

# Self-Stabilizing Autonomic Recoverer for Eventual Byzantine Software

(Extended Abstract)

Olga Brukman\*, Shlomi Dolev\*

Department of Computer Science, Ben-Gurion University, Beer-Sheva, 84105, Israel  
{brukman,dolev}@cs.bgu.ac.il

Elliot K. Kolodner

IBM Research Lab in Haifa, University Campus, Haifa 31905, Israel  
kolodner@il.ibm.com

## Abstract

*We suggest to model software package flaws (bugs) by assuming eventual Byzantine behavior of the package. In particular, the package has been tested by the manufacturer for limited length scenarios when started in a predefined initial state; the behavior beyond the tested scenario may be Byzantine. Restarts (reboots) are useful for recovering such systems.*

*We suggest a general yet practical framework and paradigm, based on a theoretical foundation, for the monitoring and restarting of systems. An autonomic recoverer that monitors and restarts the system is proposed, where: The autonomic recoverer is designed to handle different tasks given specific task requirements in the form of predicates and actions. DAG subsystem hierarchy structure is used by a consistency monitoring procedure in order to achieve gracious recovery. The existence and correct functionality of the autonomic recovery is guaranteed by the use of a kernel resident (anchor) process, and the design of the process to be self-stabilizing. The autonomic recoverer uses the new scheme for liveness assurance via on-line monitoring that complements known schemes for on-line ensuring safety.*

**Keywords:** *self-stabilization, Heisenbugs, automatic recovery, liveness, safety, monitors, restarters.*

## 1. Introduction

**Software Contains Bugs:** Computers that manage critical systems make the issues of correctness and faultless flow of long-lived and continuously running programs extremely important. Complex systems cannot be fully verified since verification of large systems may require an unreasonable amount of time and space. Such systems usually contain flaws – software bugs. The software industry tests software products extensively to eliminate flaws as much as possible. Usually software is tested by executing a large, but bounded and non-exhaustive, set of input/output with bounded length scenarios starting from a predefined initial state. Faulty, undesired and unplanned behavior may occur due to scenarios that were not tested prior to releasing the software and maybe hard to reproduce. Some of these flaws, called by some Heisenbugs, sometimes seem to disappear or alter their behavior when one attempts to probe or isolate them, because the probing process changes the execution environment. For example, the use of a debugger sometimes alters the operating environment significantly enough that faulty code, such as that which relies on the values of uninitialized memory, behaves quite differently. The software industry jargon includes many additional terms describing various types of bugs in running systems, e.g., resource leaks, indicating that these software systems contain software flaws (bugs).

**Bugs Maybe Harmful:** On the other hand, a consumer of critical systems would like to have a warranty that the system will operate as it should (e.g., [15, 13]). It is not always enough to be reimbursed when the software does not oper-

---

\* Partially supported by IBM faculty award, NSF grant 0098305, the Israeli ministry of defense, the Israeli ministry of Trade and Industry, and the Rita Altura trust chair in computer sciences.

ate properly. A software system for control of a nuclear reactor, that malfunctions may cause damage that is not on the same scale with the software cost. Consumer requirements may be categorized into safety requirements that ensure that nothing bad happens and liveness requirements that ensure that eventually something good happens [1].

**The Situation is Addressed:** The growing interest in autonomic computing (e.g., [19]), including automatic recovery (e.g., [24, 20]), self-managing systems, self-healing systems, and evolving systems, reflects the desire and need for robust and stable systems as opposed to systems that are optimized for performance.

**A Known Theory of Fault Tolerance:** Self-stabilization [12] is a strong fault tolerance property for systems that ensures automatic recovery once faults stop occurring. A self-stabilizing system is designed to start in any possible configuration where processors, processes, communication links, communication buffers, etc. are in an arbitrary state — arbitrary variable values, arbitrary program counter, etc.. The designer assumes that at least some of the programs are executed and proves that under his/her assumptions of program execution the system converges. Note that wrong programs, programs with bugs, may exhibit any behavior and therefore have no guarantees.

**Fault Models Should Reflect Reality:** Up to now, the research into self-stabilizing systems and also systems that model faults by Byzantine behavior has not coped with the fact that software packages contain bugs with very high probability. In particular, both self-stabilization and Byzantine theory limit the number of faulty processes that can be handled (e.g., [14]). Fortunately, software packages usually work as required for an execution period following their start from initial states. This initial correct behavior can be attributed to the testing done by the software manufacturer, starting the software in its initial state and considering bounded length executions. Software produced using this type of testing leads to scenarios where systems administrators and users occasionally reboot machines or processes to cope with the failure of long and continuously running systems. Self-stabilization and Byzantine theory could be enriched to exploit this phenomena.

**Our Approach in a Nutshell:** We design self-stabilizing schemes that add a monitoring layer to existing processes to ensure liveness and safety. We present system architecture that monitor and restart (reboots) subsystems automatically in an hierarchical manner. The hierarchal approach enables the restart of one subsystem, while other subsystems are unaffected. Therefore, only part of the system maybe rebooted.

**Monitoring Design:** For achieving liveness requirements we monitor heartbeats of processes; liveness cannot be

achieved when no process exists. New replacement processes are forked when missing processes are identified, see [5] for such an approach for non hierarchical system. Moreover, we will need to make sure that there exists at least one active monitor that is waiting for heartbeats in every system instance, no process replacement will take place when no process is active. We suggest a monitor, which is part of the resident kernel of the operating system of every processor, that is responsible to wake up the basic missing processes. In contrast, the scheme in [6] is based on a circular design, where the monitor and the restarter are mutually responsible for the existence of each other.

The growing use of commercial off-the-shelf software components, forces the system designer that integrates the components to rely on the correctness guarantees of the components' suppliers. Several techniques have been designed to limit the dependency of the integrator on the software component suppliers (e.g., object oriented design, runtime reflection techniques). Still there are legacy software packages that are not programmed according to these new guidelines. We propose to employ monitoring processes that record the input output sequence of each subsystem by using new monitoring techniques (e.g., runtime reflection or RTR), or by means of recording the system calls to the operating system (using e.g., unix/linux strace) made by the monitored subsystem. Our scheme ensures that the subsystems are alive and, moreover they are progressing towards the accomplishment of their mission. The monitoring processes will also check safety requirements, expressed as safety predicates. This scheme will prevent process from performing illegal operations that violate the safety requirements. Note that our additional layer should be thin enough to allow a full correctness proof of its code, and should also be self-stabilizing itself in order to ensure eventual recovery.

**Integrating Theory and Current Practical Approaches:** Current research has explored hierarchical restart as a tool for achieving high system availability [24, 17, 8, 7]. In the course of this research, some systems have been built [6, 5]. These systems demonstrate the effectiveness of restart, but these results are isolated in the sense that they are not integrated into a single framework. Moreover, there is no tie to rich fault tolerance theory, namely self-stabilization and Byzantine faults, that lead to the precise definition of desired properties and hence provable design and performance. We list several specific drawbacks of known techniques: (1) the hierarchies considered in [6] were an empty graph and tree only; (2) process monitoring is based only on heartbeats sent by processes; thus, checking that the process or subsystem is active, but not that it is actually in a consistent state and making progress; and (3) the monitoring-restarting layer itself may crash, there is no anchor monitor

in the kernel as we suggest.

