Computing and Informatics, Vol. 28, 2009, 3–28

RESOURCE AWARE RUN-TIME ADAPTATION SUPPORT FOR RECOVERY STRATEGIES

Rodica Tirtea

University of Oradea
Universitatii Street, 1
410087, Oradea, Romania
e-mail: rtirtea@uoradea.ro

Geert Deconinck

K. U. Leuven, ESAT/ELECTA
Kasteelpark Arenberg, 10
B-3001, Leuven, Belgium
e-mail: gdec@esat.kuleuven.be

Manuscript received 13 October 2005; revised 1 October 2007
Communicated by Ladislav Hluchý

Abstract. The selection of recovery strategies is often based only on the types and circumstances of the failures. However, also changes in the environment such as fewer resources at node levels or degradation of quality-of-service should be considered before allocating a new process/task to another host or before taking reconfiguration decisions. In this paper we present why and how resource availability information should be considered for recovery strategiesadaptation. Such resource aware run-time adaptation of recovery improves the availability and survivability of a system.

Keywords: Recovery strategy, fault tolerance, adaptation, resource monitoring, availability

1 INTRODUCTION

Distributed heterogeneous systems involved in the control of infrastructures, such as electric power infrastructure, need to ensure reliable services regardless of faults and changes in the environment [1]. At middleware level, fault tolerance architecture could incorporate mechanisms for adaptation to assure dependable control of the components of the infrastructure. Recovery strategies are used to allow reconfiguration of the system (e.g. graceful degradation) based on the circumstances of the failure.

The main purpose of the recovery is to improve dependability of the system. Recovery uses error detection and reconfiguration to enable the non-failed components to deliver acceptable services. As the adaptive fault tolerance mechanisms in distributed systems are incorporated at middleware level, the detection and recovery infrastructure is middleware-based (i.e. software).

Recovery strategies in distributed systems include actions such as migration or restart of processes/tasks on other nodes. Selection of an appropriate recovery strategy usually depends on the type of failure and the circumstances of the failure event. The available resources of the future host (for the migrating processes) are usually not taken into account, neither is the performance of the network. We consider that those run-time conditions, if neglected, can negatively impact the reliability of the fault-tolerant systems.

In this paper we present why and how *available resources* (e.g. memory, CPU) should be considered together with the type of failure and the circumstances of the failure in the selection of recovery strategies. This allows adaptation of service and of the system to the environmental conditions.

After a short presentation of target application, a mathematical model for generating a composite indicator based on sampled parameters is introduced in this paper. The mechanism for monitoring resources at the node level is described and it is presented how this can be used in the selection of a recovery action (e.g. allocation of processes on/to overloaded nodes should be avoided). Adaptive fault management techniques can be deployed based on the output of monitoring mechanisms. The appropriate recovery strategy is selected at run-time and uses a selection mechanism described in this paper. An example is given, as proof of concept, to show how monitoring mechanisms can be used in fault management.

The fault-tolerant architecture integrating the resource monitoring mechanism achieves dynamic reconfiguration of the recovery strategies based on the changes in the environment. Also, the resource monitoring mechanism increases the differentiation between node crash and network problems for failure suspected nodes. Another advantage of using this mechanism is the dynamic adaptation of resource allocation for an overall increase in application availability [2].

2 DEPAUDE AND ADAPTATION IN DYNAMIC ENVIRONMENT

Deregulation of the electricity market in Europe, the increased number of players on the market and the distributed geographical nature of the electrical power infrastructure are determining factors in the increasing complexity of the underlying distributed automation systems. Hence centralized control of those distributed automation systems is not an option. As such, the need of distributed control raises need for dependable and survivable communication support between all those distributed components [3].

2.1 DepAuDE Middleware Architecture

The DepAuDE (Dependability for embedded Automation systems in Dynamic Environment with intra-site and inter-site distribution aspects) middleware architecture integrates fault tolerance support into distributed embedded automation applications. One of the design purposes of the DepAuDE middleware architecture is to assure dependable functionality in spite of changes in the environment [4]. One target application domain for DepAuDE is the control of Primary Substation Automation System, in the electric power infrastructure.

The target distributed system is composed of a collection of sites that are geographically distributed. Each site is composed of nodes (hosts). The sites are interconnected by an *intra-site dedicated communication network* (see Figure 1). The sites are interconnected via gateways by an *inter-site communication network*, which can be through e.g. Internet or more costly via dedicated channels (e.g. dial-up, satellite, others). The target application uses heterogeneous machines running different operating systems, real-time or non real-time systems such as Windows NT/2000/XP, GNU/Linux and real-time operating system QNX Neutrino [5].

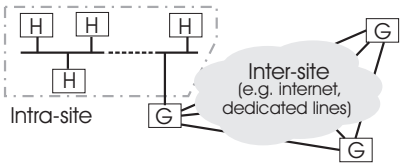


Fig. 1. Intra-site and inter-site communication levels (G=gateway, H=host)

A middleware level architecture was chosen as the most suitable solution for integration of those heterogeneous distributed components. All the heterogeneous systems can be integrated at middleware level to provide the application with required services.

2.2 Communication Scenario

Between the nodes of the target distributed system, there are two types of traffic (Figure 1):

a) between nodes of the same site, *intra-site* communication and

b) between nodes part of different sites, hence *inter-site* communication.

In case a), for intra-site communication, relying on a private network, we assume that communication is reliable (i.e. using reliable protocols) and that the response time is predictable.

In case of inter-site traffic the gateway decides how to handle the communication (e.g. based on messages priority, or based on request for confidentiality encrypted or not). As this inter-site traffic goes through an outside environment with varying characteristics (e.g. variable delays), the adaptation mechanism has to cope with these varying characteristics in order to maintain the dependability of the overall system. Furthermore, the gateways can be made redundant in order to deal with gateway failures. Similarly, real-time operating systems can be used to handle intra-site real-time requirements and intra-site node failures can be handled by the fault tolerance middleware. This makes it a fair assumption that most failures will come from the inter-site communication system (on which we have no control), because we can limit the amount of intra-site failures arbitrarily by adding the necessary redundancy as intra-site network is under our full control.

The communication between the sites of the distributed system is divided from priority point-of-view in different levels of Quality-of-Service (QoS) [6]. The most critical traffic is denoted *QoS 1* and is dedicated to commands, control, alarms and recovery messages between systems. The less critical levels are for inter-process traffic, while the lowest priority is dedicated to management and configuration traffic. QoS 1 traffic requires low delays and high probability of correct transmission. The amount of inter-site traffic is relatively low with respect to intra-site communication on one hand and with Internet load on other.

We define three options for inter-site communication. Each option has advantages and disadvantages:

Internet communication: due to the low cost, this is the most appropriate option in context of which there is sufficient bandwidth and low delays; used for all levels of QoS traffic; in case of reduction of bandwidth or higher delays, the lower levels of traffic have lower priority for delivery;

Multipath routing through internet: multipath routing relies on Redundant Source-Routing (RSR). RSR consists in sending the same information to the destination using different and possibly non-overlapping paths. The usage of more than one path can improve the characteristics of the transmission; delivery probability increases and delivery delay decreases. The main disadvantage of RSR is given by the increase in traffic, more bandwidth being used for the same

information. So, multipath routing through the Internet should be used only for QoS 1 traffic in case of delays and absence of response for QoS 1 traffic; the lower priority levels of traffic are not using this method due to the disadvantages of redundant source-routing [7];

Dedicated connection: due to high cost, the dedicated connection is used only in case of insufficient bandwidth or unacceptable delays for QoS 1 traffic for a time interval Δt.

