Termination Detection in an Asynchronous Distributed System with Crash-Recovery Failures

Felix C. Freiling[1], Matthias Majuntke[2], and Neeraj Mittal[3]

[1] University of Mannheim, D-68131 Mannheim, Germany
[2] RWTH Aachen University, D-52056 Aachen, Germany
[3] The University of Texas at Dallas, Richardson, TX 75083, USA

1   Termination detection

In practice, it cannot easily be detected whether a computation running in a distributed system has terminated or not. Thus, suitable observing algorithms are required to solve this problem of *termination detection*.

A termination detection algorithm involves a computation of its own and the computation it observes without interfering it. Additionally, it satisfies two properties: (1) it should never announce termination unless the underlying computation has in fact terminated. (2) If the underlying computation has terminated, the termination detection algorithm should eventually announce termination.

For the definition of *termination*, the states of processes are mapped to just two distinct states: active and passive. An active process still actively participates in the computation while a passive process does not participate anymore unless it is activated by an active process. In message-passing systems, which we also assume here, activation can only be done by receiving a message. A widely accepted definition of termination is that (1) all processes are passive and (2) all channels are empty.

Related Work Many algorithms for termination detection have been proposed in the literature (see the overview by Matocha and Camp [1]). Most of them assume a perfect environment in which no faults happen. There is relatively little work on fault-tolerant termination detection (e.g. [2,3]). All this work assumes the crash-stop failure model meaning that the only failures which may occur are crash faults where processes simply stop executing steps.

2   Problems in the crash-recovery model

In this paper we revisit the termination detection problem in the more severe crash-recovery failure model. Roughly speaking, in the crash-recovery model, processes are allowed to crash just like in the crash-stop model but they are also allowed to restart their execution later. We are unaware, however, of any termination detection algorithm for the crash-recovery model.

Solving the termination detection problem in the crash-recovery model is not an easy task. First of all, it is not clear what a sensible definition of termination is in the crash-recovery model. On the one hand, the classical (fault-free) definition of termination as mentioned above is clearly not suitable: If an active process crashes, there is always the possibility that it recovers later but there is no guarantee that it actually will recover. So an algorithm is in the dilemma to either making a false detection of termination or to possibly waiting infinitely long. On the other hand, the definition used in the crash-stop model is also not suitable: An algorithm might announce termination prematurely if an active process which was crashed recovers again. As a strict generalization, we introduce the definition of *robust-restricted termination*: (1) all alive and temporarily crashed processes have to be passive and (2) all the channels towards such processes have to be empty. Only crashed processes that will never recover, need not to be taken into account here.

Second, detecting robust-restricted termination in a crash-recovery system — even equipped with failure detectors — proves to be impossible to solve. Termination detection can be reduced to the problem of implementing a failure detector which is able to predict the future — of course not being feasible. Thus, we introduce the notion of *stabilizing* termination detection in which false termination detection announcements are allowed and may be revoked a finite number of times. The restriction to the *stabilizing* crash-recovery model in which all processes eventually either stay up or stay down (that is, the crash-recovery model eventually behaves like the crash-stop model) is also necessary. We present an algorithm for solving the stabilizing termination detection problem in the stabilizing crash-recovery model that uses a failure detector which is strictly weaker than the perfect failure detector [4]. The main idea of the algorithm is that every process logs the messages it sends and receives. By exchanging this information every process knows which messages it still has to expect. When a passive process does not expect any messages — its incoming channels are empty — it proposes to announce termination using a broadcast primitive. Termination is actually announced, if all live processes agree on announcing termination.

In summary, the results give insight into the additional complexities induced by the crash-recovery model in contrast to the crash-stop model.

References

1. Matocha, J., Camp, T.: A Taxonomy of Distributed Termination Detection Algorithms. J. Syst.Softw. **43**(3) (1998) 207–221
2. Wu, L.F., Lai, T.H., Tseng, Y.C.: Consensus and Termination Detection in the Presence of Faulty Processes. In: ICPADS, Hsinchu, Taiwan (1992) 267–274
3. Mittal, N., Freiling, F., Venkatesan, S., Penso, L.D.: Efficient Reduction for Wait-Free Termination Detection in a Crash-Prone Distributed System. In: DISC, Cracow, Poland (2005) 93–107
4. Majuntke, M.: Termination Detection in Systems Where Processes May Crash and Recover. Diploma Thesis, RWTH Aachen University (2006) https://pi1.informatik.uni-mannheim.de:8443/pub/research/theses/diplomarbeit-2006-majuntke.pdf.