Journal of Computer and System Sciences 96 (2018) 50–73

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Practically-self-stabilizing virtual synchrony[☆]

Shlomi Dolev[a,1], Chryssis Georgiou[b], Ioannis Marcoullis[b,*,2], Elad M. Schiller[c]

[a] Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel
[b] Department of Computer Science, University of Cyprus, Nicosia, Cyprus
[c] Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden

A R T I C L E   I N F O

Article history:
Received 11 July 2016
Received in revised form 19 January 2018
Accepted 11 April 2018
Available online 21 April 2018

Keywords:
Practically-self-stabilization
Virtual Synchrony
Practically unbounded counters
State machine replication

A B S T R A C T

The virtual synchrony abstraction was proven to be extremely useful for asynchronous, large-scale, message-passing distributed systems. Self-stabilizing systems can automatically regain consistency after the occurrence of transient faults. We present the first practically-self-stabilizing virtual synchrony algorithm that uses a new counter algorithm that establishes an efficient practically unbounded counter, which in turn can be directly used for emulating a self-stabilizing Multiple-Writer Multiple-Reader (MWMR). Other self-stabilizing services include membership, multicast, and replicated state machine (RSM) emulation. As we base the latter on virtual synchrony, rather than consensus, the system can progress in more extreme asynchronous executions than consensus-based RSM emulations.

1. Introduction

Virtual Synchrony (VS) is an important property provided by several Group Communication Systems (GCSs) that has proved to be valuable in the scope of fault-tolerant distributed systems where communicating processors are organized in process groups with changing membership [3]. During the computation, groups change allowing an outside observer to track the history (and order) of the groups, as well as the messages exchanged within each group. The VS property guarantees that any two processors that both participate in two consecutive such groups, should deliver the same messages in their respective group. Systems that support the VS abstraction are designed to operate in the presence of fail-stop failures of a minority of the participants. Such a design fits large computer clusters, data-centers and cloud computing, where at any given time some of the processing units are non-operational. Systems that cannot tolerate such failures degrade their functionality and availability to the degree of unuseful systems.

Group communication systems that realize the VS abstraction provide services, such as *group membership* and *reliable group multicast*. The group membership service is responsible for providing the current *group view* of the recently live and connected group members, i.e., a processor set and a unique *view identifier*, which is a sequence number of the view

[☆] An extended abstract and a technical report of this paper appeared in [1] and [2], respectively.
[*] Corresponding author.
E-mail addresses: dolev@cs.bgu.ac.il (S. Dolev), chryssis@cs.ucy.ac.cy (C. Georgiou), imarco01@cs.ucy.ac.cy (I. Marcoullis), elad@chalmers.se (E.M. Schiller).
[1] The research was partially supported by the Rita Altura Trust Chair in Computer Sciences; the Lynne and William Frankel Center for Computer Science; the Ministry of Foreign Affairs, Italy; the grant from the Ministry of Science, Technology and Space, Israel, and the National Science Council (NSC) of Taiwan; the Ministry of Science, Technology and Space, Infrastructure Research in the Field of Advanced Computing and Cyber Security; and the Israel National Cyber Bureau.
[2] The research is partially-supported by a Doctoral Scholarship program of the University of Cyprus.

https://doi.org/10.1016/j.jcss.2018.04.003
0022-0000/© 2018 Elsevier Inc. All rights reserved.

installation. The reliable group multicast allows the service clients to exchange messages with the group members as if it was a single communication endpoint with a single network address and to which messages are delivered in an atomic fashion, thus any message is either delivered to all recently live and connected group members prior to the next message, or is not delivered to any member. The challenges related to VS consist of the need to maintain atomic message delivery in the presence of asynchrony and crash failures. VS facilitates the implementation of a replicated state machine [3] that is more efficient than classical consensus-based implementations that start every multicast round with an agreement on the set of recently live and connected processors. It is also usually easier to implement [3].