We suggest a general yet practical framework and paradigm, based on a theoretical foundation, for the monitoring and restarting of systems. Our work also addresses the drawbacks outlined above. We employ a rich system hierarchy DAG structure instead of a tree or even no hierarchy at all [6, 18]. We include the monitoring and restarting of subsystems by using predicates and actions, as opposed to restart only when the process does not send its heartbeat [5]. We propose a generic distributed architecture for any type of system, as opposed to the narrow case of data structure monitoring suggested in [11]. We also propose schemes for liveness assurance via on-line monitoring that complements the scheme for ensuring safety presented in [15].

The rest of the paper is organized as follow. The system settings are described in Section 3. Section 4 present the different components and approaches used for implementing the autonomic recoverer. In Section 5 we demonstrate the autonomic recoverer using the well known tournament mutual exclusion algorithm. In Section 6 we describe our experience in implementing autonomic recoverer for printer systems. Section 7 concludes with a few remarks.

## 2. Related Work

### 2.1. ROC Project

[17, 24, 7, 8]

**2.1.1. Medusa** Medusa [5] is a virtually distributed system composed of arbitrary number of very stable independent processes. Each process has two functions. The first function is monitoring one of the user applications running, making sure it's alive. The second function is monitoring another such processes in the environment and keeping record of them. Upon detection of other failed process in the environment, the process will initiate the failed process restarting with some state information it has recorded.

**2.1.2. Mercury** Mercury [6] was created to improve satellite ground station availability. This system, unlike Medusa, supports component dependency. It has a predefined restart tree structure - a hierarchy of restartable control nodes and components, in which components are leaves in subtrees rooted at control nodes, which are highly fault-isolated. Restart at some component will cause restarting of the entire corresponding subtree rooted at closest control node. The restart tree structure is carefully calculated, based on profound statistics, to minimize mean time to recover and maximize mean time to fail.

### 2.1.3. JAGR:An Autonomous Self-Recovery Application Server [9]

## 2.2. On Modeling and Tolerating Incorrect Software

[2]

## 2.3. Kinesthetics eXtreme

[20]

## 2.4. Automatic Detection and Repair in Data Structures

[11]

## 2.5. Mictosoft Cluster Service

[26]

## 2.6. Closed Loop Design for Software Rejuvenation

[18]

## 3. The System Architecture

The *system* is represented by a set of *processes* that are an abstraction of applications executed by *processors*. When convenient, we also include *phantom processes* in the set of the processes. The phantom processes are in fact a set of global variables (or *locations*) that reside in the memory of the processors (and have some common semantics). A processor is a multitasking entity that may execute several processes. Each process is modeled by a state machine that executes *atomic steps* of *program*, which may be buggy. An atomic step  $a = \langle j, s, s', io \rangle$ , is a state transition from  $s$  to  $s'$ , by a process  $p_j$  together with internal calculations and input or output operation (*io*) which, is in fact interaction with other processes. The system supports also event driven operation, where during the execution of an atomic step an invocation of an interrupt called *event* that enforces the order of the next processes to be activated takes place. The communication capabilities of processes is defined by a directed communication graph  $G(V, E)$  where an edge  $(i, j)$  denote the possibility of  $p_j$  to receive information

from  $p_i$  by means of messages or shared memory (see [12] for more details on modeling distributed systems). The *system configuration* consists of a vector  $s_1, s_2, \dots, s_n$ , of the states of the processes in the system, and the contents of the communication devices, either the contents of the messages queues  $m_{1,2}, m_{1,3}, \dots, m_{i,j}, \dots$ , or the shared communication registers  $r_{1,2}, r_{1,3}, \dots, r_{i,j}, \dots$ . An *execution* is a sequence  $E = c_1, a_1, c_2, a_2, \dots$  of configurations  $c_i$  and atomic steps  $a_i$  such that  $c_{i+1}$  is reached from  $c_i$  by the execution of  $a_i$ . An execution  $E$  is *fair* if every process executes a step infinitely often in  $E$ .

A *subsystem* is a set of dependent processes where subsystems can be nested according to an *hierarchical directed acyclic graph* defined by the system designer. A subsystem may include one or more processes.

*Software/task specification function* is a function  $ss(I) = IO$ , where  $I \in \mathcal{I}$  is a particular sequence of inputs in the set  $\mathcal{I}$  of all possible (finite or infinite) sequences of inputs, and  $IO \in \mathcal{IO}$  is a particular sequence  $i_1, o_1, i_2, o_2, \dots$  of alternating inputs and outputs in a set  $\mathcal{IO}$  of such sequences. The set  $\mathcal{IO}$  defines the desired behavior of the software/task. A (sub)system  $sub_i$  *respects its specification function*  $ss_i$  in an execution  $E$  with inputs outputs sequence  $IO$ , if  $IO \in \mathcal{IO}$ .

The system is dynamic in the sense that processes, processors and the communication graph may be changed during the execution of the system. Still it is assumed, for obvious reasons, that the system consists of at least one processor. Furthermore, we assume that any processor has exactly one special process, which we call the *OS-resident Monitor and Restarter* (OMR)<sup>1</sup>. The program of this monitor process is formally verified to be correct according to the task it has to fulfill (as we describe in the sequel), and is also self-stabilizing. In other words starting in any possible state of the OMR the OMR eventually fulfills its task. The operating system of a processor is designed to ensure that the OMR is present and active e.g., the clock interrupts of a processor may trigger activation of the OMR that resides in a ROM. The OMR ensures that there are monitor processes for each task.

For the sake of proving correctness we assume that a reboot/restart/fork action of a process/subsystem results in process/subsystem that respects its specification function  $ss$  forever. Since we prove correctness from an arbitrary initial state, we will satisfy the requirements (that we state next) in every long enough period in which restarted subsystem respects its specification function (this proof technique is frequently used for proving self-stabilization [12]).

<sup>1</sup> Techniques for implementing the OMR appear in [16].

**Definition 3.1** An execution  $E$  is (*restart-supporting fair execution*) *rsf-execution* iff  $E$  is a fair execution in which every subsystem  $sub_i$  that is initialized during  $E$  respects its specification function  $ss_i$ .

We are now ready to state the requirements.

**Requirement 3.1** Every *rsf-execution*  $E$  has a suffix in which the system respects its specification function  $ss$ .

## 4. The Autonomic Recoverer

In this section we detail the different components and approaches used for implementing the autonomic recoverer. We start with a definition for the OS-resident monitor and restarter that serves as an anchor for the activity of the autonomic recoverer.

### 4.1. OS-Resident Monitor and Restarter

Monitoring and recovering is performed by monitoring processes that trace the activity of existing processes and groups of processes that form subsystems, and initiate restart actions when required. We must guarantee the existence of these monitoring processes. We suggest to have a single operating system resident process called *OS-resident Monitor and Restarter* (OMR) that is a permanent process of each operating system (or each computer). Every operating system will have to ensure that the OMR exists. Processors may run processes that achieves different independent *tasks* (or goals), for example one set of processes will be responsible for managing printing activity, others for communication among processors. The OMR will receive monitoring predicates and appropriate restart actions for each task, and will be responsible to continuously ensure the existence of monitoring processes for the tasks.

A *subsystem* is a group of processes that are correlated by consistency predicate. The processes constituting a subsystem do not necessarily reside in the same computer/operating-system. Thus, it is possible that a particular OMR will monitor subsystems that reside in several computers. In other words, while the responsibility for the existence of the OMR is machine based, and therefore may be guaranteed by the machine operating system, OMRs may monitor the well-being, and be responsible for restarting. The OMRs from different computers may communicate with each other to distribute responsibilities and to exchange portions of the state of the distributed subsystem.

