Performance Evaluation for Distributed System Components

Petre DINI, Gregor v. BOCHMANN, Raouf BOUTABA

University of Montreal, CP 6128, Succ. CENTRE-VILLE
Computer Science and Operation Research Department
Montreal, (Qc), H3C 3J7, Canada

Computer Research Institute of Montreal
1801, McGill College street, #800
Montreal, (Qc), H3A 2N4

Abstract

The performance evaluation of hardware and software system components is based on statistics that are long views on the behavior of these components. Since system resources may have unexpected behavior, relevant current information becomes useful in the management process of these systems, especially for data gathering, reconfiguration, and fault detection activities. Actually, there are few criteria to properly evaluate the current availability of component services within distributed systems. Hence, the management system can not realistically select the most suitable decision for reconfiguration. In this paper, we present a proposal for a continuous evaluation of component behaviour related to state changes. This model is further extended by considering different categories of events concerning the degradation of the operational state or usage state. Our proposals are based on the possibility of computing at the component level, the current availability of this component by continuous evaluation. We introduce several current availability features and propose formula to compute them. Other events concerning a managed object are classified as warning, critical or outstanding, which leads to a more accurate operational view on a component. Several counter-based events are thresholded to improve predictable reconfiguration decisions concerning the usability of a component. The main goal is to offer to the management system current relevant information which can be used within management policies. These policies could refer to the enhancement of the trading-based system services, the flexible polling frequency tuned with respect to the current evaluation, or particular aspects related to dynamic tests within distributed systems. Implementation issues with respect to the standard recommendations within distributed systems are presented. Finally we describe how the reconfiguration management system can use these features in order to monitor, predict, improve the existing configuration, or accommodate the polling frequency according to several simple criteria.

Keywords: *current availability, component health, adaptive reconfiguration*

1: Introduction

In the object-oriented approach, system resources are represented by objects portraying many behavioural facets. Commonly, two distinct specifications of the behavior of a resource co-exist. One specification describes the operational behavior of a resource, i.e. what it does independently of any management, its own functionality: the TCP/IP protocol, modem behavior, bridge's operations. The other specification defines how a resource can be managed in correlation with the operational behavior, i.e. its management behavior. It constitutes the managed object definition, that must be written in accordance with the information structure of the MIB (Management Information Base). A management system is an application which consists of specialized managing objects playing different management roles, such as monitoring, fault detection, or reconfiguration. These roles are fulfilled based on the information they collect across management operations on managed objects, by interpreting the results of these operations that come back. In the pro-active management approach, the collection of information is initiated by managing objects. Often, real resources behave unexpectedly. When a relevant event happens to or in the resource, its appropriate managed object may spontaneously generate notifications. Consequently, the managing objects interpret not only the operation results, but also object notifications. To allow this interaction between managing and managed objects, management operations and notifications are two important features which must be defined by the management specification. Management operations constitute the management interface of a managed object. In object-oriented system specifications, a managed object is an instance of a type. Consequently, a type definition must be documented with visible properties, favouring the interaction of its instances with the management system. These properties are represented by 1) state attributes which represent the operational, usage or administrative state of the concerned resource, 2) management operations, that can affect either the managed object attributes, at the instance level, or the managed object as a whole, 3) matching rules to apply the CMIP filters [4], 4) management behavior, according to the func-

tional behavior of resources, 5) notifications and circumstances to emit them, 6) specific packages, as well as its position in the inheritance hierarchy [ISO/IEC 10165-4]. In this paper, we are concerned with the use of state changes and notifications to evaluate the performance of a system component, which becomes relevant in self-reconfigurable distributed systems.

One of the more challenging problems associated with distributed systems is the subject of system management. Monitoring the functionality of complex distributed systems implies various specific *management activities* with respect to resource allocation, dynamic system changes, and evaluation of these changes in order to offer the QoS (Quality of Service) desired by the users. In small systems, management activities are performed either internally, by network operating systems, or externally, in an ad hoc manner, by the system operator. Since dynamic prescribed changes or unexpected component behaviors, due to either users or the resource availability, are often possible within distributed systems, management aspects have became increasingly complex. The system operator must correlate its reconfiguration actions with respect to dynamic changes occurring in a system. Since within large computer systems the human operator is overwhelmed by various conflicting situations, management systems gradually overtake most human operators' tasks.

Since dynamic properties allow an active management, we focus in this paper on state changes and notifications. The main purpose of this paper is to propose a dynamic quantitative evaluation of a system component behavior, which may be considered in automatic reconfiguration policies. The goal is to offer to the management system current relevant information, which can be used within management policies implemented as the operational behavior of managing objects. Our proposal is based on the possibility of computing, across a managed object, the current availability of the resource it represents by a continuous evaluation. A model of the *current availability* is presented. We introduce several current availability features of a managed object, and propose formula for computing them. Other events concerning the operational state of a managed object are classified as *warning*, *critical* or *outstanding*, which leads to a more accurate operational view on a component, i.e. different alarms concerning the operational state are classified with respect to their relevance. A combination of these two concepts allows us to define the *health* of a managed object. For those managed objects portraying a maximum capability, the capability range is *thresholded* between idle and busy, to accurately capture the loading. Consequently, several counter-based events concerning the usage state are evaluated with respect to *threshold1* and *thershold2*, which lead to a warning or critical usage state. We define the relation *is-better-than*, that creates an ordering between system components offering an identical or similar service. Based on our model, several monitoring and reconfiguration policies are described, as well as particularities for implementing them.