2.3 Recovery Scenario

2.3.1 Separation of Recovery Code

The target architecture presented in Section 2.1 uses a recovery language to express and manage fault tolerance provisions as presented in [8]. In [8], in the recovery language approach, two distinct parts are available to the programmer, namely the part dedicated for the application using a service language, i.e. the programming language addressing the functional design concerns, and the second part – a special-purpose linguistic structure (called recovery language) for the expression of error recovery and reconfiguration tasks. The recovery language supplies input for the application service language when an error is detected from an underlying error detection layer, or when some erroneous condition is signaled by the application processes. The two parts can be seen as two flows and they are executed concurrently, co-operating with each other. This separation both at design and at run-time brings down the complexity of designing and maintaining distributed dependable systems. It also allows modifying the error *recovery code* while the system is being executed, which may be used while developing and testing the target application [8].

Error recovery and reconfiguration are specified as a set of guarded actions, i.e., actions that are executed when a pre-condition is fulfilled. These actions are called, in this paper, *recovery actions*. If more than one recovery action is used for error recovery and reconfiguration tasks and those sets of recovery actions are switched (e.g. based on environmental conditions) then an adaptive recovery mechanism is supplied, and we call each of those sets *recovery set*. In this work, each recovery set (as mentioned above) consists of recovery actions and is described by such a code for error recovery and reconfiguration.

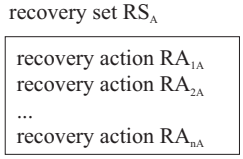


Fig. 2. Recovery sets consist of recovery actions

In Figure 2 there is a representation of recovery set, recovery action relationship. Recovery actions may consist of conditioned actions, e.g. actions are executed only if certain error/available resources conditions are met.

An example follows describing a 3-and-a-spare system, which is a triple modular redundancy system with a backup. If one of the three components fails, it is replaced by the backup. The 'group1' contains all 3 triple modular redundancy components while 'task4' represents the backup. ARIEL recovery language is used [8] to describe the recovery, and, on the right side, the lines are explained. This is an example of a recovery action within a recovery set.

```
IF [FAULTY group1] THEN   // If error detected in group1
      STOP task@1         // stop faulty component
      WARN task~1         // warn others
      START task4         // start the spare component
FI                        // end if
```

2.3.2 Recovery Strategies Adaptation

Different adaptation scenarios can be designed to handle the changes in the environment, which have impact on communication or on the performance of the system (e.g. an overloaded system may generate delays). Those scenarios are described, here in this paper, by e.g. *recovery set A (RS_A)* and *recovery set B (RS_B).*

Those recovery sets are switched at run-time [9] based on the environmental conditions [10], see *loop 1* in Figure 3. The actions within the recovery strategies are executed in case of error detection (e.g. timeout of the watchdog) or event notification (e.g. change of conditions), see *loop 2* in Figure 3.). The recovery actions are selected based on the context of the detected errors and on the available resources of the nodes in case of tasks/processes allocation [10].

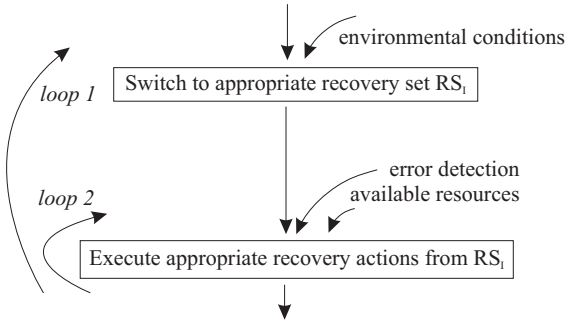


Fig. 3. Two adaptation levels

In this way we have two separate levels of adaptation:

- selection of the recovery action from within a recovery set based on the type of the error detected and the available resources (see Section 4) and

- switching between available recovery sets based on environmental conditions.

The number of recovery sets can be extended with recovery set C, D etc. according to the specific requirements or conditions of the distributed application. For instance, an additional recovery set for rejuvenation [11] and maintenance (e.g. software or hardware upgrades) can be added. In this case, for instance, the exhausting resources (e.g. memory leakage, due to aging) will trigger the switch to this recovery set (executing software rejuvenation [11]). Because rejuvenation actions include reboot of a node/task, maintenance actions can be coupled to optimize activity. However, only the actions supported by underlying Basic Services API (see Section 2.4.1) can be used in the adaptation mechanisms.

Based on communication scenario from Section 2.2, *recovery strategies* can be defined. Those recovery strategies consist of recovery sets dedicated on one side for intra-site level operation and on the other side for inter-site level communication. Because inter-site level communication can influence the adaptation scenario at intra-site level (e.g. if the site is isolated, a service degradation should occur) recovery strategies encapsulate adaptation at both inter-site and intra-site levels.

We are using two major adaptation strategies described by two recovery strategies: *normal recovery strategy* and the second, *"Plan B" recovery strategy*, dedicated for critical situations. Both recovery strategies consist of recovery sets for intra-site level or inter-site level adaptation. For instance, for the inter-site communication, the *normal recovery strategy* is a low cost one, for which the communication via internet is sufficient. On the other side *"Plan B" recovery strategy* includes at inter-site level a recovery set describing appropriate actions for switching to a dedicated line in case that Internet connection is lost or overloaded (e.g. due to denial-of-service attack).

This switch of recovery sets and strategies helps from two perspectives. First, this allows a hierarchical view of recovery, improving recovery scalability in distributed systems (e.g. a more restrictive recovery strategy allows only switching between recovery sets reflecting restricted conditions). Second, the switch of recovery sets (and strategies) allows faster recovery (i.e. some of the conditions are already verified to switch to appropriate recovery set/strategy, so less time spent in verification of conditions inside recovery sets).

In the following sections it is shown how those recovery adaptation means fit in DepAuDE architecture.

2.4 Environmental Change Detector (ECD)

2.4.1 Using ECD in DepAuDE

In DepAuDE, at middleware level (Figure 4), a module called *Environmental Change Detector (ECD)* is implemented to gather sampled data supplied by node or network monitoring mechanisms. The output of the ECD goes to the *Recovery Interpreter*

(RINT), which interprets the recovery code [8]. As such, it can be used for selection of recovery actions, switching of recovery sets or recovery strategies.

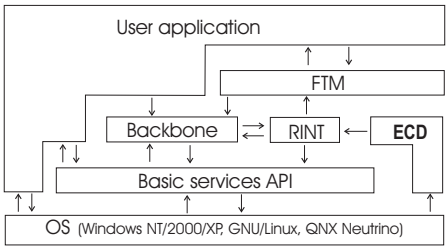


Fig. 4. ECD in DeAuDE architecture

The *Basic Services API*, called also Basic Service Library (BSL) in DepAuDE has as main objective OS-independence and transparent communication. For this purpose the BSL abstracts the OS and offers the required OS related functionality to all other system components. The OS-independence comes at a cost. This cost is the requirement that for each supported OS an implementation of the BSL for this specific OS is necessary.

One of the industrial applications of BSL in DepAuDE is the Primary Substation Automation System (PSAS) which consists of embedded hardware and software located in a substation of electricity distribution [7]. As PSAS has a certain number nodes (industrial systems requiring real-time operating systems and PCs), the task allocation is stored in a table which is used by the BSL. The functionality of *Basic Service API* consists of startup and shutdown, node control, process and thread control, communication, time handling, semaphores. Concerning tasks, BSL supports the following actions: create, restart, isolate, stop task. Each application task has at least one standby replica on different node (as such reconfiguration does not involve moving large amount of data; the status/location of tasks is reflected on the allocation table used by BSL).