The OMR must be *self-stabilizing* to ensure that eventually the monitor-restart process will execute their tasks, namely monitor the activity of the system and take restart

actions when appropriate. We assume that the operating system itself is self-stabilizing, ensuring the existence of the OMR. In addition, the OMR is a self-stabilizing process that can be started in an arbitrary state and eventually act according to the monitor predicates and restart actions.

## 4.2. Monitor Predicates and Recovery Actions

The autonomic recoverer is designed to handle computer system tasks when it receives *recovery tuples* for each task. Associate with each task is a set of *recovery tuples* in the form of  $\langle \text{monitor predicate}, \text{restart actions} \rangle$ .

The program of every monitor-restarter, including the OMR, is designed to use any set of the above recovery tuples, repeatedly evaluate the *monitor predicate* and activate the coupled *restart action* in case the monitor predicate holds. The monitor predicates trace the process states, namely the value of variables of the process and the way they are changed during the execution. In addition, monitor predicates for subsystems examine the variables of the composite automaton that includes the processes of the subsystem. The well-being of the subsystem as a whole is examined, namely the monitor predicates examine that the states of the processes in the subsystem are mutually compatible.

For each component, either a single application process or subsystem, there are two categories of predicates [1]. The first category is *safety problem predicates* (or in short safety predicates), these predicates ensure that nothing “bad” will happen. When a safety problem predicate holds, the system violates a “not to do command”. The complementary category is *liveness problem predicates* (or in short liveness predicates), which ensure that something “good” will happen. A liveness predicate does not hold if the process is alive and progressing toward performing the desired task. A liveness predicate should identify a violation of a “to do command” and therefore should identify livelocks or deadlocks of a subsystem.

Figure 1 depicts two processes, each represented by a square. The monitor-restarter processes are represented by a vertical rectangle while the set of recovery tuples that each monitor-restarter process uses as an input, is depicted as an horizontal rectangle. The recovery tuples are assigned to every monitor-restarter by the appropriate OMR. The OMR in turn, receives these tuples as an input from the designer of the system for the task. We note that it is possible to use an automatic generator of recovery tuples, though we do not address this issue here. The subsystem  $sub_1$  that consists of both the processes is monitored and restarted as well, the monitor-restarter process for this subsystem and the set of predicates appear in analogous location relative to the box that represents  $sub_1$ .

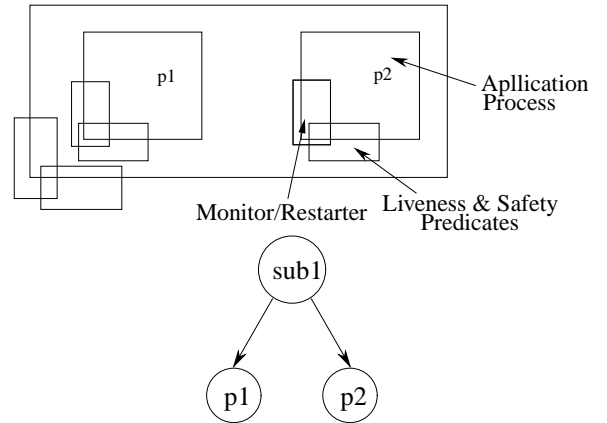


Figure 1: Tree Hierarchy of Subsystems

## 4.3. Gracious Hierarchical Recovery

The monitoring predicates of a subsystem  $sub_i$  assume that every subsystem/processes  $sub_j$  that is part of  $sub_i$  is in a *legal state*, namely that the monitoring tuples of  $sub_j$  do not hold. We suggest to use a *gracious scheduler* that notifies the monitor-restarter of  $sub_i$  to wait for completion of a restart when a monitoring predicate of  $sub_j$  holds and  $sub_j$  is being restarted. The gracious scheduler is responsible to allow a subsystem  $sub_i$  to complete the restart routine (after it is invoked) before the monitors start monitoring the state of  $sub_i$  again. Note that the subsystem predicates should reflect dependencies (hierarchy) and compatibilities of processes and subsystems rather than the activity of each of the subsystem components in isolation. The predicates of subsystems should be designed to reflect the subsystem hierarchy (in addition to the way the gracious scheduler respects the hierarchy). Namely, the scope of every predicate of a subsystem should not be a scope of one of its component. Altogether, restart will be performed at the lowest level of the hierarchy that is possible. The system hierarchy can be presented by a directed acyclic graph. Any subsystem may be a part of a bigger subsystem. Moreover, subsystems may share subsystems, without containing each other. Figure 2 depicts a situation in which  $sub_1$  is part of  $sub_2$  and  $sub_1$  is also a part of  $sub_3$ , but  $sub_2$  and  $sub_3$  do not contain each other.

Another aspect of the gracious recovery is the nature of the restart action taken when a monitor detects inconsistency. The restart actions can be defined by system supplier or by system user. Restart actions may vary, depending on process internal state at the time the restart procedure is started and on the previous restart actions taken. Restart actions could reinitialize process, roll-back process to the last safe state recorded, reschedule the process or kill processes.

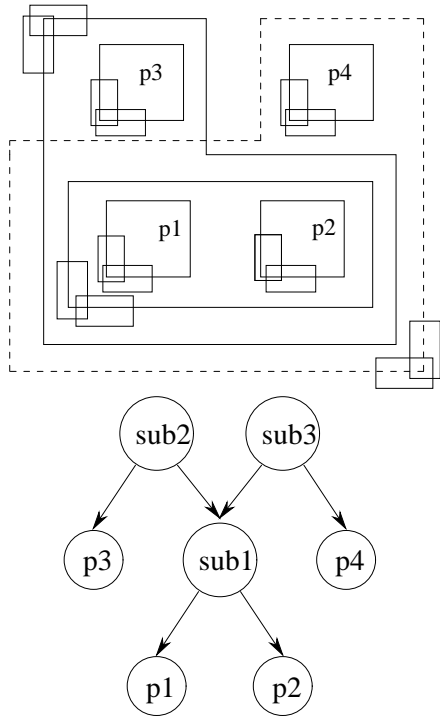


Figure 2: DAG Hierarchy of Subsystems

#### 4.4. On-line Safety and Liveness Assurance

On-line safety assurance techniques were suggested in [15], where a library of safety monitors is designed. We now present a complementary approach that can be used to assure liveness properties. Livelock can be detected by recognizing loop occurrence in the finite state machine that represents a subsystem. If livelock occurs during execution then the finite state machine that represents the subsystem will enter the same state twice, while there is no progress in spite of the activity of the subsystem. The idea is to examine on-line (important portions of) the state of a subsystem  $sub_i$ , to identify a subexecution,  $E_{circ} = c_1, c_2, \dots, c_j$ , where: (1) the (portion of) state of  $sub_i$  is identical in  $c_1$  and  $c_j$ , (2) the processes that run  $sub_i$  execute steps during  $E_{circ}$ , and (3) no progress has been done towards the subsystems task during  $E_{circ}$  (for example no process entered the critical section).

In the next lemma we use  $|S_i|$  to denote the number of possible states (variables values) for the (important portion of a) subsystem  $sub_i$ . The lemma essentially correlates the state portion size used to detect no-liveness with the length of history to be kept and time required for achieving the indication.