The outline of this paper is the following. Shortcomings of related work are presented in Section 2. In Section 3, we introduce the models for the performance evaluation: the current availability and its derived features. Section 4 presents monitoring and reconfiguration policies based on our models: how these new properties can be used by the management system in order to solve current allocation or monitoring problems. Section 5 presents the implementation experience of our proposal related to timestamps in a real system. Section 6 focuses on the utility of this proposal, while Section 7 gives some conclusions on our proposal and future work.

2: Related work

2.1: States, actions, and state changes

Different classes of managed objects have a variety of *state attributes*. A change of at least one state attribute value determines a *state change* for the concerned component. A state change requires an *action* to be fired for modifying an attribute value. This action could be either *internal* to a managed object (*event*), or *external*, issued from the managing system (*command*). Consequently, a managed object representing a real resource provides two kinds of state attributes. The first category refers to the attributes whose values are updated by the resource itself in order to correctly present its *operationalState* (*enabled*, *disabled*). A refinement of the operational state *enabled* can be specified by another state attribute called *usageState* (*idle*, *active*, *busy*). By interpreting these values, the managing system may either apply policies within the subsystem with respect to the state of one particular component, or decide to directly react across an administrative state attribute of this component, by means of commands. Commands act on the second category of state attributes, that is, attributes whose values are managed by the managing system. For instance, the value space *{locked, shutting down, unlocked}* of the *administrativeState* attribute is modified only by the managing objects.

State change aspects within distributed systems are important, since they are the basis for building management policies. Since state combinations leads to a complex management task, the *state change management* considering all component parameters is difficult to be performed. Hence, the managing system needs a *unique criterion* to evaluate and decide actions in response to *state changes*. In many situations, the resource allocation is based on human decisions in an ad hoc manner, and thus, it takes no advantages from an optimal and automatic process. More current information is needed on state changes with respect to internal actions, which realistically characterize *current changes* at the *component level*.

Despite several global approaches, the evaluation of the quality of the operational state using an unique criterion is currently *based on statistics*. In the following, we propose *a continuous evaluation of state changes*, based on the possibilities of measurement within distributed systems.

2.2: Performance evaluation

Reliability, *maintainability*, and *availability* are major topics with respect to the utilization of a software and hardware component. In order to cope with state change issues, many funndamental performance models based on state spaces, and taking into account these concepts have been proposed [1][2][10][11][12][13][16]. The *availability* represents the probability that a system *is able to work* at any time during a given period. All studies consider that a component continuously alternates from the operational state *enabled* to the repair state *disabled* [2][10]. Actual studies are entirely based on *probabilistic approaches*, where appropriate density functions *statistically approximate* the component availability. The case of *operational interruptions without repair is less studied*, although it is quite common within distributed systems, since it implies a *continuous measure*. However, even if a measure is continuously performed, the instant of a failure cannot be predicted, and statistic values serve only as a relative comparison [10]. Pagès and Gondran have introduced the *instantaneous failure rate* and *instantaneous repair rate* for a system, but their calculi use statistics laws, and only a *manual method* for solving small systems is proposed [12].

2.3: Continuous measurements

The problems are: 1) what must be represented as relevant information within a managed object, and 2) how this information could be accurately interpreted by the managing objects. The first question is answered by existing standard recommendations. A first proposal for the standardization of state attributes of managed objects is given in [ISO/10164-2]. Two attributes define internal component states, i.e. governed by internal events, namely *operationalState* and *usageState*, whereas the *administrativeState* attribute allows management commands (external events). Based on space values of these attributes, several critical combinations are identified as relevant for the managing objects. A special record class, called *stateChangeEventClass*, is proposed in order to record state changes or other relevant counter-based, or type-based events. Based on this model, we can compute several continuous parameters in order to evaluate the current availability of a component, regardless statistics laws.

To answer the second question, there are some difficulties related to the complexity of calculi. However, the values of previous parameters can be entirely *computed*, based on the *state model* proposed by standards. Consequently, the managing system has neither a unique criterion to evaluate and decide actions in response to *state changes*, nor to globally evaluate the state of many cooperating components with respect to the type of their relations at a given moment. As a result, the transparency of management decisions is difficult to be achieved, guaranteed or improved.

Also, the lack of the pertinent computable information at the component level does not allow us to infer predictive management decisions with respect to the component behaviour. Moreover, when changes within system reconfigurations occur due to relation changes or component changes, the management system can neither evaluate the degree of the enhancement of services, nor detect critical areas with respect to a degradation of offering these services. Since there are no criteria to properly evaluate the current availability of services within a DS (Distributed System), the management system can not realistically select the most suitable solution.

3: Model for the performance evaluation

Our proposal concerns three related facets of state attributes and notifications. First, we define the semantics and computing formula for the current availability, based on operational state changes. Second, a combination between the current availability and severity-based events is used to define the health of a system component. Finally, some thresholded events concerning the usage state are proposed and interpreted in concert with the health.