FTM (Fault Tolerance Mechanisms) are the set of single-version software fault tolerance provisions [12] that are adopted within DepAuDE and integrated into its intra-site architecture. From the point of view of the programmer making use of these mechanisms, FTM are a set of ready-to-call software components such as watchdog timers, exception handling tools, etc. From a developing point of view, FTM are sets of functions that make use of the Basic Service API to communicate, launch tasks, and synchronize and communicate their error detection to the architectural component that collects and organizes them into a database (in the DepAuDE backbone).

The *Backbone* is a distributed application collecting system and application status information and coordinating error recovery and reconfiguration. The objectives of the backbone are: collecting and maintaining event notifications produced by the FTM, the Basic Service API, and the application; deducing properties of faults producing the event notifications; managing error recovery by executing the run-time

executive of the selected recovery strategy. One of the tasks of the backbone is that of managing the database of event notifications produced by the FTM, the Basic Service API and the application tasks. Exploiting functionality provided by monitoring, error detection and the control of the communication flow, another task of FTM is to detect and recover from communication faults (e.g. using multipath routing through Internet).

RINT gets the current recovery set, composing queries to the database (from Backbone) and executing error recovery actions depending on the result of the queries (see Figure 5). RINT is similar with a stack-based instruction set machine: results are stored and read from a run-time stack. Current environmental conditions are periodically receiving by RINT from ECD to select the most suited error recovery strategy based on the hints of a failure detector, and also on the changes in the environment.

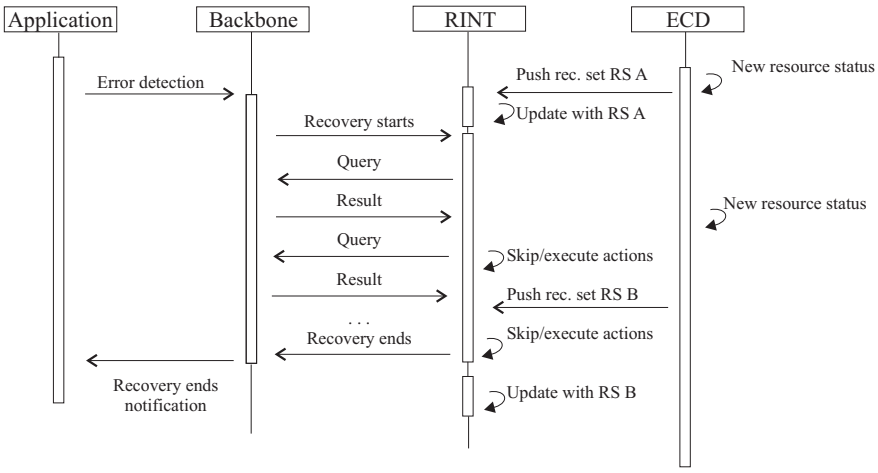


Fig. 5. Switching recovery sets A, B (RS A, RS B). Timing view

ECD sends notification to RINT if certain defined thresholds (as value and/or as time interval) are reached and/or if monitored data is back in the normal range. Those notifications from ECD are used by RINT for recovery adaptation. For instance, in Figure 6, based on the notification received by RINT from ECD, the recovery set is switched (updated) with the current recovery set when RINT is free. In this simplified example, the recovery set contains the recovery code associated to the environmental conditions detected by monitors.

2.4.2 Proposed ECD Functionality

A functional representation of Environmental Change Detector is presented in Figure 7. ECD can incorporate *Node monitors* (sampling for instance CPU usage,

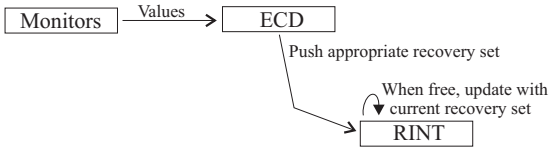


Fig. 6. Integration of monitored data into recovery set (simplified model)

available memory etc.), *Network monitors* (i.e. delivering bandwidth availability data) and *Other monitors* (e.g. for EMI).

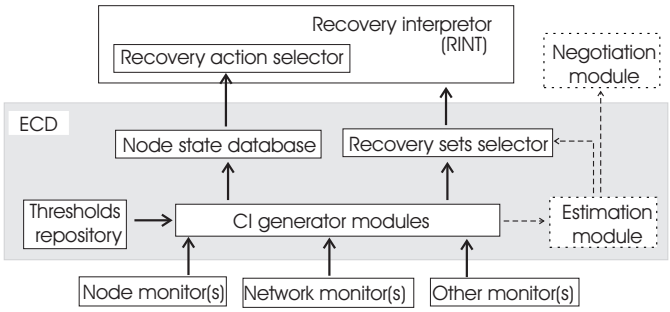


Fig. 7. General representation of ECD and the adaptation layer

The ECD contains a *Thresholds Repository* as reference database for comparing the sampled data (data supplied by the monitors) with the threshold values. The repository allows easy changes/adjustment of the thresholds. Based on the Threshold Repository values, the input values are compared, and, using a mathematical model, a *Composite Indicator (CI)* is generated in *CI generator module*. The mathematical model for generation of CI is described in detail in Section 3.

For the node state determination, the CI is stored in *Node state database*, together with a *duration counter* used for describing resource state of the node. More details are presented in Section 3. The state of the nodes can be used by *Recovery action selector* module to select/execute given recovery actions at run-time.

Other modules, *Estimation module* and *Negotiation module*, could be further developed in order to estimate future behavior and to negotiate QoS policies.

3 DATA ANALYSIS. INTEGRATING RESOURCE MONITORING INFORMATION

As already mentioned, we consider that the load of a node can influence the performance and the dependability of a system. In this context, there is a need for a quantification of the factors/parameters sampled at node (or network) level. Examples of such factors/parameters are available memory, CPU usage, etc. for node level or available bandwidth etc. for network level.

The *composite indicator (CI)*, proposed in this section, encapsulates the sampled value and also the relevance of each parameter, based on a weight matrix. How this CI is used for the switch of recovery sets, at intra-site level, is presented in Section 4.

3.1 Mathematical Model for Generation of the Composite Indicator

In this subsection, the general mathematical model for computing the composite indicator of a component (e.g. node) is given. Different factors can have a higher or a lower influence/impact on the evaluation of a system performance. In order to capture the importance of each factor, the best option would be to use a weight associated for each parameter. A general formula to compute a general CI influenced by n factors $f_1, \ldots, f_n$ with their associated weights $w_1, \ldots, w_n$ is given in Equation (1).

$$\mathrm{CI} = w_1 \times f_1 + w_2 \times f_2 + \ldots + w_n \times f_n \tag{1}$$

As different factors have different ranges, we consider $t + 1$ sub-domains for each factor to simplify the representation, so that t thresholds separate those sub-domains.

As an example in this description, we consider *t=2* thresholds for node CI called *alert threshold* and *alarm threshold* and the names for domains separated by those thresholds are: *normal*, *alert* and *alarm*. From now on, the term 'above' will denote a situation when the sampled value reached a certain threshold to a more critical sub-domain, and the term 'below' will denote a situation when the sampled value did not reach the threshold. Also, we use 'first' threshold as the first comparing value above the normal state of a parameter, all other thresholds being above this one.

The thresholds can be adjusted independently of the sampled parameters composing the CI. The thresholds are chosen based on statistical analysis [13].