**Lemma 4.1** *In any execution  $E$  of  $|S_i| + 1$  or more config-*

*urations there exists a subexecution  $E_{circ} = c_1, c_2, \dots, c_j$  in which the state of  $sub_i$  in  $c_1$  and  $c_j$  is identical.*

**Theorem 4.2** *If there is  $E_{circ}$  then there is an infinite execution in which liveness does not hold.*

#### 4.5. Techniques for Implementing the Autonomic Recoverer

We detail here possible techniques for monitoring software packages. First we assume that the system designer receives a software package as a *black box* without knowing the precise implementation of the package. The autonomic recoverer uses the specifications/requirements defined for this package and the fact that the processes that execute the software of these package must have inputs and outputs, say in the form of system calls, to trace the activity of the package (see e.g., [22], unix/linux strace). There is a benefit in designing monitors that are not tailored to a specific implementation, the system may be changed over time while the monitor will ensure that the new version satisfies the task requirement. The term *evolving systems* is sometimes used for describing systems that change over time.

There are also technologies that allow run time observation of state. One such example is the run-time reflection (RTR) tool that accompanies modern software packages. We note that Java has library (java.lang.reflect) that provide RTR tools. Similar tools exist in C++ [25], in the CORBA and COM systems. More involved approaches for monitoring software systems may be achieved by using even more information on the implementation, e.g., using information in the form of the source code.

### 5. Autonomic Recoverer for the Tournament Mutual Exclusion

In this section we choose the  $n$  mutual exclusion tournament algorithm of [23, 3] to demonstrate a design of an autonomic recoverer for a specific task. We start with an intuitive description of the mutual exclusion tournament algorithm. There are  $n$  processes  $p_1, \dots, p_n$  that attempt entering the same critical section. A tournament approach is used to solve the mutual exclusion problem among these processes. Every pair of processes competes to get hold on (enter a critical section represented by) a specific leaf of the tournament tree. The algorithm used for this is essentially a mutual exclusion algorithm for two processes. Only one process of each pair succeeds entering (the critical section represented by each) one of the tree leaves. In general, for each tree node there are at most two nodes that compete

to enter the critical section represented by the node. A process that succeeds in entering a tree node competes (with at most one more process) to enter the parent node in the tree. The process that reaches the tournament tree root may enter the critical section of the algorithm.

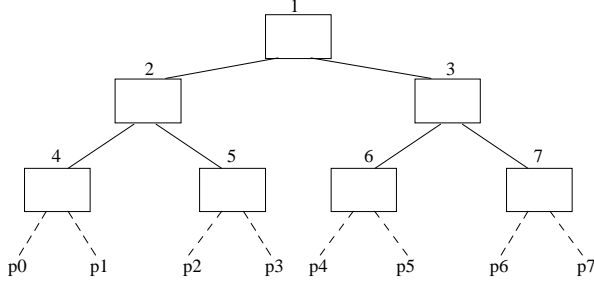


Figure 3: The tournament tree for  $n = 8$

Nodes of the tournament tree are numbered as follows. The root is numbered 1; the left child of a node numbered  $v$  is numbered  $2v$  and the right child is numbered  $2v + 1$ . The leaves of the tree are numbered  $2^k, 2^k + 1, \dots, 2^{k+1} - 1$ , where  $k = \lceil \log n \rceil - 1$ . Figure 3 is an example of the tournament tree for eight processes ( $n = 8, k = 2$ ). Each process is started by running the MAIN procedure described in Figure 4, lines 12-15. The communication operations (that include read/write access) in which atomic steps terminate are marked in Figure 4 by bold font letters. For example, the atomic step that starts in executing line 12 of Figure 4 terminates in the execution of line **f1** that immediately follows it, in which a value is written that triggers an event.

```

procedure NODE( $v$ : integer,  $side$ : {0..1})
f1: interact( $mlps_v, mps_i, flagEntry, time()$ )
1: Want $v$ [ $side$ ] := 0
2: wait until (Want $v$ [ $1 - side$ ] = 0 or Priority $v$  =  $side$ )
3: Want $v$ [ $side$ ] := 1
4: if (Priority $v$  =  $1 - side$ ) then
5:   if (Want $v$ [ $1 - side$ ] = 1) then goto line 1
6: else wait until (Want $v$ [ $1 - side$ ] = 0)
f2: interact( $mlps_v, mps_i, flagCritical, time()$ )
7: if ( $v = 1$ ) then //at the root
8:   (Critical Section)
9: else NODE( $\lfloor \frac{v}{2} \rfloor, v \bmod 2$ )
f3: interact( $mlps_v, mps_i, flagExit, time()$ )
10: Priority $v$  :=  $1 - side$ 
11: Want $v$ [ $side$ ] := 0
end procedure

procedure MAIN( $n, p_i$ )
12: if  $i$  is even
13:   NODE( $\lceil \frac{n+i}{2} \rceil, i \bmod 2$ )
14: else
15:   NODE( $\lceil \frac{n+i-1}{2} \rceil, i \bmod 2$ )
end procedure

```

Figure 4: The tournament algorithm

The algorithm, as is, does not cope with eventual Byzantine behavior of the processes. An eventual Byzantine process may block the other process advancing towards the root, may enter the critical section when other process is already executing it and may exhibit other undesired behavior. The task of the autonomic recoverer is to handle such scenarios and guarantee eventual legal execution.

```

procedure OMR( $\langle MonitorPred, RestartAct \rangle_1, \dots, \langle MonitorPred, RestartAct \rangle_N$ )
1: do forever
2:   for each  $1 \leq t \leq N$ 
3:     if  $\exists MonitorRestart(task_t)$ 
4:       fork  $MonitorRestart(\langle MonitorPred, RestartAct \rangle_t)$ 
end procedure

```

Figure 5: OS-resident Monitoring-Restarting Process for  $N$  tasks

The operating system constantly runs an OMR process that appears in Figure 5. The OMR is described by a list of monitoring predicates and restart actions for  $N$  different tasks that the user request the operating system to monitor. The OMR constantly checks if each of these tasks has a monitor-restarter process and if there is no such process it starts one. The monitor-restarter process of a task, described in Figure 6, is responsible for the existence of the task processes and its well-being. Each task has a list of predicates that the processes performing this task must not satisfy. If some predicate holds, the appropriate restart action will be enforced by the monitor-restarter. Both the OMR and the task monitor-restarter are generic and independent of the monitored tasks specifications. The recovery tuples, given as an input to the OMR are task dependent.

```

procedure MONITORRESTART( $\langle MonitorPred, RestartAct \rangle$ )
1: do forever
2:   for each  $mp_k \in MonitorPred \wedge ra_k \in RestartAct$ 
3:     if  $mp_k \wedge ra_k$ 
end procedure

```

Figure 6: Monitoring-Restarting Program

The tournament mutual exclusion algorithm is designed for achieving coordination among  $n$  processes, using  $n - 1$  tree nodes. [[ delete In the course of designing the autonomic recoverer, we found it convenient, not to monitor only subsystems that include subsets of the  $n$  processes of the tournament mutual exclusion algorithm, but also to monitor the variables that represent tree nodes.]] We monitor two types of processes, tournament processes and location processes. Tournament process,  $TP$ , is an original process competing for mutual exclusion. We assume that we may observe the value of the variable  $v$ , indicating the tournament tree node that represents the last critical section the

process won. Location process,  $LP$ , is in fact a phantom process (object store [4]) in the original algorithm which we found convenient to refer to.  $LP$  represents the variables used for implementing a node in the tournament tree, the *Priority* variable and the *Want* array. [[Subsystems include subsets of both tournament and location processes and defined as follows.]] For each node  $i$  in the tournament tree a subsystem  $sub_i$  is defined. If the tournament tree node  $i$  is a leaf, the subsystem consists of the node location process and the two tournament processes that may attempt entering this node. For any other node, the subsystem of the node consists of the node location process and the subsystems of the left and the right tree children.