The state change model within the large distributed systems identifies the major cooperating parts involved in the management of state changes. A simplified model of DSs and the interaction with its managing system is presented in Figure 1. Real DS resources are abstractly represented by managed objects in management repositories, conforming to MIBs. Currently, real DS resources send events, e.g. *enableState* or *disableState*, reporting their *operational state changes*. For example, the event *enableState* determines the operationalState attribute value to be *enabled*.

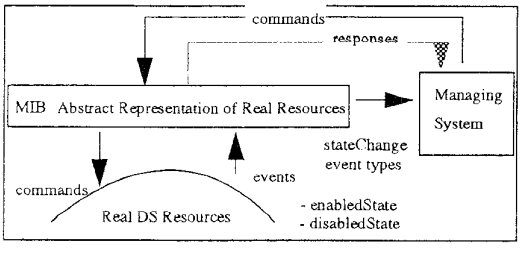

Figure 1. State change model for real DS resources

3.1: Current availability

From the management point of view, we distinguish three kinds of availability: 1) *actual*, 2) *estimated*, and 3) *objective*. The *actual availability* represents the model of the real world. Its value may continuously change, according to the concerned system resource. The *estimated availability* defines the availability value, as calculated at different instants in time. The *objective availability* is the availability value, as considered by the system specification. The objective availability is that claimed by the component producer, as a result of statistical measurements across many groups of products, and over long time (sometimes, it is called *asymptotic availability*), e.g. 0.9996 for a satellite channel [16]. The estimated availability is commonly computed each time an operational state change occurs. Its value is valid up to the next computing, and constitutes the unique value considered by the managing system. Hence, hereafter we refer to it as *current availability*, as viewed by the managing system. Between two sequential computations, the actual availability value really defines the component availability. The current availability substitutes at the management level the actual availability during this interval. Consequently, when the current availability is computed, its value is equal to that of the actual availability. In time, the current and actual availability tend to the objective availability.

According to the preceding definitions, the current availability represents an estimation obtained by measurements, based on timestamps of operational state change events, as shown in Figure 2.

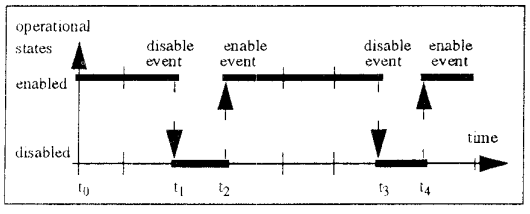

Figure 2: Abstraction of operational state changes by state diagrams

A useful representation of state transitions by using the FSM (Finite State Machine) shown in Figure 3, allows us to identify *internal or external events* involved in the management of component states. The *insert* event represents the first event used to communicate to a system that a new resource has been inserted. The first initialisation of its corresponding managed object is performed by the *start-period* event. We are concerned with the resource behaviour after this event occurs.

Each time the component is removed from a system (*remove* event), an *insert* event and then, a *start-period* event are required in order to re-start the component. After an event *start-period* the operational state is enable (Figure 3). A *remove* event can act in both operational states of a component. Internal events (*enable, disable*) determine operational state changes and simultaneously the occurrence of the appropriate external notification (*enableState, disableState*) sent to the appropriate managing object. We can measure this interval between two consecutive events *remove* and *insert*. In conclusion, this model permits all measures for creating statistics on the availability. In the following, we are concerned with how to compute *current availability values.*

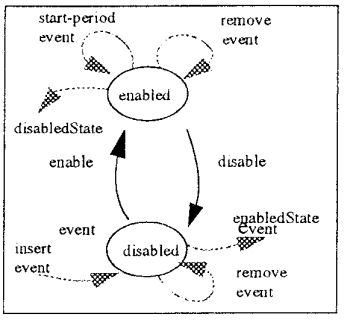

Figure 3. Finite state representation of the operational state of a real resource

where the events have the following meaning:

start-period: external start event at the first initialisation of the resource represented by these states and transitions;

start-period action, when the resource has been deliberately disconnected by external reasons (such of resource removals) and the real state was enabled;

start -period action, when the resource has been deliberately removed, being in the state disabled;

insert: each time a resource begins a new period after a repair activity or maintenance activity (after *remove* event)

enable: operational state change event from the disabled state to the enabled state ;

disable: operational state change event from the enabled state to the disabled state

3.2: Current availability and derived features

Current availability is a feature of a system component representing the availability of component's services up to a given time. Its value defines how long this component has been in the operational state enable since its initialization.

Consequently, the current availability value is a *continuous function of time*, defined as a quotient between the amount of time where the resource was been in the enabled state (commonly called *operational time*) and the

observation period (between the time t_0 of the event *start-period* and the time t of the end of the observation period).

$$T = [t_0, \infty\ ],\ t_0 \text{ is the timestamp where the event start-period occurs} \quad (I)$$

$$a:\ T \rightarrow [0, 1.0]$$

$$a(t) = \frac{1}{t - t_0} \int_{t_0}^{t} \lambda(t)\, d\, t,$$

$$\text{where } \lambda(t) = \begin{cases} 1, \text{ if (operational state at time t = enabled)} \\ 0, \text{ otherwise} \end{cases}$$

As shown by formula (I), the actual availability value is a *continuous function of time*, defined as a quotient between the amount of time where the resource was been in the enabled state (commonly called *operational time*) and the observation period (between the time t_0 of the event *start-period* and the time t of the end of the observation period).