The number of sampled parameters integrated in computing the CI is denoted by c. In context of multiple counters and thresholds, the Equation (1) can be rewritten as a sum of matrix multiplications in Equation (2) with W as a column matrix of c vectors containing the weights for each sampled parameter.

$$\mathrm{CI} = W_1 \times F_1 + W_2 \times F_2 + \ldots + W_c \times F_c \tag{2}$$

On the first positions of the equation are the most important parameters (that have a higher impact on the behavior of the system) and on the last positions are the least important ones.

$$W = \begin{pmatrix} W_1 \\ W_2 \\ \vdots \\ W_c \end{pmatrix} \tag{3}$$

$$F_i = \begin{pmatrix} f_{it} \\ \vdots \\ f_{i2} \\ f_{i1} \end{pmatrix}, \text{ where } f_{ij} = \begin{cases} 0 & \text{if sample value of counter } i \text{ is } < th_{ij}, \\ 1 & \text{if sample value of counter } i \text{ is } \geq th_{ij}. \end{cases} \quad (4)$$

In Equation (3), vector matrixes W_i are components of the weight matrix W. The t cells in the column matrix F_i, can have only one of the two values 0 or 1 as in Equation (4).

Inside the weight matrix W, the weights are powers of 2. We have chosen those weights for practical reasons. First, in this way the computation of CI can be reversed and based on the positive integer value of CI (CI will always be a positive integer) can be identified which parameter was determining a certain composite threshold passed. Also, using this model for computing CI, we can use (e.g. in case of C programming) the *integer unsigned type* and save values for later statistical analysis. This will use 32 bits for a CI which in this case can be composed for instance of different combinations of number of thresholds and counters, with the restriction $t \times c \leq 32$. (e.g. 16 monitored parameters with 2 thresholds, meaning 3 intervals of values normal, alert and alarm).

The weight matrix is given in Equation (5).

$$W = \begin{pmatrix} 2^{t\times c-1} & 2^{(t-1)\times c-1} & \dots & 2^{2\times c-1} & 2^{c-1} \\ 2^{t\times c-2} & 2^{(t-1)\times c-2} & \dots & 2^{2\times c-2} & 2^{c-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 2^{t\times c+1} & 2^{(t-2)\times c+1} & \dots & 2^{c+1} & 2^{1} \\ 2^{(t-1)\times c} & 2^{(t-2)\times c} & \dots & 2^{c} & 2^{0} \end{pmatrix} \quad (5)$$

Based on this model the composite indicator can take values from 0 to $2^{t\times c} - 1$. CI has the value 0 when none of the parameters is above any threshold, meaning all parameters are in normal range. If any of the parameters goes above first threshold then the CI has a value greater than 0. CI takes value $2^{t\times c} - 1$ when all the sampled parameters are in the worst case scenario, above all thresholds, so CI = $2^0 + 2^1 + \ldots + 2^{t\times c-1}$.

3.2 CI Definition and Thresholds

The values of CI can be interpreted in different ways. An option would be to establish general thresholds for CI. For instance, the simplest option would be to use the following rule: "If **any** parameter i ($1 \leq i \leq c$) is above its threshold j (th_{ij}) then the CI is above the general th_j threshold (with $1 \leq j \leq t$)."

This is the rule used in this paper. In this case, CI will have t thresholds and $t + 1$ sub-domains of values as in Equation (6).

$$\mathrm{CI} = \begin{cases} 0, & \text{if no parameter is above the first threshold } th_1; \\ 1 \ldots 2^c - 1, & \text{when } \exists \text{ parameter above the threshold } th_1 \text{ and none} \\ & \text{is above } th_2; \\ \vdots & \\ 2^{(t-1)\times c} \ldots 2^{t\times c} - 1, & \text{when } \exists \text{ parameter above the threshold } th_t. \end{cases} \tag{6}$$

Given the rule introduced above, for a simple case with CPU usage and available memory as parameters, based on thresholds from references [13], the representation of the normal/alert/alarm zones and CI alert/alarm thresholds are distributed as in Figure 8.

As mentioned in [13], the values of CPU usage and available memory thresholds are determined empirically, i.e. from observations. Further mathematical analysis work could be carried out to detect the most appropriate thresholds given the application, system architecture and the OS installation combination (i.e. those threshold values would not apply to embedded systems where total memory is smaller than defined thresholds).

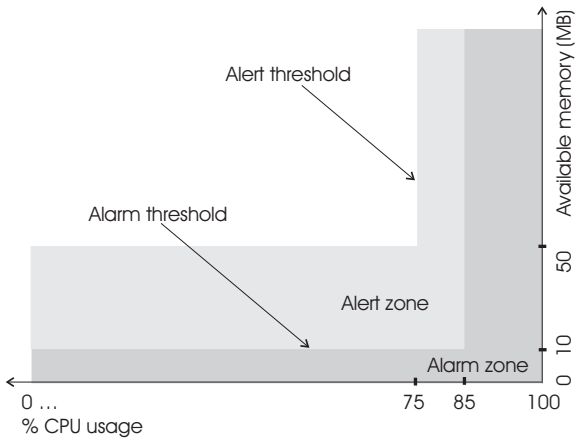


Fig. 8. 2D representation of available memory vs. CPU usage, and the thresholds

4 RECOVERY ADAPTATION USING CI

4.1 Recovery Action Selection Based on Node Resource Monitoring

As already mentioned, different actions are described inside recovery strategies to be taken in case of error detection. If the actions require restarting/starting of processes/tasks on different nodes then the decision should be taken also based on the available resources of the future host.

In our resource monitoring mechanism for the node we considered counters such as available memory, CPU usage, and numbers of reboots. In Figure 9 there is a schematic representation for determining the CI for the node resources based on three sampled counters ($c = 3$) and using two thresholds ($t = 2$).

Each node is sending messages to the Environmental Change Detector with information on *CPU usage, Available memory* and *Up time*. From the value of the *Up time* the number of reboots for a given interval of time is computed.

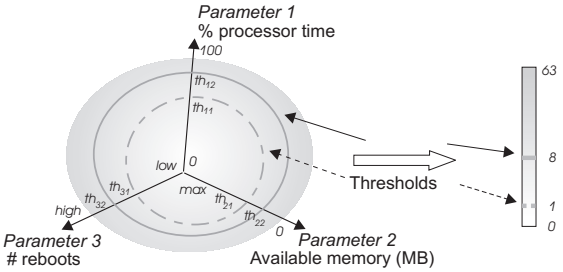


Fig. 9. CI generation based on sampled values for three counters and two thresholds

The CI generator (Figure 9) receives the data structure and compares the values with the corresponding ones stored in the *Thresholds repository*. As there are $t = 2$ thresholds and $c = 3$ counters, according to the rule presented in the previous section, the weight matrix has 3 lines and 2 columns. In this case, Equation (2) (based on Equations (3), (4), (5)) can be written as:

$$\begin{aligned} \text{CI} &= \sum_{i=0..2} W_i \times F_i \\ \text{CI} &= (\,2^5 \quad 2^2\,) \begin{pmatrix} f_{01} \\ f_{00} \end{pmatrix} + (\,2^4 \quad 2^1\,) \begin{pmatrix} f_{11} \\ f_{10} \end{pmatrix} + (\,2^3 \quad 2^0\,) \begin{pmatrix} f_{21} \\ f_{20} \end{pmatrix}, \end{aligned}$$

where $f_{ij} = \begin{cases} 0 & \text{if sampled values of counter } i \text{ is } < th_{ij}, \\ 1 & \text{if sampled values of counter } i \text{ is } \geq th_{ij} \end{cases}$ and th_{ij} is the threshold j of the counter i.