A monitor-restarter process needs, in certain situation, to identify an activity of a process, for example, to identify the fact that the process entered the critical section. The monitor-restarter may miss such an event as it is scheduled to be executed independently. Therefore, flags that are set either directly by the original process, or as a side effect of its system calls or object store access (and later reset by the monitor as an acknowledgment), can serve the monitor-restarter to trace the activity of the process. We note that predicates that ensure that a process indeed executes the lines of its code in order can be added to the predicates that monitor the actions of the tournament processes in Figure 8. To simplify the presentation and the design we have used an hybrid approach in which the monitor-restarter of each process is both *event driven* and continuously working. Where during the execution of certain atomic steps an event occurs triggering actions of the monitor-restarter, in addition, the monitor-restarter is scheduled to execute steps even if no event triggers the step activity. Lines  $f_1, f_2, f_3$  are added to the NODE algorithm in Figure 4. Every time a process executes line  $f_i$  its monitor-restarter and respective location process monitor-restarter are activated with specific flag values and time of activation. Every time the monitor-restarters are activated they may take a snapshot of the process they are responsible for and keep the snapshot record as part of history record.

The monitor-restarter program is defined by a sequence of recovery tuples where we use  $mp_i$  to denote the  $i$ th monitoring predicate, and  $ra_i$  to denote the coupled restart action executed when  $mp_i$  holds. In addition we use  $ma_i$  to denote the  $i$ th monitor-restarter action, which is an action used for updating the monitor-restarter state, e.g., adding snapshot to history.

The recovery tuples used for the mutual exclusion task appear in Figure 7. The mutual exclusion task monitor-restarter will make sure that in any given moment there are  $n$  distinct tournament processes, lines  $mp_1-ra_1$ . Each of these  $n$  processes will have its monitor-restarter process, lines  $mp_2-ra_2$ . In addition the mutual exclusion

task monitor-restarter will ensure that there are  $n - 1$  location processes and their respective monitors-restarters, lines  $mp_3-ra_3$ . The functions  $processes(\langle process\ code \rangle)$  and  $forkProcesses(\langle process\ code \rangle, n)$  are used for ensuring the existence of the above processes. Function  $processes(\langle process\ code \rangle)$  finds all processes that run  $\langle process\ code \rangle$ . A possible way to implement  $processes(\langle process\ code \rangle)$  function is to have support from the operating system for accessing the value of the program counter of the processes, and using the program counter value as an indication for the code the process is currently executing.

The function  $forkProcesses(\langle process\ code \rangle, n)$  checks the existence of processes  $ps_1, ps_2, \dots, ps_n$  that are running the code  $\langle process\ code \rangle$  and if some of them do not exist the function forks them.

$\langle MonitorPred, RestartAct \rangle_{ME, n}$ $mp_1: \text{if }  processes(TP)  \neq n$ $ra_1: \quad \text{forkProcesses}(TP, n)$ $mp_2: \text{if } \exists ps_i \in processes(TP) \wedge \nexists \text{monitor}(ps_i)$ $ra_2: \quad \text{forkMonitorRestarter}(\langle MonitorPred, RestartAct \rangle_{ps_i})$ $mp_3: \text{if }  processes(LP)  \neq n - 1$ $ra_3: \quad \text{forkProcesses}(LP, n - 1)$ $mp_4: \text{if } \exists lps_i \in processes(LP) \wedge \nexists \text{monitor}(lps_i)$ $ra_4: \quad \text{forkMonitorRestarter}(\langle MonitorPred, RestartAct \rangle_{lps_i})$
--

Figure 7: Recovery tuples for the TME (tournament mutual exclusion) task.

The monitor-restarter of a tournament process appears in Figure 8. The liveness of a process is monitored by checking that the process makes no read/write accesses in spite of scheduling and granting CPU time for the process execution. Object store approach or system calls tracing can be used to detect such a situation. Note that a process that does not perform input/output actions on the memory must be deadlocked, and therefore needs to be killed, lines  $mp_1-ra_1$ . One safety concern is addressed by the *consistent* function that checks that the process follows the correct path in the tournament tree from the appropriate leaf to the root node,  $v_1$ , enters the critical section and then starts all over again. If a process is not following its correct path it is killed, lines  $mp_2 - ra_2$  in Figure 8.

Additional safety concern is that when process reaches the critical section, it does not stay in the critical section forever. This is in fact beyond the responsibility of the mutual exclusion task, since the mutual exclusion specification (e.g., no starvation) is defined assuming that no process stays forever in the critical section. Still, the application, whether it is printing, accessing a data base, or any other application, should provide us with predicates to detect whether a process that is in the critical section is faulty.

The function `inCSTooLong` uses these application dependent predicates to decide whether a process should be killed and then be restarted by the task monitor-restarter in Figure 7 lines `mp1-ra1`.

```

⟨MonitorPred, RestartAct⟩psi
ma1: historypsi := historypsi · snapshot(psi)
mp1: gotCPU(psi) ∧ not madeReadWriteRequests(psi)
ra1: kill(psi)
mp2: if not consistent(vpsi, historypsi)
ra2: kill(psi)
mp3: if inCSTooLong(historypsi)
ra3: kill(psi)

```

Figure 8: Recovery tuples for tournament process  $ps_i$

The location (process) monitor-restarter appears in Figure 9. The location monitor-restarter of location  $i$  monitors all tournament processes that obtained the critical section of location  $i$  (on the way to the root location, for which  $i = 1$ ) and the tournament processes that compete to enter its critical section. The location monitor-restarter may find the above set of tournament processes by, say, scanning the processes program counters and the value of their  $v$  variables, line `ma1`. The location monitor-restarter checks the history of these processes and verify that there are no two processes simultaneously in the critical section of the location. The monitor-restarter will kill the processes in the location if there are two or more processes in the critical section of the location; and then, if a similar configuration is later reached, the monitor-restarter may kill the entire subsystem, lines `mp2-ra2`.

A situation in which more than two processes are competing to enter (possibly one of them has entered) the critical section is addressed by lines `mp3-ra3`. Again, we use the procedure `kill` in a possible gracious fashion, first killing specific process and if the system does not recover the subsystem of the node is reinitialized.

In lines `mp4-ra6` the monitor-restarter deals with starvation scenarios. Let `firstLocationEntryv` be the earliest time, among the immediately proceeding times, in which a competing process in location  $v$  started executed line 2 of procedure `NODE` (called with the value  $v$ ). Assume `firstLocationEntryv` is entry time of the process  $ps$ . If after `firstLocationEntryv` time there were two entries into the location critical section, it means there were two times in a row in which a process(es) beat  $ps$  in location  $v$ , succeeding entering the location critical section while  $ps$  is waiting. Since the tournament mutual exclusion algorithm does not allow such a scenario (for the sake of preventing starvation) we do not allow it as well. `ra6` kills the processes or the subsystem gradually.

Lines `mp4` and `mp7-ra7` prevent deadlock situations. If there is only one process in a location then it means that this

process must succeed to enter the next location on the path to the root: there is no other competing process for the location critical section. Thus, if the process is the single process in location, the process makes some steps and stays in the same state, the process must be deadlocked (the process may repeat the same transitions forever without having any progress) and needs to be restarted. If there are two processes in location then one of them should make progress: enter the next location node on the path to the root node. As in case for one process, if there is no such progress, processes execute steps and their combined state stays the same for infinitely many configurations, then processes are deadlocked and need to be restarted.