For management purposes, this formula must be used at any time $t \in T$. However, polling responses and notifications are issued at different timestamps $t_i \in T$. For the computation mechanism, this formula is time and space consuming when it is applied at each state change event, because the system must memorize the behavioral history of state change as <event type, timestamps> and recompute the formulae (I) each time the actual availability value is requested. Consequently, to calculate the $a(t_i)$ values, we utilize an on-line recursive formula (II). This way is less expensive in terms of computing and storage costs.

The actual availability can is calculated each time an *state change event occurs*, or *on-demand* at the initiative of managing objects, i.e. between two state changes. For the definition of computing formula, we consider first two consecutive operational state change events, as presented in Figure 4. Each state change event has the timestamps of its occurrence attached to. The formula (II) allows us to compute the current availability $a(t_i)$, which is valid within the interval $[t_i, t_{i+1})$, by knowing the current availability $a(t_{i-1})$, which is valid within the interval $[t_{i-1}, t_i)$.

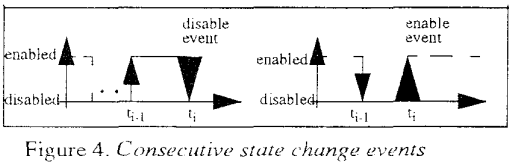

Figure 4. *Consecutive state change events*

$$a(t_i) = \frac{1}{t_i - t_0} \left(a(t_{i-1}) \times (t_{i-1} - t_0) + \lambda(t_i^-) \times (t_i - t_{i-1})\right) \quad (II)$$

where $\lambda(t)$ has the same expression as in formula (I), and t_i^- means just before t_i.

Current availability tendency $\underline{a}([t_{i-1}, t_i], \varepsilon_0)$ is a qualifier of the $a(t)$ variation in an interval delimited by two consecutive operational state changes. As we have seen, the $a(t)$ values vary over time. $\underline{a}([t_{i-1}, t_i], \varepsilon_0)$ is an informal evaluation of the current availability variation with respect to an acceptable variation $\pm\varepsilon_0$ within $[t_{i-1}, t_i]$. The definition of the $\underline{a}([t_{i-1}, t_i], \varepsilon_0)$ is the following:

$$T = [t_0, \infty\ ],\ [t_{i-1}, t_i] \subset T \quad (III)$$

$$\underline{a}([t_{i-1}, t_i], \varepsilon_0) : [t_{i-1}, t_i] \rightarrow \{\text{stable, ascendent, descendent}\}$$

$$\underline{a}([t_{i-1}, t_i], \varepsilon_0) = \begin{cases} \text{stable} & \text{if } |\bar{a}_{id}(t_i) - \bar{a}_{id}(t_{i-1})| \leq \varepsilon_0; \\ \text{ascendent} & \text{if } \bar{a}_{id}(t_i) > \bar{a}_{id}(t_{i-1}) + \varepsilon_0, \text{ and} \\ \text{descendent} & \text{if } \bar{a}_{id}(t_i) < \bar{a}_{id}(t_{i-1}) - \varepsilon_0. \end{cases}$$

where t_{i-1} and t_i are consecutive timestamps of operational state change events.

Based on the current availability values, we introduce the *weighted average of current availability* $\bar{a}(t)$ as a measure with emphasis on recent $a(t)$ values. The more recent $a(t)$ values should be taken into account with a higher weight. Based on Figure 2, where the computed values are performed at t_i, we consider the weighted current availability which emphasizes the latest a(t) values by an exponential factor. This factor takes into account the length of the interval that has been considered:

$$\bar{a}(t_i) = (a(t_{i-1}) + \lambda \times a(t_i) \times \exp(t_i - t_{i-1})) / (1 + \exp(t_i - t_{i-1})) \quad (IV)$$

wherein $\lambda = 0$, if the operational state within $[t_{i-1}, t_i)$ is disabled, and $\lambda = 1$, if the operational state within $(t_{i-1}, t_i]$ is enabled.

3.3: Alarm refinement in operational state enable

We classify alarms occurring in the operational state *enabled* into three severity levels: *warning alarms*, *critical alarms*, and *outstanding alarms*. For example, *low-level-1* of the tank-toner of a printer is a warning alarm. After this alarm occurred, the printer still works, the text quality is good enough, but, if this alarm is not cancelled, i.e. handled by processing its origin and eliminate its causes, the QoS may degrade. We call this state *warningEnabled*. If the cause of this alarm is solved, the operational state becomes *enabled*. If not, a critical alarm may occur, i.e., *low-level-2*, when the printer operational state becomes *criticalEnabled*. From this state, the printer could become *disabled*, if the alarm *low-level-3* occurs. Otherwise, the printer becomes *enabled*, if the cause of the alarm low-level-2 is completely eliminated, or *warningEnabled* if the cause is only partly eliminated. Each time an outstanding alarm occurs, the state disabled is reached. For example, the warming alarm, i.e. a high internal temperature of a printer, determines the de-activation of printer services, regardless the current operational state. The model of alarms classification must be defined for each component type, as shown in Figure 5. It is not mandatory for a component type to prescribe all these kinds of alarm types. Additionally, it is not relevant

if this alarm is sent by the concerned component, or it is captured by other components across their relationships. This means that, even if the concerned object does not specify such of alarms, the transitions of its state change model can be ensured by other partners.