For this case, for CI we have 3 intervals of values (according to Equation (6)) as presented below:

- CI = 0, when none of the counters reached the alert threshold, and we consider this as normal state;
- CI $>$ 0 and CI $\leq$ 7 ($\leq 2^c - 1$), when at least one of the sampled value of counters is above the alert threshold, but below the alarm threshold, and we consider that the composite indicator is also above the general alert threshold;
- CI $>$ 7 ($> 2^c - 1$), when at least one of the sampled values of counters is above the alarm threshold then the composite indicator is considered to be above the

general alarm threshold. (The maximum value for CI in this case is $CI = 63$ $(= 2^{t \times c} - 1)$.)

Based on the values of the CI and on the period for which the CI is above a certain threshold we can define different states for the node:

- a *normal state*, when CI is below any threshold, and in this case, there are available resources at node level for new tasks/processes to be restarted/migrated;
- an *alarm state* when CI is above alarm threshold for a Δt interval of time considered to be critical, and in this case, restart/migration on the node of new tasks/processes should be avoided, and
- one or more *transient alert states* when CI is above the alarm threshold but the interval is still shorter than Δt.

The presence of the *transient alert* state(s) is required for the stability of the system, the Δt interval of time should be sufficient in order to avoid transient switching back and forth in case of fluctuations in the load of the system. We use a counter, called *Alarm duration counter (ADC)*, to compare its value with the Δt interval of time, and in case it is greater to trigger the switch to alarm state. More detailed explanation about ADC is presented in Section 4.3.

4.2 Local Monitoring. Implementation Considerations

The software monitoring mechanism is hardware and operating system dependent. Due to the heterogeneous nature of the distributed system of the application, dedicated modules had to be implemented for each operating system. Even if there are tools available for monitoring (see Section 5.1 regarding related work) they have two major inconveniences, they are not lightweight enough and they are not easy to integrate in the target application. Because of this, a module was developed for capturing the available resources for operating systems such as Windows NT/2000/XP, GNU/Linux and real-time operating system QNX Neutrino [5].

Some counters are easily accessible for Windows using PerfMon [14], a Performance monitor utility, while on other operating systems (GNU/Linux, QNX Neutrino) they are not easy accessible. As such, we limited the number of counters to *CPU usage*, *available memory* and *up time* for number of reboots.

The data structure supplied to ECD for this implementation running on all mentioned operating systems is composed of CPU usage, the available memory and the up time. From the value of the Up time the number of reboots is computed. The sampled parameters, CPU usage and Available memory values are scaled, for the interval 0..100. The Up time is given in seconds. The structure used to send those data contains also a flag (inconsistency flag).

The inconsistency flag is set to 0 if there were no error messages and the output values are in the normal range. In this case, ECD can make decision based on the data sent in the structure.

The flag is set to 1 in the other cases: in case of a detected error, or if functions give inconsistent values. For instance, in absence of a pre-defined function returning CPU usage %, this can be computed from values of *up time* and *idle time* from hardware counters, at given time interval (e.g. every second). However, if those values are not expressed on the same units or the sampling frequency is not accurate, inconsistent values (e.g. negative cpu usage) can be obtained. In case of Linux, kernel versions before 2.4, running on a system with two microprocessors, perhaps due to non-atomic reads and updates, the idle time can go backwards. The inconsistent values are ignored and not shown for instance in case of top command. Also, in case of the QNX Neutrino RTOS, unexpected values can appear due to the *timer quantization error* [15]. This is explained by the 153 nanoseconds discrepancy at each millisecond between the request and the hardware response, so, the sampling frequency is not accurate [15]. We have to deal with those situations, and the flag will be set on 1 if the values are outside the definition domains. In case of the flag set on 1, the Environmental Change Detector ignores the values received. ECD will make no decision upon those values.

The implementation has a low performance impact (below 1 % CPU usage), uses C as programming language and the length of the code varies from about 130 lines in case of Win NT/2000/XP (where dedicated functions are provided), to almost double in case of Linux. Additional overhead may be required for reliable communication. This depends on the underlying system particularities, mainly on the robustness of the underlying BSL and operating system (i.e. if reliable communication is provided, there will be no additional overhead, otherwise provisions for reliable communication need to be incorporated in the middleware).

As we already mentioned, the threshold values are taken from references, and we use the same values independent of the operating system. Performance comparison between Windows NT, Linux and QNX as the basis for cluster systems is presented in [16]. One of the conclusions of the material is that there is *no clear-cut advantage* from one operating system over the other. From our perspective this *justifies* the use of same thresholds values for same configuration hardware but running different OS.

4.3 Example of CI Generation and Switch of Recovery Sets

In this subsection an example is given for better understanding of different concepts introduced in this paper (e.g. CI, ADC etc.).

According with the assumptions and definitions presented already, we use 2 types of recovery sets: one for the case of node in normal state, from resource point of view, and one for the case of node in alarm state, from resource point of view.

At node level, few but relevant data should be collected for the performance evaluation and characterization of resources. Some of the important categories to monitor are CPU, memory, disk I/O and network. It should be noticed that the resources are all interdependent. A bottleneck in one will affect the others. For example, if there is not enough memory then CPU usage will increase, because the

CPU spends more time on page swapping. This helps us determine the order/weight in the weight matrix of the sampled parameters, i.e. the ones that have higher influence on others will have a higher weight associated.

In this example, *CPU usage, available memory* and *up time* are sampled. The thresholds from reference [13] are used for CPU usage, alarm threshold at 85 % and alert threshold at 75 %. For available memory the alarm threshold is at 10 MB and alert threshold at 50 MB. In this context, with three parameters and two thresholds, the CI takes values from 0 to 63 ($= 2^{t \times c} - 1$).

Figures 10 and 11 from this section contain values sampled during ten hours of testing on a system running computation intensive applications (on a double microprocessor Linux 2.4 machine). Two representative intervals were selected, one for fluctuating conditions, but not triggering change in recovery sets, in Figure 10 to show why stability feature (in our case ADC) is required, and in Figure 11 a representative situation when recovery sets are changed.

The stability of the system is important. To avoid unnecessary oscillations of the system (frequent switches of recovery strategies) we use the ADC counter. The process of incrementing the ADC starts when the composite indicator 'crosses' the alarm threshold (e.g. moment A in Figure 10). If the ADC counter is above the limit ΔT and the composite indicator is above the alert threshold then the state changes to alarm state (e.g. moment C in Figure 11).

In our testing environment, the sample values for available memory and the numbers of reboots were below the alert thresholds for both intervals, therefore their graphs are omitted. In this context, from the way we defined the composite indicator, CI can take only 3 values. CI = 0 when CPU usage is below the alert threshold, CI=2 (2^1) when is above the alert threshold but below the alarm threshold and CI = 18 ($2^4 + 2^1$) when the CPU is above the alarm threshold.

$$\mathrm{CI} = \begin{pmatrix} 2^5 & 2^2 \end{pmatrix} \begin{pmatrix} f_{01} \\ f_{00} \end{pmatrix} + \begin{pmatrix} 2^4 & 2^1 \end{pmatrix} \begin{pmatrix} f_{11} \\ f_{10} \end{pmatrix} + \begin{pmatrix} 2^3 & 2^0 \end{pmatrix} \begin{pmatrix} f_{21} \\ f_{20} \end{pmatrix},$$

where f_{00}, f_{01}, f_{21}, $f_{22} = 0$.

