NUMBING TRANSACTIONS TO AVOID STOPPING THE SYSTEM ACTIVITY*

R. de Juan Marín and L. Irun-Briz and F.D. Muñoz i Escoí

Instituto Tecnológico de Informática - Universidad Politécnica de Valencia
Camino de Vera, s/n - 46022 Valencia, SPAIN
Email: {rjuan, lirun, fmunyoz}@iti.upv.es

1. INTRODUCTION

Fault tolerance is one of the most tackle challenges in the field of Distributed Systems, which for critical-mission systems is moved up to its most exigent level, with the target of High Availability. For those systems, the main Quality of Service (QoS) measurement consists in providing High Availability. This goal requires to guarantee[1] that the system responsiveness is up and working during the 99.9999 % of time. And, in addition to the percentage of availability time, it is also critical for most systems the *length of failures* as another QoS measurement. This second measurement indicates the length of the unavailability periods when a failure arises. In a typical 24/7 application promising high availability, the provided services must be unavailable for less than 5 minutes per year. But in addition, for critical-mission applications, this availability promise will not be useful if a single failure produces an unavailability period of 5 minutes, since this is a lapse long enough for failing in its availability requirements. Consequently, bounding the *length of failures* conforms a complementary goal not only for critical-mission applications requiring 24/7 High Availability, but also for any Fault Tolerant system.

Update-Everywhere Replication[2][3][4] has proven to be an adequate way for providing fault tolerance, and hence, high availability. This technique ensures that a service provided by the distributed system can be served by several system nodes at any time, and hence, a failure only happens if all of the replicas fail, also benefitting performance.

An obvious and commonly accepted fact is that as a consequence of the High Availability promise, protocols[5] (either for replicating, recovering, or administering the distributed system) may allow any replica included in the system to be able to serve requests at any time. Even as a common goal of any Fault Tolerant System (not only for Highly Available systems), this remains true.

But sometimes, this is not possible, due to constraints introduced by the presence of transactional contexts, which in particular situations force the system to reach a point where no activity exists at any node, in order to perform special actions, as for example, an administration task, or a node recovery. Take as example an administration task executed to replace (or upgrade) the replication protocol used to maintain a distributed database. This change must be done synchronously at every system replica, since the replication protocol is a distributed piece of software which must be coherent along the system. Moreover, a protocol change must also take into account the running transactions at the replacement time. The management of the protocol change could become extremely complicated and costly if it were performed in presence of active transactions, since those active transactions should be "suspended" by the *old* protocol, and "restored" by the *new* one. To this end, the system must also maintain every single "transactional context" along the normal system work, and the new started protocol must recover it, thus degrading the overall system performance. Consequently, a reasonable approach consists in performing the replacement when the system contains no active transactions. To this end, a classical approach makes the system prevent the execution of new transactions at any node once the protocol replacement is requested, and hence, gradually stop the activity at each node in order to reach the needed inactivity condition. Obviously, this gradual stop of the activity degrades the system availability and performance, and this is the reason for most of authors to discard such alternative, and try to provide protocols including specific transaction-context reconstruction techniques. Unfortunately, these techniques, as shown above, introduce relevant overheads during the normal activity of the system.

As seen, there are many situations in which inactivity points are needed to perform certain tasks in a distributed system. In particular, for transactional distributed systems, as for example distributed databases, the achievement of those inactivity points comes with the necessity of keeping the context of every transaction running in the moment of performing the task. This suggests two alternatives: the first one, gradually stops the activity at any node and then performs the task; and the second alternative, intended for highly available systems, performs the task without stopping activity, but requires the continuous management of

*This work has been partially supported by the Spanish grant TIC2003-09420-C02-01.

transaction contexts, and their reconstruction when the system can continue after the task.

We announce here an alternative to reach inactivity points, which is not based on transaction-context reconstruction, at the time it avoids to stop the service at any replica for unconvenient long time periods. Our proposal is based on the schedule of an inactivity point, agreed by every replica, combined with a technique we name *numbing transactions*.