If there is no such progress, it must be a lock of some kind. Therefore, there will be circular execution without progress. The processes combination is in a deadlock (the processes may repeat the same transitions forever without having any progress) and needs to be restarted. Again, restart maybe gradually executed.

```

⟨MonitorPred, RestartAct⟩lpsv
ma1: processesInLocationv :=
  {psj : psj ∈ processes(TP), v ∈ prefix(path(psj), vpsj)}
ma2: historylpsv := historylpsv · snapshot(lpsv)
mp2: if v = 1 ∧ twoCritical(history(lpsv))
ra2: kill(processesInLocationv, subsystemv)
mp3: if |processesInLocationv| > 2
ra3: kill(processesInLocationv, subsystemv)
mp4: if |processesInLocationv| > 0
ma5: firstLocationEntryv :=
  min{flagEntryv,psj | psj ∈ processesInLocationv}
mp6: if entry(history(lpsv)|firstLocationEntryv) > 1
ra6: kill(processesInLocationv, subsystemv)
mp7: if entry(history(lpsv)|firstLocationEntryv) ≥ 0 ∧
  ∃k, l | k < l ∧
  snapk, snapl ∈ history(lpsv)|firstLocationEntryv ∧
  snapk = snapl ∧
  steps(history(processesInLocationv))(k, l)
ra7: kill(processesInLocationv, subsystemv)

```

Figure 9: Recovery tuples for location process  $lps_v$

## 5.1. Correctness Proof

We now prove that the autonomic recoverer ensures that the system eventually fulfills the mutual exclusion requirements, namely, (1) in any configuration at most one process is executing the critical section (safety), (2) processes enter the critical section infinitely often (no deadlock), and (3) every process that continuously requests to enter the critical section eventually enters the critical section (no starvation).

Note that Lemma 5.3 could be concluded from Lemma 5.4, thus it would be enough to prove only Lemma 5.4. We will prove both lemmas for the readers' sake.

In the sequel we use the notation  $pc(p, v, c)$  for the value of the program counter of process  $p$  in configuration  $c$ , where  $v$  defines the (recursive) version of the NODE algorithm execution that we are interested in.

First we show that no process stays in the critical section forever.

**Lemma 5.1** *Every rsf-execution  $E$  has a suffix  $E'$  such that if there is a configuration  $c_i \in E'$  in which  $pc(p_k, 1, c_i) = 8$  then there is a configuration  $c_j \in E'$ ,  $j > i$ , such that  $pc(p_k, 1, c_j) \neq 8$ .*

**Proof:** Assume towards a contradiction that there is suffix  $E''$  of  $E$  for which  $\forall l \geq i$   $pc(p_k, 1, c_l) = 8$ . By lines  $mp_3$ - $ra_3$  in Figure 8, the process monitor-restarter will recognize the long stay of  $p_k$  in the critical section, and will kill  $p_k$ . Thus,  $E''$  does not exist. ■

In the next lemma we will show that the safety property holds: eventually no two processes enter the critical section simultaneously.

**Lemma 5.2** *Every rsf-execution  $E$  has a suffix  $E'$  such that  $\forall c \in E' \nexists p_i, p_j : pc(p_i, 1, c) = 8 \wedge pc(p_j, 1, c) = 8$ .*

**Proof:** Assume towards a contradiction that there are infinitely many configurations in  $E$  in which it holds that  $\exists p_1, \dots, p_k : \forall i \in 1..k$   $pc(p_i, 1, c) = 8$ . For the sake of the proof it is enough to observe only two processes that enter critical section simultaneously. By the fact that the tournament mutual exclusion algorithm has a finite number of tournament processes, namely  $n$  processes, there must be two specific processes,  $p'$  and  $q'$ , for which there are infinitely many configurations  $c \in E$ , such that  $pc(p', 1, c) = 8 \wedge pc(q', 1, c) = 8$ .  $p'$  and  $q'$  enter and, by Lemma 5.1, leave the critical section infinitely often during  $E$ . Consider a suffix  $E''$  of  $E$  in which every entrance of  $p$  ( $q$ ) to the critical section is followed by the execution of line  $f_3$  of Figure 4, by  $p'$  ( $q'$ , respectively).

Let  $c \in E''$  be a configuration such that  $pc(p', 1, c) = 8$  and  $pc(q', 1, c) = 8$ . Assume, without loss of generality, that  $p'$  was the last process to execute the atomic step in which line  $f_3$  has been executed. The execution of line  $f_3$  by  $p'$  activates  $mlps_1$ .  $mlps_1$  will find that the predicate  $mp_2$  in the location monitor-restarter (Figure 9) holds. Restart action  $ra_2$  will be taken: processes  $p'$  and  $q'$  will be killed. By the *rsf* property of the execution both the reinitialized  $p'$  and  $q'$  will respect their specification, changing states according to the code of the algorithm in Figure 4. Moreover, if any additional configuration in which two processes enters the critical section simultaneously occurs then action  $ra_6$  will initialize the entire system, and obviously by the *rsf* property of  $E$  the system will respect its specification. ■

The concern of the next lemma is liveness, in other words that infinitely often some process enters the critical section. The two next lemmas assume that all the  $n$  processes of the tournament mutual exclusion constantly compete for the critical section (when they are not in the critical section). The proof for the case in which only a subset of the processes compete for the critical section is similar.

**Lemma 5.3** *Every rsf-execution  $E$  has infinitely many configurations  $c \in E$  such that  $pc(p, 1, c) = 8$  for some process  $p$ .*

**Proof:** Suppose towards a contradiction that there is a suffix  $E'$  of  $E$  such that  $\forall c \in E' \forall p$   $pc(p, 1, c) \neq 8$ . Let  $v_h$  be the index of the smallest indexed location node such that there are infinitely many configurations  $c \in E'$  for which there exist processes  $p_i$  such that the value of  $pc(p_i, v_h, c)$  is in any line except 8 (the program counter can be 8 only if  $v_h = 1$ , but then  $pc(p_i, 1, c) = 8$  and the lemma holds). In other words,  $v_h$  is the smallest indexed node visited infinitely often by tournament processes. There must exist such node because tournament processes execute their code. In the worst case scenario processes enter only tournament tree leaves and then  $v_h = v_{2^{\lceil \log n \rceil - 1}}$  (leftmost tournament tree leaf). By the fact that the tournament mutual exclusion algorithm has a finite number of tournament processes, there must exist a particular process  $p$  that is in location node  $v_h$  for infinitely many configurations  $c \in E'$ .

There are two possible scenarios for such process  $p$ . In the second scenario due to its byzantine behaviour, process might enter other location node,  $v_l$ . By definition of  $v_h$ ,  $v_l > v_h$ . Either execution of line  $f_1$  of Figure 4 by process  $p$  or process scheduler will activate  $mps_p$ . The predicate  $mp_2$  of Figure 8 will hold: *consistent* function makes sure that process moves correctly on the path to the root node of the tree.  $mps_p$  will invoke  $ra_2$  of Figure 8. By the *rsf* property of  $E$  after restart  $p$  respects its specification function and moves correctly on the path to the root node. By this fact and by definition of  $E'$ , after restart  $p$  will reach  $v_h$  and will stay there, e.g.,  $p$  will follow the second scenario described below.

In the second scenario, process  $p$  stays in the location node  $v_h$ , i.e., there is a suffix  $E''$  of  $E'$  in which  $\forall c \in E''$   $pc(p, v_h, c) \in \{2, 6\}$ , where  $\{2, 6\}$  are indexes of lines of the code of Figure 4. The program counter has to be constant because otherwise  $p$  would not stay in the same location node.