general classification:	*example for a printer*
warningAlarms	warningAlarm: low-toner-level1
criticalAlarms	criticalAlarm: low-toner-level2
outstandingAlarms	outstandingAlarm: low-toner-level3, warming, tray-empty

Figure 5. Alarm classification in the operational state enabled

3.4: Thresholds of the usage state

Many system resources have a limited capacity of their services. This can be expressed as number of clients simultaneously served, memory space, or buffer space. For example, a Lantastic network operating system allows a maximum of 80 stations, but the QoS is fuliy guaranteed only up to 20, while over 75, its services are very slow. Such kinds of thresholds are also typical for CPUs, multimedia servers, e-mail servers, etc. Consequently, we define between idle and busy, two thresholds specific to different types of components. When the capacity occupancy exceeds the first threshold, the usage state becomes warningActive, whereas after the second threshold, the usage state is criticalActive. When a resource is first initialized, its usage state is idle. When a new user is served, the usage state becomes active. According to changes of the number of clients, or the use of the resource capacity, and with respect to threshold1 and threshold2, the component is either in the warningActive state, or critical Active state, respectively. If a new user is served at the limit of the maximum capacity, the the original maximum capacity decreases because of a failure, or a user request a lot of capacity, the usage state becomes busy.

The model of thresholding usage state values is presented in Figure 6. All components portraying a limited capacity and one or both thresholds must accordingly adapt their usage state.

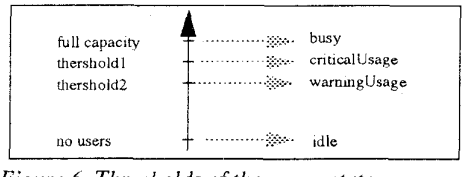

Figure 6. Thresholds of the usage state

4: Performance evaluation

The rationale to compute the current availability and to extend the operational state model and usage state model was to dynamically and transparently capture the real behavior of a system resource and accordingly adapt reconfiguration activities.

4.1: Policies

These models of the component behavior are useful for the system monitoring, and to create client-server like cooperation relations, achieving the best QoS across these cooperation relations. We describe further the use of our proposal concerning the current availability, operational state, and usage state to build management policies concerning the system monitoring and cooperation relation establishement. A management policy may be a part of, or the whole behaviour of, a management application. A management policy is a statement of the form:

```
<policy-name>::= if <condition>
                 then {<actions>
```

where <condition> is a predicates on property values of a system component, while <actions> represent management actions. These management actions can be simple updates of component properties, or complex management activities performed by managing objects.

A management policy can implicitly use generic well-known policies, such as *max* {a, b}, *min* {a, b}, *priority ordering* {a, b, c, d...}, the *FIFO* policy, etc.

Simple management policies can be combined. Potential conflicts between different actions prescribed by each policy are partly solved by using generic policies.

4.2: Monitoring policies

Monitoring policies may independently use criteria based either on the current availability, operational state, or usage state. All following policies concern a given component, or a given set of components. A simple monitoring policy based on the current availability could be expressed as:

```
P1:   if ā(t)< a0
      then
        administrativeState = shutting-down
        and
        pollingFrequency = f (ā(t)), as defined in [5].
```

where a_0 is a threshold of the current availability defined by the management system for those system components playing critical roles for particular applications. Such a component may be a satellite channel, a CPU, an operating system, or a host playing the role of a management station.

A policy based on the operational state could be:

```
P2:  if opState = criticalEnabled, then
         administrativeState = in-active-test
```

or

```
P3:  if opState = warningEnabled
     then
       administrativeState = in-passive-test
       and
       pollingFrequency = f (ā(t)).
```

The usage state values may be used to build monitoring policies. These policies ensures an optimal solution for load-balancing algorithms ([3], see also Section 4.3).

P4: if usState = criticalActive,
then
administrativeState = shutting-down

or

P5: if usState = warningActive,
then
pollingFrequency = g(usState), as defined in [6].

These simple policies can be combined to monitor more complex situations. The problem raised in this case is the potential contradiction between management actions independently specified by each policy. Some generic policies are used to solve these contradictions, which are particular to different contexts. Let us consider P3 and P5, leading to the policy P35. Then, the choice of the polling frequency is performed by the generic policy expressed as max {a, b}.

P35: if opState = warningEnabled,
and
usState = criticalActive
then
pollingFrequency = max {f ($\bar{a}$(t)), g(usState)}

Other contradictions may appear with respect to the changes decided for the administrative state. In this case, a policy establishing a priority between the values of the administrative state is requested. For example, if we consider the policies P1 and P3, the policy P13 may be as follows:

P13: if $\bar{a}$(t)< a_0,
and
opState = warningEnabled
then
administrativeState = shutting-down
and
pollingFrequency = f ($\bar{a}$(t)).

In this case, a priority policy between the states of the administrative model has been introduced, i.e. shutting-down *has-a-greater-priority* than in-passive-tests.

