{\Phi\}$
    do begin
        if $\exists I,\ I \subseteq W \wedge \forall i \in I \Longrightarrow \tilde{\rho}_i - \rho_i > 0$
        then begin
            try to assign L to nodes in I in such a way as
            nowhere $\tilde{\rho}_i$ is exceeded;
            if $\exists i,\ \tilde{\rho}_i - \rho_i \leq 0$
            then leave previous assignment as it is,
            call for flow control and exit;
            else exit;
            fi;
            end;
        else declare system crash and exit;
        fi;
        end;
    od;
end;
```

Fig. 3

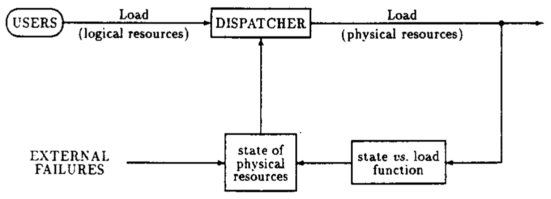

Fig. 4

3. Some Experimental Results

An analytic simulation package for systems based on the simple M/M/1 model treated in [SCH 84] has been built and it is described in [2AR 89]. Figure 5 is an output example which shows a plot of *BDI* against the mean utilization factor of the nodes in a database of 35 distinct files distributed, with replication, on 11 nodes. The rescheduling policy in this case simply consists in a *uniform* distribution among the surviving copies of the workload on the files belonging to a failed node. It can be seen that an increase in the timeout value *T* results in a lower degradation for a given workload. In another example, shown in figure 6, the higher timeout value allows the system to survive pretty well, with a low *BDI*, while the lower value induces complete failure.

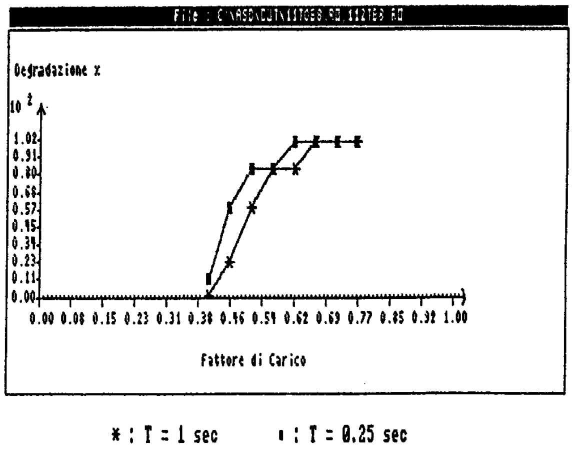

Fig. 5

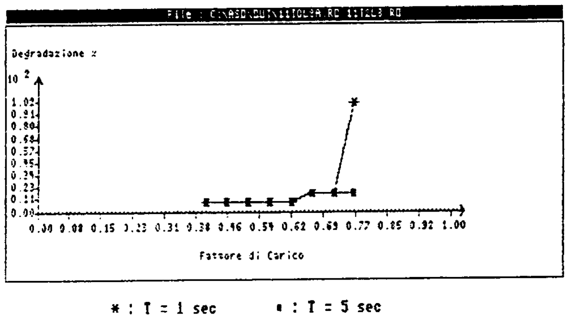

Fig. 6

4. Conclusions and future work

It comes, from the previous discussion, that the behaviour of the routing algorithm is vital for the dependability of the system; the transitions induced by the algorithm should be stable or at least transitively stable. We should speak in general of the dependability of control algorithms in the same way as of any other hardware or software component in the system, being cautious not to confuse it with the existence of software bugs.

The routing algorithms of our examples can be expressed both in iterative or recursive form, and we could put some upper bound on the number of iterations or of recursive calls to check for runaway. In particular, even without applying formal termination proofs, it is easy to find that, for some set of input data, algorithm *NAIVE_ASSIGNLOAD* eventually terminates with $W = \{\Phi\}$, thus showing intrinsic instability.

Moreover, if we release the non repairability constraint (i. e. we admit also the case that $W = W + n_i$), another well known form of dynamic instability of routing algorithms arises, since the workload can be continuously switched among the nodes without ever being processed. Such an "oscillating" behaviour, also known as "processor thrashing" [ELZ 86], sends t_r to infinity anyway. Again, is a check on the proper termination of the algorithm a sufficient test condition?

Qualitative features of the workload, such as the read/ update ratio, can influence the system dependability as well [SCH 86]. In fact, resilient updating operations usually result in a higher workload than queries, due both to updates broadcasts to multiple copies and to the concurrency control algorithms of the local DBMSs.

Far from bringing solutions, this contribution should raise a discussion on the role of the algorithmic components in evaluating and assuring the dependability of complex computer systems.

5. References

BAR 65 R. E. Barlow, F. Proshan – Mathematical Theory of Reliability – Wiley (1965)

BOY 81 R.S. Boyer, J. Strother Moore (Eds.) – The Correctness Problem in Computer Science – Academic Press (1981)

COM 84 Special issue on Software for Industrial Process Control - *IEEE Computer*, Vol. 17, n. 2 (1984)

ELZ 86 D. L. Eager et Al. – Adaptive Load Sharing in Homogeneous Distributed Systems – *IEEE Transactions on Software Engineering*, Vol. SE-12, n. 5, pp. 662-675 (1986)

GHE 87 C. Ghezzi, D. Mandrioli – Theoretical Foundations of Computer Science – Wiley (1987)

HAR 87 D. Harel – Algorithmics: the Spirit of Computing – Addison-Wesley (1987)

KIN 89 R. P. King et Al. – Management of a Remote Backup Copy for Disaster Recovery – *Princeton University CS-TR-198-88*, pp.28 (1989)

KOP 89 H. Kopetz et Al. – Distributed Fault-Tolerant Real-Time Systems: the MARS Approach – *IEEE Micro*, Vol. 9, n. 1, pp. 25-40 (1989)

LEV 86 N. Leveson – Software Safety: What, Why, and How – *ACM Computing Surveys*, Vol. 18, n. 2 (1986)

LiM 82 M. Livni, M. Melman – Load Balancing in Homogeneous Broadcast Distributed Systems – *Proc. ACM Comp. Network Performance Symp.*, College Park, pp. 47-55 (1982)

MEY 88 J.F. Meyer, L. Wei – Analysis of Workload Influence on Dependability – *Proceedings of FTCS-18*, Tokio, pp. 84-89, (1988)

SCH 84 F. A. Schreiber – State Dependency Issues in Evaluating Distributed Database Availability – *Computer Networks*, Vol. 8, n.3, pp. 187-198 (1984)

SCH 86 F. A. Schreiber – Information Systems: a Challenge for Computers and Communications Reliability – *IEEE Journal on Selected Areas in Communications*, Vol. SAC-4, n. 6, pp.1077-1083 (1986)

SCH 88 F. A. Schreiber, G. Rosolini – An Algebraic Description of Some State-Dependent Failure Mechanisms – *Information Processing Letters*, Vol. 29, n. 4, pp. 207-211 (1988)

STA 85 J. A. Stankovic – Stability and Distributed Scheduling Algorithms – *IEEE Transactions on Software Engineering*, Vol. SE-11, n. 10, pp. 1141-1152 (1985)

TFG 88 S. K. Tripathi et Al. – Load Sharing in Distributed Systems with Failures – *Acta Informatica*, Vol. 25, pp. 677-689 (1988)

2AR 89 A. Rocchetti, A. Rusconi – Affidabilità dei Sistemi Distribuiti – Progetto di laurea, Politecnico di Milano (1989)