By the fact that the tournament mutual exclusion algorithm has a finite number of tournament processes, namely  $n$  processes, there is an execution suffix in which the number of processes in any location  $v$  is fixed. Moreover, by  $mp_3$  of Figure 9 that is continuously monitored by  $lmps_v$

the number of processes in  $v$  is eventually at most two (if the number of process exceeds two then  $ra_3$  in Figure 9 kills the processes in  $v$ ).

To summarize, there exists suffix  $E_f$  of  $E''$  such that  $\forall c \in E_f$  : the set of processes in location  $v_h$  is of size one or two (by definition of  $v_h$  it can not be of size zero), each process in the set act accordingly to the second scenario. By the facts that

- (1) each process is a finite state machine, therefore a particular state combination must appear infinitely often in execution  $E_f$
- (2) the processes in location node  $v_h$  execute steps

$mlops_{v_h}$  (monitor-restarter of location node  $v_h$ ) that works continuously is able to recognize livelock situation (predicates  $mp_4$  and  $mp_7$  will hold). If at this point the history is corrupted, making it impossible for  $mlops_{v_h}$  see the situation correctly and initialize restart, once correct snapshots start to be accumulated in the history log then eventually  $mlops_{v_h}$  will be able to recognize livelock and restart processes. Therefore, restart action  $ra_7$  is invoked. Processes in location node  $v_h$  are killed and if the same situation occurs again the entire subsystem is reinitialized. ■

In the next lemma we show that a process that (constantly) requests entering the critical section eventually enters the critical section.

**Lemma 5.4** *Every rsf-execution  $E$  has a suffix  $E'$  such that  $\forall p_k \exists c_i \in E' : pc(p_k, 1, c_i) = 8$ .*

**Proof:** Suppose towards contradiction that there is a suffix  $E'$  of  $E$  such that some particular process  $p$  is the only process does not enter critical section, i.e. the only process for which there is no configuration  $c \in E'$  such that  $pc(p, 1, c) = 8$ . As in Lemma 5.3, let  $v_h$  be the index of the smallest indexed location node visited by some processes infinitely often, e.g. there are infinitely many configurations  $c \in E'$  in which the value of  $pc(p_i, v_h, c)$  for some process  $p_i$  is any line except 8 (the program counter can be 8 only if  $v_h = 1$ , but then  $pc(p, 1, c) = 8$  and the lemma holds). As we have shown in Lemma 5.3 eventually  $p$  will be “stuck” in location node  $v_h$ , e.g. there is a suffix  $E''$  of  $E'$  such that  $\forall c \in E'' pc(p, v_h, c) = 2, 6$ .

Again, as in Lemma 5.3, by the fact that the tournament mutual exclusion algorithm has a finite number of tournament processes, namely  $n$  processes, and by definition of  $v_h$  there is a suffix  $E_f$  of  $E''$  in which the number of processes in location  $v_h$  is one or two.

The set of processes in location node  $v_h$  can be either constant (the set of processes will include process  $p$  and

may be other constant process) or continuously changing (it will include  $p$  and some different process each time). Consider the case in which the set of the processes in the location node  $v_h$  is constant and includes only one process,  $p$ . Exactly as in the proof of Lemma 5.3, by the facts that

- (1) process  $p$  is a finite state machine, therefore a particular state combination must appear infinitely often in execution  $E_f$
- (2)  $p$  executes steps

$mlops_{v_h}$  (monitor-restarter of location node  $v_h$ ) that works continuously is able to recognize livelock situation (predicates  $mp_4$  and  $mp_7$  will hold). Restart action  $mp_7$  will be invoked, restarting process  $p$  or subsystem of location node  $v_h$  if situation repeats itself.

Consider the case in which set of processes is constant and includes two processes,  $p$  and some other constant process  $q$ . By definition of contradiction there must be infinitely many configuration  $c \in E_f$  such that  $pc(q, 1, c) = 8$ , meaning that process  $q$  constantly “over steps” process  $p$  and enters critical section.  $mp_6$  of Figure 9 that is continuously monitored by  $mlops_{v_h}$  will hold. Restart action  $ra_6$  of Figure 9 will be invoked, restarting both  $p$  and  $q$ . If after restart the situation will repeat itself  $ra_6$  will restart the subsystem of location node  $v_h$ .

If the set of processes constantly changes, including  $p$  and some different process each time,  $q_i$ , then again, by definition of contradiction there must be infinitely many configuration  $c \in E'$  such that  $pc(q_i, 1, c) = 8$ , meaning that processes  $q_i$  always “over step” process  $p$  and enter critical section. As in case for constant set of processes of size two, processes involved in starvation scenario ( $p$  and  $q_i$  first and if unsuccessful - subsystem of location node  $v_h$ ).

In all three cases (one process in location and two cases of two processes in location) corrupt history will not prevent from  $mlops_{v_h}$  to make a restart eventually. As it was explained in Lemma 5.3, once correct snapshots start to accumulate, the history will have a correct suffix. Eventually  $mlops_{v_h}$  will be able to recognize livelock and restart the processes.

By rsf property of  $E$  the above invocations of  $ra_7$  and  $ra_6$  by  $mlops_{v_h}$  will ensure that there will be infinitely many configuration in which process  $p$  will enter critical section. ■

## 5.2. Self-Stabilization

We now address the self-stabilization property of the system. Note that self-stabilization proof of the autonomic recoverer needed to be made for each monitored task, using

task recovery tuples. We assume that OMR is an integral part of the self-stabilizing operating system kernel [16], therefore, we can assume OMR's constant presence. OMR code is self-stabilizing, thus it can start working from arbitrary state, in this case this means it can start running from any line of the code. After completing one full loop it accomplishes its goal – every process in the system has monitor-restarter that checks appropriate recovery tuples.

Each process's/subsystem's monitor-restarter code is self-stabilizing as well. We can be sure that recovery tuples assigned to the monitor-restarter are correct after restart: each time restart takes place OMR will overwrite recovery tuples of the monitor-restarter. The arguments for monitor-restarter self-stabilization are similar to those of OMR. As OMR, monitor-restarter can start working from arbitrary state, meaning from any line of the code. Recovery tuples are correct after restart. After completing one loop, monitor-restarter will have checked all recovery tuples ensuring monitored process safety and liveness. In the worst case scenario, if monitor-restarter starts working from restart action line in which all processes in the system need to be restarted, the whole system is restarted.

This "chain" of self-stabilizing processes, OMR and then processes'/subsystems' monitors-restarters, makes sure that each process/subsystem is monitored with appropriate recovery tuples and that monitors-restarters will bring eventually every process/subsystem in the system to the safe state.

The autonomic recoverer state could be corrupted by assigning corrupted history to monitors-restarters. We note that history variables are initialized whenever restart takes place. Suppose the monitor-restarter starts with an arbitrary history. New (correct) snapshots are accumulated as a part of the history. Eventually each component will have a history with correct suffix. Then every monitor-restarter will see the true picture of the process state and act accordingly. Note that having corrupted history can cause monitors-restarters make an unnecessary restarts but it can not prevent restarts being made.

Another issue is designing recovery tuples in self-stabilizing manner. Tuples should be as independent of each other as possible, not to rely on the information observed in previous tuples, such that every tuple describes some safety or liveness concern. When monitor-restarter could start working from checking any arbitrary state.