4.3: Establishing cooperation relations

Commonly, establishing cooperation relations implies many kinds of constraints, expressing the requested QoS or the current state of the system resources which must interact. In distributed systems, identical or similar services can be offered by many resources. QoS issues concerning static properties of potential cooperating resources, offered as interface constraints by their appropriate managed objects, have been presented in [6]. In the following, we emphasize QoS constraints related to the real performance of cooperating objects. The problem is, how to select the most available server for a client, based on the current measures and models that we have proposed in this paper. We define the relation *is-better-than*, represented as *"C1 ≻ C2"* and read *C1 is-better-than C2*, between two components *C1* and *C2*, if, from the management point of view, the *QoS offered by C1* is better than the *QoS offered by C2*, based on their current availabilities, operational state values, and usage state values. The same relation can be used for ordering operational state values and usage state values, as well as current availability values. By definition,

enabled ≻ *warningEnabled* ≻ *criticalEnabled* ≻ *disabled*;
idle ≻ *warningUsage* ≻ *criticalUsage* ≻ *busy*, *and*
$\bar{a}(t)_{C1} \succ \bar{a}(t)_{C2}$, if $\bar{a}(t)_{C1} > \bar{a}(t)_{C2}$,
where $\bar{a}(t)_C$ represents $\bar{a}(t)$ of the component C.
$\mathcal{A}[t_k, t_l], \varepsilon_0)_{C1} \succ \mathcal{A}[t_k, t_l], \varepsilon_0)_{C2}$, if $\mathcal{A}[t_k, t_l], \varepsilon_0)_{C1} > \mathcal{A}[t_k, t_l], \varepsilon_0)_{C2}$
with $[t_k, t_l] \subseteq [t_{i-1}, t_i]$

We define the *health* of a component with respect to its weighted current availability and its operational state, as $h(t)_C = < opState, \overline{a(t)}>_C$. We present several decision policies based on the health of a component and its usage state. Consequently, a management policy *C1* ≻ *C2*, called *operational-state-first*, can be defined as:

$$h(t)_{C1} \succ h(t)_{C2} \Leftrightarrow ((opState_{C1} \succ opState_{C2}) \vee ((\bar{a}(t)_{C1} \geq \bar{a}(t)_{C2}) \wedge (opState_{C1} = opState_{C2}))$$

while a management policy *C1* ≻ *C2*, called *current-availability-first*, can be defined as:

$$h(t)_{C1} \succ h(t)_{C2} \Leftrightarrow (\bar{a}(t)_{C1} > \bar{a}(t)_{C2}) \vee ((\bar{a}(t)_{C1} = \bar{a}(t)_{C2}) \wedge (opState_{C1} \succ opState_{C2}))$$

QoS is directly dependent of the loading for many types of servers. Consequently, we define the tuple $<health, usState>_C$, as a potential performance of a given component C. Consequently, a management policy *C1* ≻ *C2*, called *health-first*, can be defined as:

$$C1 \succ C2 \Leftrightarrow ((h(t)_{C1} \succ h(t)_{C2}) \vee ((h(t)_{C1} = h(t)_{C2}) \wedge (usState_{C1} \succ usState_{C2}))$$

while a management policy *C1* ≻ *C2*, called *usState-first*, can be defined as:

$$C1 \succ C2 \Leftrightarrow (usState_{C1} \succ usState_{C2}) \vee ((usState_{C1} = usState_{C2}) \wedge (a(t)_{C1} \succ a(t)_{C2}))$$

Other policies can be used to select a server, e.g. a cost-based policy [7]. Consequently, we can combine cost and functional aspects in different policies to select a better server. A policy called *cost-only* can be defined as:

$$C1 \succ C2 \Leftrightarrow (cost_{C1} < cost_{C2}).$$

5: Implementation aspects

5.1: Definition of the current availability function and its computing algorithm

We are only concerned here with the on-line computing using the *on-line current availability function* and the *on-line algorithm* implementing it. For simplicity we will

subsequently use the $A_{id}(t_i) = \{a(t_i), \underline{a}(t_i), \underline{a}([t_{i-1}, t_i], \varepsilon_0)\}$ notation to describe the properties related to the current availability at the time t_i for a system component *id*. The *on-line approach* refers to the *in-time* and *recurrent* computing of $A_{id}(t_i)$ values. For each component, all *change events* and *timing stamps* are recorded somewhere, as previously presented. The *on-line function* infers new $A_{id}(t_i)$ values, based on $A_{id}(t_{i-1})$ values, by considering the state change event (disable, enable). The on-line function requests as input data only the component identifier, the identification of the change event, and the $A_{id}(t_{i-1})$. For simplicity, in the on-line algorithm we consider the current availability tendency threshold $\varepsilon_0 = 0$.