The same equation could be represented as in Table 1. Also here, the parameters have different impact in the composition of CI due to their importance (weight) and value. The importance of a certain parameter in CI is reflected in the table by its position between the columns. In this case, we assign available memory a higher importance because it does not change rapidly (if there is little available memory this can impact also the CPU usage), CPU usage is on the second position and the number of reboots on the last position.

Thresholds	Alarm threshold passed			Alert threshold passed		
Parameters	memory	CPU	reboot	memory	CPU	reboot
Weight	2^5	2^4	2^3	2^2	2^1	2^0
Value	0	1	0	0	1	0

Table 1. An example of composite indicator computation

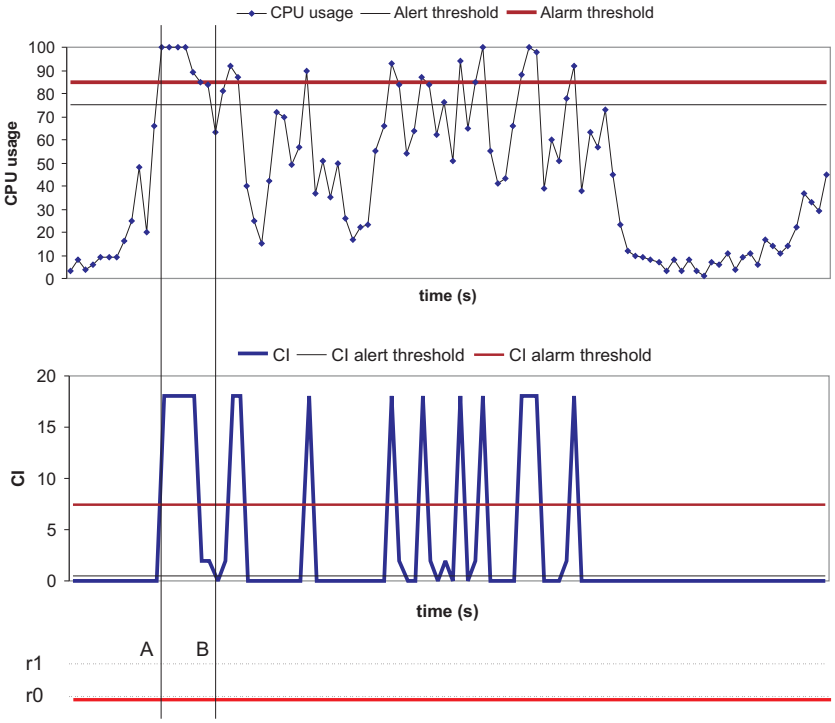


Fig. 10. Case a. Alarm duration counter is not triggering alarm recovery set: any interval AB is smaller than ΔT

Also in Table 1 it can be noticed that the columns are repeated as many thresholds we have. In this way, the group of columns on the left side reflects the higher values of the sampled parameters (a value of '1' reflects the fact that the values of the parameters are above the higher threshold). After sampling, in case the sampled value is below any threshold, '0' is filled in both positions of the counters (e.g. Reboot and Memory columns in the table). If the alert threshold is passed, '0' is filled in Alarm column of the counter and '1' in the Alert column. In case that the sampled value is above the Alarm threshold then in both positions of the counter '1' is filled (e.g. CPU columns) in this case.

We consider the values from the table in binary representation, MSB on the left (corresponding to weight 2^5) and LSB on the right side (corresponding to weight 2^0). Converting from binary to decimal we obtain the CI. In the case represented in Table 1 the CI takes the value of 18. Based on this model, it can be noticed that the integer value of CI can be easily converted to binary and determine which parameter was above which threshold.

Figure 10 presents a case with CPU usage passing the thresholds for short intervals of time: alarm threshold is reached at the moment denoted with A, determining initialization and incrementing of the ADC counter. The situations presented in Figure 10 will trigger no recovery/adaptation actions, because those intervals are shorter then ΔT (ADC $< \Delta T$) where ΔT is a reasonable interval of time (e.g. minutes). The ADC counter is reset at moments B, when the CI gets below the alert threshold. It can be noticed in Figure 11 that this situation repeats (CI reaches the value of 18 again, but for few seconds only). So, based on this, recovery code is not changed, ($r0$ being further used).

Figure 11 presents the case in which the alarm recovery strategy is triggered due to a longer low resource circumstance. AB interval is larger than ΔT. At moment C there is a change in recovery sets from normal to alarm and at the moment B there is a change from alarm recovery sets to normal one.

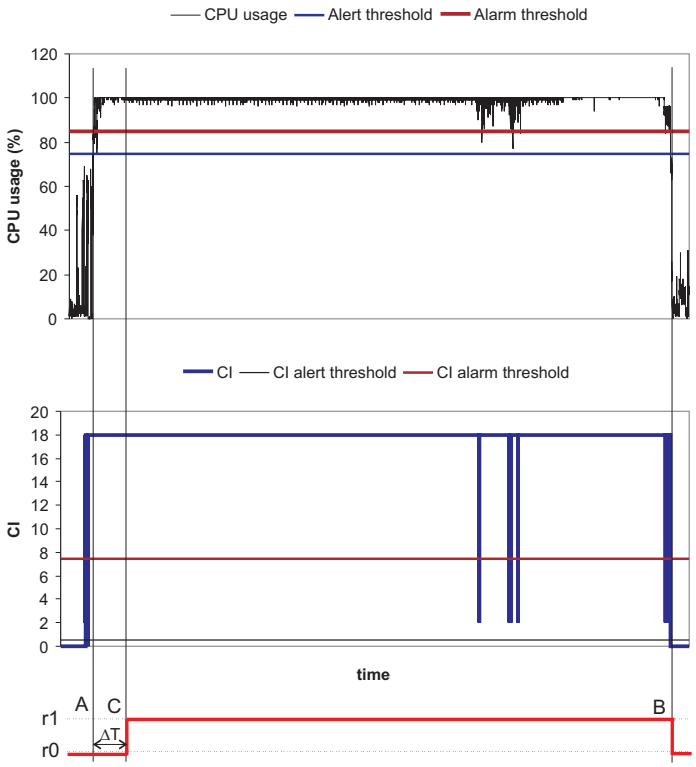


Fig. 11. Case b. Alarm duration counter triggers another recovery set (r1) at time C = A + ΔT. When the CI goes below the alert threshold, at moment B, the recovery set used will switch to r0.

Figure 12 presents the state diagram for two thresholds (alert and alarm) and with 3 parameters monitored. So CI has three domains of values: $0, 1, \ldots, 7$ and $8, \ldots, 63$.

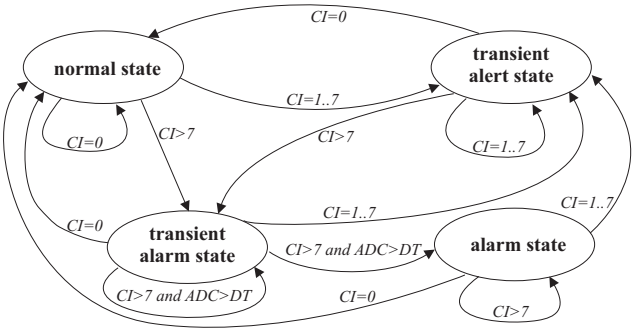


Fig. 12. State diagram. CI = composite indicator, ADC = alarm duration counter, DT = ΔT.