## 6. Printers Experience

Partial implementation of a prototype for autonomic recoverer has been carried out and was used to solve a problem in a real life system [10]. The encountered problem was

printers availability in our department's printers system. If user has tried to print format inconsistent *pdf*, *postscript* or *doc* file, this file would stall the printer. After investigating the problem, we found out that the bottle neck lies in Line Printer Daemon (LPD) actions. When file is sent to be printed, it enters printing jobs queue. LPD continuously checks this queue, sends files to the appropriate printers and sees printing job being completed. In case of problematic file, first LPD sends it to the printer as usually. Printer is unable to print it due to format inconsistencies and notifies LPD about it's failure. Upon printer's failure, LPD is programmed to make a retry regardless of the printer's failure reason. LPD will send the problematic file to the same printer again, causing the scenario to repeat itself over and over again. The printer will be stalled by LPD retries and will not be able to print other jobs sent to this printer.

The solution till now was that when system administrator was called by users, he would recognize the problem and solve it by removing the problematic job from the printing jobs queue. Usually, both owner of problematic file and owners of other printing jobs as users send their jobs again and again, thinking that it "has been lost" by printing system. In this case, the problematic job could cause the trouble again. Additional problem is the waste of resources caused by resending of printing jobs by users. Paper, tuner, power and printing time that is wasted on duplicated printing could be enormous if files are big. LPD is a widely used printing server protocol and an integral part of printing systems in many organization and can not be replaced easily. Therefore, autonomic recoverer can be used efficiently to solve the problem.

Given the autonomic recoverer infrastructure we need only to design recovery tuples for LPD processes. In those tuples, predicate restrict the number of retries LPD can make for one file. If retries are exhausted, when one of the following restart actions would take place. First possible restart action is format transformation of the files, for instance from *pdf* to *postscript* or from *doc* to *postscript*<sup>2</sup> and trying to print the transformed file again. Another restart action is sending file to other, less format sensitive printer. Finally, if none of the above restart actions solved the problem or was possible the last resort restart action is sending error message to the printing job owner and removing the problematic file from printing jobs queue. The system has proved itself in solving the problem efficiently, improving printing system service and saving resources.

---

2 Our practical experience proved that transforming problematic file usually helps, probably because format transformation engines create strict and correct formats.

## 7. Concluding Remarks

Computer systems and their management are becoming increasingly complex. One of the most important challenges of the computing industry is the design and development of autonomic and self-healing systems to deal with this complexity. We believe that the theory foundations of self-stabilization could serve as a basis for the new paradigms required to build such systems. Our work takes a first step in this direction. We combine knowledge of self-stabilization, observations about how systems behave (they work when started from initial states, but their behavior decays and failures occur over time), and the ideas of hierarchical restart and monitoring to develop a theory and a practical framework and paradigm.

We devise a general scheme for the design of robust systems that can be used as a building block for autonomic systems. The integrated system design includes robust and modular monitors and restarters. The integrated system is self-stabilizing and therefore ensures automatic recovery.

**Acknowledgment:** we thank Shlomi Levi and Omri Cohen for implementing a prototype for an autonomic printer recoverer.

## References

- [1] B. Alpern and F. B. Schneider. "Defining Liveness", *Information Processing Letters*, vol. 21, Number 4, pp. 181-185, 1985.
- [2] A. Arora and M. Theimer. "On Modeling and Tolerating Incorrect Software", *CIRM Seminar on Self-Stabilization*, Luminny, France, 2002.
- [3] H. Attiya and J. L. Welch. "Distributed Computing: Fundamentals, Simulation and Advanced Topics", McGraw Hill Companies, pp.78-81, 1998.
- [4] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, L. Yerushalmi. "Towards an Object Store", *Proc. of the Twentieth IEEE / Eleventh NASA Goddard Conference on Mass Storage Systems and Technologies*, San Diego, April 2003.
- [5] G. Candea. "Medusa: A platform for highly available execution". Technical report, CS244C: Distributed Systems Lab (class project paper), Stanford University, June 2000.
- [6] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Garg, and R. Gowda. Reducing recovery time in a small recursively restartable system. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2002)*, Washington, D.C., June 2002.
- [7] G. Candea and A. Fox. "Designing for high availability and measurability". In *Proceedings of the 1st Workshop on Evaluating and Architecting System Dependability (EASY)*, Goteborg, Sweden, July 2001.
- [8] G. Candea and A. Fox. "Recursive restartability: Turning the reboot sledgehammer into a scalpel". In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS VIII)*, Elmau, Germany, May 2001.
- [9] G. Candea and E. Kiciman and S. Zhang and P. Keyani and A. Fox. "JAGR: an Autonomous Self-Recovering Application Server". In *Proceedings of the 5th International Workshop on Active Middleware Services (ASM-2003)*, Seattle, WA, June 2003.
- [10] S. Cohen and S. Levy and O. Brukman and S. Dolev. "Self-Stabilizing Autonomic Recoverer for Printing Systems". Technical report, 2003-18, Computer Science Department, Ben-Gurion University, August 2003.
- [11] B. Demsky and M. Rinard, "Automatic Detection and Repair of errors in Data Structures", Technical Report MIT-LCS-TR-875, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, December 2002.
- [12] S. Dolev. *Self-stabilization*. The MIT press, March 2000.
- [13] S. Dolev and Y. Haviv. "Self-Stabilizing Soft Error Resilient Microprocessor" *International Conference on Dependable Systems and Networks, IEEE Computer Society (DSN 2003)*, pp. B-18, B-19, 2003.
- [14] S. Dolev and J. L. Welch. "Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults," *Proc. of the Second Workshop on Self-Stabilizing Systems*, (WSS 1995), pp. 9.1-9.12, 1995. Also in *Proc. of the 14th Annual ACM Symp. on Principles of Distributed Computing (PODC 1995)*, pp. 256, 1995.
- [15] S. Dolev and F. Stomp. "Safety assurance via on-line monitoring," *Proc. of the Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001)*, pp. 101-108, 2001.
- [16] S. Dolev and R. Yagel. "Toward Self-Stabilizing Operating Systems", manuscript, 2003.
- [17] A. Fox and D. Patterson. "Self-Repairing Computers", *Scientific American*, June, 2003
- [18] Y. Hong, D. Chen, L. Li, K. S. Trivedi, "Closed Loop Design for Software Rejuvenation", *Workshop on Self-Healing, Adaptive, and self-MANaged systems (SHAMAN)*, 2002.
- [19] IBM. Autonomic computing. <http://www.research.ibm.com/autonomic>, 2001.
- [20] Gail Kaiser, Janak Parekh, Philip Gross and Giuseppe Valetto. "Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems". In *5th Annual International Active Middleware Workshop*, June 2003, pp. 22-30.
- [21] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", *ACM Trans. on Programming Languages and Systems*, Vol. 4, No. 3, pp. 382-401, 1982.
- [22] Mercury Interactive, LTD, <http://www-heva.mercuryinteractive.com>.
- [23] G. L. Peterson and M. J. Fischer, "Economical solutions for the critical section problem in distributed system", in *Proc. of the 9th ACM Symposium on Theory of Computing*, (1977), pp. 91-97.
- [24] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and

N. Treuhaft. "Recovery Oriented Computing(ROC): Motivation, definition, techniques and case studies", UC Berkeley Computer Science Technical Report UCB/CSD-02-1175, Berkeley, CA, March 2002.

[25] Reflection for C++. <http://www.grret.ru/knizhnik/cppreflection/docs/reflect.html>, 2003.

[26] W. Vogels and D. Dumitri and K. Birman and R. Gamache and M. Massa and R. Short and J. Vert and J. Barrera and J. Gray. "The Design and Architecture of the Microsoft Cluster Service - A Practical Approach to High-Availability and Scalability", *Proc. of the 28th International Symposium on Fault-Tolerant Computing (FTCS 1998)*, June, 1998