Transient faults. Transient violations of design assumptions can lead a system to an arbitrary state. For example, the assumption that error detection ensures the arrival of correct messages and the discarding of corrupted messages, might be violated since error detection is a probabilistic mechanism that may not detect a corrupt message. As a result, the message can be regarded as legitimate, driving the system to an arbitrary state after which, availability and functionality may be damaged forever, requiring human intervention. In the presence of transient faults, large multicomputer systems providing VS-based services can prove hard to manage and control. One key problem, not restricted to virtually synchronous systems, is catering for counters (such as view identifiers) reaching an arbitrary value. How can we deal with the fact that transient faults may force counters to wrap around to the zero value and violate important system assumptions and correctness invariants, such as the ordering of events? A self-stabilizing algorithm [4] can automatically recover from such unexpected failures, possibly as part of after-disaster recovery or even after benign temporal violations of the assumptions made in the design of the system. To the best of our knowledge, no *stabilizing virtual synchrony* solution exists. We tackle this issue in our work.

Practically-self-stabilization. A relatively newself-stabilization paradigm is *practically-self-stabilization* [5–7]. Consider an asynchronous system with bounded memory and data link capacity in which corrupt pieces of data (stale information) exist due to a transient fault. (Recall that transient faults can result in the appearance of corrupted information, which the system tends to spread and thus reach an arbitrary state.) These corrupted data may appear *unexpectedly* at any processor as they lie in communication links, or may (indefinitely) remain “hidden” in some processor’s local memory until they are added to the communication links as a response to some other processor’s input. Whilst these pieces of corrupted data are bounded in number due to the boundedness of the links and local memory, they can eventually force the system to lose its safety guarantees. Such corrupt information may repeatedly drive the system to an undesired state of non-functionality. This is true for all systems and self-stabilizing systems are required to eradicate the presence of all corrupted information. In fact, whenever they appear, the self-stabilizing system is required to regain consistency and in some sense stabilize. One can consider this as an adversary with a limited number of chances to interrupt the system, but only itself knows *when* it will do this.

In this perspective, self-stabilization, as it was proposed by Dijkstra [8], is not the best design criteria for asynchronous systems for which we cannot specifically define *when* stabilization is expected to finish (in some metric like asynchronous cycles, for example). The newer criterion of practically-stabilizing systems is closely related to pseudo-self-stabilizing systems [9], as we explain next. Burns, Gouda and Miller [9] deal with the above challenge by proposing the design criteria of *pseudo-self-stabilization*, which merely bounds the number of possible safety violations. Namely, their approach is to abandon Dijkstra’s seminal proposal [8] to bound the period in which such violations occur (using some metric like asynchronous cycles). We consider a variation on the design criteria for pseudo-self-stabilization systems that can address additional challenges that appear when implementing a decentralized shared counter that uses a constant number of bits.

Self-stabilizing systems can face an additional challenge due to the fact that a single transient fault can cause the counter to store its maximum possible value and still (it is often the case that) the system needs to be able to increment the counter for an unbounded number of times. The challenge becomes greater when there is no elegant way to show that the system can always maintain an order among the different values of the counter by, say, wrapping to zero in such integer overflow events. Arora, Kulkarni and Demirbas [10] overcome the challenge of integer overflow by using non-blocking resets in the absence of faults described [10]. In case faults occur, the system recovery requires a blocking operation, which performs a distributed global reset. This work considers a design criteria for message passing systems that perform in a wait-free manner also when recovering from transient faults.

Note that, from the theoretical point of view, systems that take an extraordinary large number of steps (that accedes the counter maximum value, or even an infinite number of steps) are bound to violate any ordering constraints. This is because of the asynchronous nature of the studied system, which could arbitrarily delay a node from taking steps or defer the arrival of a message until such violations occur after, say, a counter wraps around to zero. Having practical systems in mind, we consider systems for which the number of sequential steps that they can take throughout their lifetime is not greater than an integer that can be represented using a constant number of bits. For example, Dolev, Kat and Schiller [6] assume that counting from zero to $2^{64} - 1$ using sequential steps is not possible in any practical system and thus consider only a *practically infinite period*, of 2^{64} sequential steps, that the system takes when demonstrating that safety is not violated. The design criteria of practically-self-stabilizing systems [11,5,7] requires that there is a bounded number of possible safety violations during any practically infinite period of the system execution. For such (message passing) systems, we provide a decentralized shared counter algorithm that performs in a wait-free manner also when recovering from transient faults.