In Table 2 there is the transition table of the finite state machine. In the table the output and the operations/actions are indicated based on the current state of the node and the inputs (CI, ADC). Inside the cell, the first information is the next state, the second is the recovery set to which there is a switch and the third is the operation done in the context given by the current state and the inputs: CI and ADC.

Current state/Input	Alarm duration counter	Normal state (NS)	Transient alert state (TATS)	Transient alarm state (TAMS)	Alarm state (AS)
CI = 0	ADC $\leq$ DT	NS/-/-	NS/-/ADC $\leftarrow$ 0	NS/-/ADC $\leftarrow$ 0	*
	ADC > DT	*	NS/rs0/ADC $\leftarrow$ 0	NS/rs0/ADC $\leftarrow$ 0	NS/rs0/ADC $\leftarrow$ 0
CI = 1..7	ADC $\leq$ DT	TATS/-/-	TATS/-/-	TATS/-/ADC++	*
	ADC > DT	*	TATS/-/ADC++	TATS/-/ADC++	TATS/-/ADC++
CI > 7	ADC $\leq$ DT	TAMS/-/-	TAMS/-/ADC++	TAMS/-/ADC++	*
	ADC > DT	*	TAMS/-/ADC++	AS/rs1/ADC++	AS/-/ADC++

Table 2. Transition table (next state/output/operation)

In Table 2, "*" means impossible case (for instance it is impossible to be in the alarm state and the ADC counter to be smaller than ΔT, denoted with DT), "_" that there is no operation or no output. ADC $\leftarrow$ 0 means that alarm duration counter takes the value of zero and ADC++ means that ADC is incremented. "rs1" means selection of recovery sets with recovery actions dedicated to the alarm state of the given node and "rs0" means selection of recovery sets with actions for normal state.

We used two thresholds in this model: alert and alarm threshold. Even if this allows the use of three domains, we use only two for recovery. The alert threshold is used to avoid oscillations in recovery decisions, together with ADC. The switch of the state from normal to alarm is triggered by ADC reaching a certain value, and

the switch from alarm state to normal state is triggered (together with setting ADC on zero) by CI reaching a value below alert threshold.

The composite indicator, as defined in Section 3 and used in Section 4, integrates three parameters (memory usage, CPU usage and number of reboots) together with two thresholds for each parameter, all in one integer. The same information can be also represented as a tuple of three values, one for each parameter. However, our integrated CI has some advantages compared with a tuple. First, it is easy to compare an integer with an integer threshold (for comparison of tuples, each value is compared separately). Then, CI is less demanding from memory perspective, because it can be stored as an integer. Also, the model presented can be extended easily for more parameters/thresholds (e.g. a 32 bit integer can encapsulate all combinations of t number of thresholds, c number of parameters, as long as $t \times c \leq 32$). The model proposed in this section allows reverse computation. From the integer value describing a node, the domains of values for all parameters can be determined (e.g. a value of 31 can be written $0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$ which means that the first parameter (available memory) is above alert threshold but below alarm threshold and the other two (CPU usage and numbers of reboots) are above alarm threshold).

The value of ADC influences the speed of reaction. A small value for ADC is useful in case of bottlenecks so that they are quickly detected. However, a system can function correctly even if it is using most of its resources for a certain time interval, for example if a system is running computationally intensive applications (e.g. simulations which can last tens of minutes with CPU usage above 85 %). Our mechanisms will consider the system in 'alarm state' even if this is not an indication of a faulty condition. In such a case the system is not going to receive new tasks, so all the resources are going to be used by the computation intensive application. Another special case are applications which show periodicity from resources consumption point of view (such a profile must be considered in order to avoid oscillations). As such, identifying the profile of resource usage for the applications running on the target system is useful before setting the ADC or other thresholds. Nevertheless, for real-time applications the designer should also make sure that under the worst execution time (e.g. due to reconfiguration), the timing constraints are met.

However, the systems addressed in this paper are running distributed applications, with intra- and inter-site aspects, and so appropriate but different mechanisms can exist for intra-site and inter-site levels.

5 RELATED WORK

This section presents material on two topics. First, performance monitoring mechanisms and their relevance for our target operating systems are analyzed in first section and then, in the second section, we identify the research work related on fault tolerance adaptation based monitoring in middleware architectures.

5.1 Monitoring Tools

The Environmental Change Detector receives data from monitors. For the node level monitoring we needed a lightweight software module running on the different OS (Windows NT/2000/XP, GNU/Linux and QNX Neutrino) used in the target application distributed system and easy to integrate in the entire DepAuDE architecture.

For some of the operating systems (e.g. GNU/Linux) a lot of applications are available for capturing the status of the resources in general and per process. They were not useful because they are mostly dedicated to capture all the possible data especially for visualization, so too much information, and not lightweight enough (e.g. 4 % CPU usage just for monitoring and visualization). We are not going to give extensive references for those monitoring tools, but only for the ones relevant in for our application.

In case of Windows NT/2000/XP, the Windows' Perfmon utility is available, as well as other dedicated functions for reading the values from the registry [14]. Those functions can be easily integrated in any application.

In case of QNX Neutrino, although the status of the system is displayed on the desktop on Photon session, we could not access these data for our monitoring module. However, an application called SPIN is available which monitors the performance of the QNX/Neutrino system [17]. This application, as the ones mentioned for GNU/Linux, is not lightweight and cannot be easily integrated in our architecture.

As we mentioned above, all those applications on all the operating systems are increasing the load of the systems. Also it is difficult to integrate them with the rest of our work. In this context the best option was to write a software module using the available functions for each operating system.

Distributed monitoring systems are not suitable for our target application. In our architecture "I'm alive" notifications were sent, and this allowed adding load related information (i.e. node CI) to those notification messages. Given the context in which the mathematical model was developed and ECD integrated, if we want to extend to other applications, other solutions could be applied for gathering monitoring data, and those could use distributed monitoring. Applications which are not using "I'm alive" notification mechanism for availability detection could take advantage of such distributed monitoring tools. Tools such as SNMP (Simple Network Management Protocol) [18] can be used to collect resource information and using a centralized CI generator to use the same model for describing the resources of the different nodes interconnected at site level. SNMP has also disadvantages due to initial setting and access rights.

5.2 Other Adaptation Approaches Based on Monitoring

An adaptive architecture that provides dependable distributed objects is developed in the AQuA project [19]. In AQuA, an object factory is implemented on each host.

The function of each object factory is to kill processes, to start processes, and to measure the host load, in order to provide information to the advisor to be used in deciding which hosts to start objects on [19]. It is shown in [19] that the factory periodically sends the load of its host to the dependability manager which decides how to assign replicas to hosts. However, there is no further information how the load of the host is influencing the decision of the dependability manager. Our work proposes, compared with the above mentioned architecture, a module for capturing different parameters in three different operating systems and a mathematical model for computing a quantitative value characterizing the available resources of the node. Also, a method for adapting recovery actions based on this information is given.

ROAFTS middleware allows configuration management and adaptation decision based on monitoring [20]. ROAFTS considers the availability of the nodes (availability detected based on heartbeat messages) for allocation of resources to active applications. The local resources are not considered. The messages between nodes are sent in two copies, using two paths, to secure reception of the messages at destination in a timely manner (ROAFTS supplies real-time support). Our work adds load information to "I'm alive" messages (similar with heartbeats messages in ROAFTS). Also, messages are sent using multiple paths (redundant source-routing) in our work, not for real-time reasons, but for recovery and adaptation purposes, and to guarantee quality-of-service support.