2. NUMBING TRANSACTIONS APPROACH

This approach is intended to be less aggressive than the stopping activity solution and simpler and less costly than the rebuild transaction approach. The underlying idea is to reach the inactivity time window numbing the start of new transactions at any node.

The work performed by the *numbing process* is as follows. When the inactivity condition request is broadcast to the system, each node run a deterministic algorithm to schedule: an *inactivity point* time that will be reached in a determined time t_0, and a *new-activity ban interval* that surrounds the inactivity point. Then, each new transaction only can be started if it is supposed, according to a statistical oracle and a probability threshold, to finish before the scheduled inactivity point is reached. But, once the new-activity ban interval is reached new transactions can not be started even if they are assumed to end before the inactivity point. The interval is in part used for dealing with the uncertainty of asynchronous (but bounded) networked systems.

Each node broadcasts a STOPPED message when it achieves the inactivity condition. If the system arrives to the scheduled inactivity point and there still exist ongoing transactions, the numbing process considers that this schedule has failed, and every node repeats the process, performing a new iteration. In each new iteration, the new scheduled inactivity point is longer, and the new-activity ban interval that surrounds it is also increased, in order to improve the success probabilities. The oracle threshold is also corrected, to improve the accuracy of the predictions. This process is repeated as long as the inactivity condition is not reached, guaranteeing that with a bounded number of iterations, the algorithm behaves equivalently to the stop activity approach commented previously. This is achieved by increasing the ban interval faster than the inactivity schedule.

3. ANALYSIS OF THE ALGORITHM BEHAVIOR

As it could be seen our approach increases the numbing level of the system until the non-activeness window is achieved. The goal of this behavior is double: to affect as least as possible the current work of the system, and to reach the inactivity system point as soon as possible. However, as it usually occurs, these two goals imply a tradeoff. Improving one implies getting worse the other one. In fact, using small inactivity intervals that will not affect very much the system work, implies smaller probabilities to reach the non-activeness point in the system. Whilst, high interval values increase the success probabilities at the cost of getting worse the system work.

We also perform an analytical study of the algorithm behavior under different scenarios. We use the notion of *load profile* in order to describe those scenarios, and its characterization is done through a number of parameters. The definition of a profile is $P = \{C_i\}$ with $C_i = (p_i, \bar{d}_i)$ where p_i is the probability of transactions of class C_i to appear, and $\bar{d}_i$ is the average length of transactions of class C_i. Our studies show the impact of the *average transaction length* of transactions, the *standard deviation* of the profile, the *compactness* of the profile (defined as the variation of lengths, as a generalization of the kurtosis of a statistical distribution), and others. Our results evidence a surprising independence of the algorithm behavior regarding most of the parameters, being the *compactness* the most relevant one.

Our experiments evidence that the overhead of the algorithm (which is only introduced during the inactivity condition search) can be kept fixed, at the cost of increasing the time the algorithm needs to reach the inactivity condition. Analogously, if the goal is to reach rapidly such condition, the overhead is increased, being the worst case equivalent to the stopping approach.

We also show a good balance between these two extremes, providing advantageous results for the most unfavorable profiles. The introduced overhead in such worst case remains below 30% in respect to the normal system performance, at the time the inactivity condition is reached in less than 7 times the average transaction length (atl) in this worst case. For a common profile, the inactivity condition is reached in 3 times the *atl*, with overheads below 8% over the normal system performance.

4. REFERENCES

[1] B. W. Lampson and H. E. Sturgis, "Crash recovery in a distributed data storage system," Tech. Rep., 1979.

[2] Jim Gray, "Why do computers stop and what can be done about it?," in *SRDS*, 1986, pp. 3–12.

[3] K.P.Birman, "Replication and fault-tolerance in the ISIS system," in *Pr.10th ACM SOSP*, 1985, pp. 79–86.

[4] P.A.Bernstein, V.Hadzilacos, and N.Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987.

[5] P.A.Bernstein, "Middleware: A model for distributed system services," *Comm. of the ACM*, vol. 39, 1996.