5.2: Implementation aspects concerning change events

To apply the computing function, we have assumed that input data events are recorded somewhere. In fact, in the actual networks, these data are registered by StateChangeEventRecord class [ISO/IEC 10164-2] which offers additional information on change events. We have identified three classes of problems which can arise by applying the algorithm implementing the on-line function, that is, (1) the change record capacity, (2) the mode of the record deletion, and (3) the scheduling of measurement periods. The *change record capacity* (the MaxLogSize attribute of the LogRecords class) limits the number of computed values if MaxLogSize value is *determinate* [ibidem]. If MaxLogSize is *indeterminate*, the algorithm can be applied any time. The *mode of the record deletion* (the LogFullAction of the LogRecords class) determines the maximum time interval where data are available for the algorithm. If the LogFullAction value is *wrap*, the earliest set of records will be deleted. Consequently, the algorithm has not a long view on input data, and it can not be retroactively used over the MaxLog Size value. If the LogFullAction is *halt*, records already in the log will be retained, but no more records will be logged. In this case, the computed $A_{id}(t_i)$ values are not updated. The algorithm can be useful for only state change records up to the *halt* moment. The *scheduling manner* affects twice the algorithm: first with respect to the input data (computing within the interval $[t_{min}, t_{max}]$), and second, related to the initialisation data. LogRecords presents the LogSchedullingPackages attribute having three option values: daily, weekly, or a $[t_{start}, t_{stop}]$ period. Thus, the input computing interval of the algorithm must be less or equal to LogRecords scheduling interval. In the case of a new scheduling period, the initial health values must be the last computed values within the previous scheduling period.

5.3: Aspects of the on-line initialisation

Two aspects related to the initialisation are relevant. The first refers to the *recording of current availability values*, and the latter to the *usability mode of the system*. The on-line algorithm needs the last computed $A_{id}(t_{i-1})$ values to calculate the updated values at t_i. Four implementation solutions are possible namely, (1) appropriate current availability attributes of the managed object representing the real DS resource represent computed values, (2) a special currentAvailabilityChangeRecord class which inherits from the StateChangeEventRecord records these values, (3) there is a special currentAvailability class inheriting from LogRecord class which records only the $\{A_{id}(t_s), s \leq i\}$ values, and (4) a special current availability data base dedicated to these values.The first case implies the addition of several attributes representing $A_{id}(t_i)$ values. Consequently, the MIB components must be slightly completed.

In the next two solutions, the new classes partially inherit also several constraints related to the mode of the *record deletion mode* and to the *scheduling manner for recording*. If the mode is *wrap*, the link between the computed health values at t_{i-1} and the computing step at t_i can be broken. If the mode is *halt* at t_s there will be several t_i ($t_s < t_i$), the on-line algorithm applies with errors across the time interval between t_s and t_i (halt(t_s) and $A_{id}(t_i)$). A similar aspect appears between stop ($period_p$) and restart ($period_{p+1}$) scheduling periods.

In the fourth case, the $A_{id}(t_i)$ records are independent of aspects arising due to the LogRecord class. Moreover, $A_{id}(t_i)$ values are individually recorded for each DS component. Consequently, the managing objects can easily evaluate in time the current availability of each component.

The input data at t_i are based on $A_{id}(t_{i-1})$. Since the registered timing is *system-use dependent*, we have identified three *functional continuity contexts* with respect to the initialization of the on-line algorithm: (1) the DS is continuously used without breaks, (2) the DS is used periodically, and (3) the DS is used intermittently without a well defined frequency. Regardless of the context, the $A_{id}(t_i)$ values must be computed at correct time stamps.

If the system *continuously runs*, the on-line algorithm allows to easily pass from a computing period (t_T) to another, since only few data must be stored namely, $A_{id}(t_T)$ values, where t_T is the length of a considered period. If the event pair (*remove, start-period*) is within the same running period of a system, we can compute the *unavailability time* for the *interruptions with repair components*.

The event *start-period* ensures the initialization as prescribed in the input data of the on-line algorithm. If the system works in these two contexts, the inactivity period is not caused by the component. Consequently, this time is ignored in computing current values. The *start-period* event for a component must consider the initial values corresponding to the values computed at the end of the precedent period.

6: Using current availability features within DSs

Let us suppose now that we have somewhere current availability values as previously introduced. In the general case, a managing system may have three kinds of views on the current availability values of real DS resources, depending on the period that the managing system keeps these values: (1) the last updated values, (2) a set of values within one period, and (3) a set of values across several successive periods. The managing system can use these values for different purposes as follows, as suggested by different kinds of policies presented in Section 3:

1. - to build availability statistics on new DS components;
2. - to update availability statistics on existing DS components;
3. - to monitor DS components with respect to guaranteed threshold values for their availability;
4. - to predict future current availability of a component;
5. - to establish consistent customer-provider cooperation relations based on the current availa - bility values.
6. - to improve the existing DS configuration;
7. - to accommodate the polling frequency according to the current availability tendency.

Since the first two cases are straightforward, we concentrate on the remainder. Let us take again the satellite channel example, whose *availability guaranteed* is A = 99.6. If, for instance, one accepts a deviation of $\varepsilon_0 = 2.5$, the *accepted availability* becomes 97.1. However, each satellite channel presents its own *current availability* at run time. Several scenarios could be considered. If the channel has a decreasing *current availability tendency*, the management system must simultaneously look for another channel (*prediction*) and indicate this aspect to the reconfiguration module. If the *guaranteed threshold* is nearly reached, the *polling frequency* must be updated, in order to capture the *current evolution* more frequently. Once the *accepted availability* is reached, the managing system must *lock* the administrativeState of the corresponding managed object, avoiding an *in-chain degradation*.

Let us suppose now that a *high priority application* needs a host node within a network. Knowing the $A_{id}(t_i)$ values, a manager can choose between the most available nodes (*allocation aspects*). Even further, if a system component needs services of another system component with several constraints expressed by *requested* $A_{id}(t_i)$ *values*, the allocation can be performed by taking into account *current availability values offered* by potential customers and other policies presented in Section 3.