6 CONCLUSIONS AND FUTURE WORK

This paper presents the integration of an Environmental Change Detector into a fault-tolerant middleware architecture developed in DepAuDE project. The objective of the ECD is to supply an automated selection of recovery strategies based on the run time environment conditions. ECD is proposed as solution for integrating the external, dynamic aspects (e.g. load of nodes and network, and other information such temperature, EMI) in the fault tolerance and adaptation decisions.

A monitoring mechanism was designed and implemented. A monitoring module running on Win NT/2000/XP, GNU/Linux, QNX Neutrino was developed to capture the resources of the nodes of the distributed system. The requirements of the monitoring mechanism at network level were presented and a mechanism for integrating those resource monitoring information in the adaptation of recovery was introduced.

The output of the monitoring mechanism is used into the Environmental Change Detector. A method to convert the metrics supplied by monitors into a composite indicator reflecting the environment conditions is proposed. The advantages of this method are shown.

Based on the values of the composite indicator recovery strategies can be dynamically switched and recovery actions can be selected based on the available resources.

Further work can be carried out. For instance, the threshold values for computing CI were taken from literature. Statistical analysis could be carried out to identify the most appropriate thresholds for given system (hardware configuration) and application. Also, based on the mathematical model presented in Section 3 analysis of more parameters sampled can be considered for a more realistic view (e.g. for industrial applications, external parameters such as temperature, EMI can be considered). Another path can be implementation of network monitoring and adaptation mechanisms using recovery adaptation at network level.

REFERENCES

[1] Hauser, C. H.—Bakken, D. E.—Bose, A.: A Failure to Communicate. IEEE Power & Energy Magazine, Vol. 3, No. 2, March 2005, pp. 47–55.

[2] Tirtea, R.—Deconinck, G.—Belmans, R.: Cost Analysis of Adaptive Fault Management. Proc. Of Annual Reliability & Maintainability Symposium (RAMS) on CD Rom, Alexandria, Virginia, USA, Jan. 24–27, 2005; 7 pages.

[3] Dondossola, G.—Lamquet, O.—Masera, M.: Emerging Standards and Methodological Issues for the Security Analysis of Power System Information Infrastructures. Proc. of 2nd Int. Conf. on Critical Infrastructures (CRIS 2004) on CDROM, Grenoble, France, Oct. 25–27, 2004; 6 pages.

[4] Deconinck, G.—De Florio, V.—Belmans, R.: Dependable Distributed Automation Systems within an Open Communication Infrastructure. Proc. CRIS Int. Conf. on Power Systems and Communication Systems Infrastructure for the Future, Beijing, P. R. China, Sep. 2002; 6 pages.

[5] QNX Software Systems web site. Availaible on: `http://www.qnx.com/`.

[6] DepAuDE project web site. Availaible on: `www.depaude.org`.

[7] Deconinck, G.—de Florio, V.—Belmans, R.—Dondossola, G.—Szanto, J.: Integrating Recovery Strategies Into a Primary Substation Automation System. Proc. Int. Conf. on Dependable Systems and Networks, June 2003, pp. 80–85.

[8] Deconinck, G.—de Florio, V.—Botti, O.: Software-Implemented Fault-Tolerance and Separate Recovery Strategies Enhance Maintainability. IEEE Trans. on Reliability, Vol. 51, June 2002, pp. 158–165.

[9] Deconinck, G.—de Florio, V.—Botti, O.: Separating Recovery Strategies from Application Functionality: Experiences with a Framework Approach. Proc. Ann. Reliability & Maintainability Symp., Philadelphia, PA, USA, Jan. 2001, pp. 246–251.

[10] Tirtea, R.—Deconinck, G.—de Florio, V.—Belmans, R.: QoS Monitoring at Middleware Level for Dependable Distributed Automation Systems. Proc. of 13th Int. Symp. on Software Reliability Engineering, Nov. 2002, pp. 217–218.

[11] Vaidyanathan, K.—Trivedi, K. S.: A Comprehensive Model for Software Rejuvenation. IEEE Trans. on Dependable and Secure Computing, Vol. 2, 2005, No. 2, pp. 124–137.

[12] Lyu, M. R.: Design, Testing, and Evaluation Techniques for Software Reliability Engineering. Proc. of 24th Euromicro Conf. on Engineering Systems and Software

for the Next Decade (Euromicro '98), Workshop on Dependable Computing Systems, Vasteraas, Sweden, pp. xxxix–xlvi.

[13] Murray, H.: Performance Perspectives. Rules-of-Thumb for Monitoring Windows NT/2000 and Domino Statistics. June 2002, available on SUN web site: `http://www-10.lotus.com/ldd/today.nsf/perf?OpenView`.

[14] Anderson, R.: Finding Leaks and Bottlenecks with a Windows NT PerfMon COM Object. January 1999, available on: `msdn.microsoft.com/library/en-us/dnperfmo/html/perfmon.asp`.

[15] Stecher, B.—Charest, M.: Tick-Tock – Understanding the Neutrino Microkernel's Concept of Time. Part I. Available on: `http://www.qnx.com/developer/articles/oct2300a/`, Part II. available on: `http://www.qnx.com/developer/articles/oct3100c/`.

[16] Kavas, A.—Feitelson, D. G.: Comparing Windows NT, Linux, and QNX as the Basis for Cluster Systems. Concurrency and Computation: Practice and Experience, Vol. 13, 2001, No. 15, pp. 1303–1332.

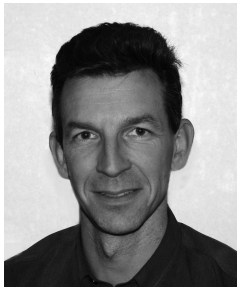
[17] Kovalenko, I.: SPIN – System Performance Monitor for Neutrino. Available on: `http://home.attbi.com/ kovalenko/qnx/spin/`, 2001.

[18] Simple Network Management Protocol (SNMP) Web Site. Available on: `http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/snmp.htm`.

[19] Ren, Y.—Bakken, D. E.—Courtney, T.—Cukier, M.—Karr, D. A.—Rubel, P.—Sabnis, C.—Sanders, W. H.—Schantz, R. E.—Seri, M.: AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. IEEE Trans. on Computers, Vol. 52, No. 1, January 2003, pp. 31–50.

[20] Kim, K. H.—Subbaraman, C.: Dynamic Configuration Management in Reliable Distributed Real-Time Information Systems. IEEE Trans. on Knowledge and Data Engr., Vol. 11, No. 1, Jan./Feb. 1999, pp. 239–254.

Rodica Tirtea is university lecturer (lector) in the Computer Science Department, Faculty of Electrical Engineering and Information Technology, University of Oradea (Romania). She received her engineering degree in 1998, her M. Sc. in computer science in 1999 from University of Oradea and her Ph. D. in engineering in 2005 from K. U. Leuven (Belgium). Her research interests include dependability and security in distributed systems.



Geert Deconinck is full professor (hoogleraar) at K. U. Leuven (Belgium). As staff member of ESAT/ELECTA (Electrical Energy and Computing Architectures), he performs research on designing dependable system architectures for industrial automation and control, assessing their dependability attributes and characterizing infrastructure interdependencies. He received his M.Sc. in electrical engineering (1991) and his Ph. D. in engineering (1996) from K. U. Leuven, and was postdoctoral fellow of the Fund for Scientific Research – Flanders (1997–2003). He is chairman of the TI society BIRA on industrial automation and a member of the IEEE SMC TC on Infrastructure Systems and Services, a member of the Royal Flemish Engineering Society and a senior member of the IEEE (Reliability, Computer and Power Engineering Societies).