Finally, if no requested services are detected, but several DS components have a decreasing current availability, the management systems may decide to reconfigure a part of the DS in order to ensure the system survivability.

This approach has been implemented in two distinct applications. The first one has considered management procedures for evaluating the provider health in a hierarchical architecture. The work has been implemented at the University of Montreal by using the language Mondel in the OSIMIS [15] environment. The second application refers to the variable polling frequency used by managers within distributed systems to get current information on component states. The optimization of the polling frequency is based on the operational state and the health evaluation. The implementation platform consists of SNMP-agents [8] and a SNMP/CMIP-proxy [17].

7: Conclusion and future work

We have presented a way to evaluate the availability of a system component in real-time. We have proposed a procedure to compute the current availability, and defined several derived features (*current availability tendency* and *weighted average of current availability*) used for monitoring system components represented as managed objects. Different aspects related to a real implementation of a computing algorithm according to existing real systems are described. Several management issues using current measured values are presented.

A combination of real-time measurements and certain refinements of the operational and usage states of a managed object allowed us to propose different management policies. Based on the relation *is-better-than*, that we proposed, we have presented various concrete combinations of simple management policies.

We have considered a single system component at a time. We have shown that even in this simple case, the QoS management can be enhanced by our approach. The algorithm for computing the current availability has been fully implemented.

The next aspect is how these component features may be combined within a subsystem having many components which interact. We are currently working on an algorithm to automatically infer similar properties for subsystems composed of several components.

Acknowledgements

This work was funded by the Ministry of Industry, Commerce, Science and Technology, Quebec, under IGLOO project organized by the Computer Research Institute of Montreal, and by a grant from the Canadian Institute for Telecommunication Research (CITR) under the Networks of Centres of Excellence Program of the Canadian Government. The authors thank Catherine Agbaw and Kamel Bendaas for implementing certain aspects related in this paper. Reviewers' comments helped us to improve the presentation.

References

[1] Harold Aschold and Harry Feingold, *Repairable Systems Reliability: Modeling, Inference, Misconceptions, and Their Causes*, Marcel Dekker Inc., New York, 1984, (Lecture Notes in Statistics, Volume 7)

[2] Alessandro Birolini, *On the Use of stochastic Processes in Modeling Reliability Problems*, Springer-Verlag, 1985 (Lecture Notes in Economics and Mathematical Systems, 252)

[3] Raouf Boutaba, Bertil Folliot, Pierre Sens, *Efficient Resource Management in Local area Networks*, in Proc. of the International Conference on Advanced Information Processing Techniques for LAN and MAN Management, IFIP WG 6.4, Versailles, France, Avril 1993, pp. III/29-38

[4] ISO/IEC 9596-1:1991, *Information technology - Open System Interconnection - Common Management Information Protocol - Part 1: Specification*, CAN/CSA-Z243.142-91

[5] Petre Dini, Catherine Agbaw, *Real-Time Pro-active Management of Distributed Systems Based on Variable Polling Frequency*, Technical Report, IGLOO Project, CRIM/Univer-sity of Montreal, December 1995

[6] Petre Dini, Gregor v. Bochmann, Isaam Hamid, *Dynamic Constraints Specification of Object Interactions Within Distributed Systems*, in "Dynamic Modification of Distributed Systems Specification using Object-Oriented Techniques", ed. Issam Hamid, Project Number 06044195, sponsored by The Ministry of Science, Culture, and Education of Japan, Japan, March 1996

[7] Petre Dini, Gregor v. Bochmann, *Modeling QoS Multimedia Costs in Distributed Systems*, The 1996 Pacific Workshop on Distributed Multimedia Systems, Hong Kong University of Science and Technology (HKUST), Hong Kong, June 25-28, 1996.

[8] Sidnie Feit, *SNMP: A Guide to Network Management*, McGraw-Hill, Inc., 1995

[9] Charles R. Kime, *System Diagnosis*, in: Fault-Tolerant Computing - Theory and Techniques, ed: Dhiraj K. Pradhan, Prentice-Hall,1986, New Jersey 07632

[10] Krishna K. Misra, *Reliability Analysis and Prediction*, Elsevier, 1992

[11] Michael K. Molloy, *Fundamentals of Performance Modeling*, Macmillam, Publishing Company, 1989, New York 10022

[12] Alain Pagès and Michel Gondran, *System Reliability: Evaluation & Prediction in Engineering*, Springer-Verlag, 1986

[13] Shahen Neyaz, *Estimation of Reliability Parameters of a Redundant System with one Standby and one Repair Facility*, Master Thesis, Concordia University, 1987

[14] J.J. Stifter, *Computer-Aided Reliability Estimation*, in: Fault-Tolerant Computing - Theory and Techniques, ed: Dhiraj K. Pradhan, Prentice-Hall,1986, New Jersey 07632

[15] *** A Guide to Implementing Managed Objects Using the GMS, Version 2.99.1, UCL, October 1992, Draft1

[16] Pramode Verma, *Modèles de performances des réseaux*, InterEditions, Paris, 1992

[17] Catherine Agbaw, *Management Data Collection and Gateways*, M.Sc. Thesis, McGill University, 1994.