QoS Self-Configuring Failure Detectors For Distributed Systems

Alirio Santos de Sá and Raimundo José de Araújo Macêdo

Distributed Systems Laboratory - LaSiD / Computer Science Departament
Federal University of Bahia, Salvador, Bahia, Brazil
{aliriosa,macedo}@ufba.br

Abstract. Failure detectors are basic building blocks from which fault tolerance for distributed systems is constructed. The Quality of Service (QoS) of failure detectors refers to the speed and accuracy of detections and is defined from the applications and computing environment under consideration. Existing failure detection approaches for distributed systems do not support the automatic (re)configuration of failure detectors from QoS requirements. However, when the behavior of the computing environment is unknown and changes over time, or when the application itself changes, self-configuration is a basic issue that must be addressed - particularly for those applications requiring response time and high availability requirements. In this paper we present the design and implementation of a novel autonomic failure detector based on feedback control theory, which is capable of self-configuring its QoS parameters at runtime from previously specified QoS requirements.

1 Introduction

Enterprise Information Technology infrastructures make use of different components of hardware and software to supply highly available, secure and scalable services for applications with distinct quality of service (QoS) requirements. In order to fulfill such requirements, these infrastructures require mechanisms which guarantee a quick reaction and recovery in the presence of system component failures. In this context, failure detectors are fundamental for monitoring failures, enabling recovery processes to be triggered. Thus, design and implementation of failure detectors have been object of intense research in recent decades [1–8].

In distributed systems over networked computers, failure detectors are implemented by monitored and monitor processes which periodically exchange messages each other. To guarantee a quick recovery in presence of component failures, the monitoring period has to be as short as possible. However, shorter monitoring periods can increase resource consumption, compromise the application response time and decrease the efficiency and speed of the detection and recovery mechanisms. In addition, complications emerge when the computational environment or application characteristics change at runtime. In these scenarios, adjusting the monitoring period automatically is a tremendous challenge which has not been appropriately addressed in the literature. Most of the published papers have mainly focused on adaptive detection which uses prediction mechanisms to calculate detection timeout without taking into account the dynamic adjusting of monitoring periods [2, 4–6]. The few papers which consider the dynamic configuration of a monitoring period do not take into account the QoS metrics generally accepted in the literature [9, 7, 8].

Paper Contributions. This paper proposes a failure detector that is capable of self-configuring its operational parameters in response to changes in the computing environment or application at runtime, according to user-defined QoS requirements. Systems with such characteristics are known as autonomic where self-configuration is one of the required properties [10]. Most difficulties in the implementation of these self-configuring failure detectors relies on the modeling of the distributed system dynamics - which is hard to characterize using specific probability distribution functions when load varying environments are considered (e.g. cloud computing environments). For modeling such a dynamic behavior of distributed systems we use the feedback control theory commonly applied in the industrial automation systems arena [11]. The proposed failure detector was completely implemented and evaluated by simulation using QoS metrics such as detection time, mistake recurrence time, mistake duration, percentage of mistakes and availability. These metrics allowed us to evaluate the speed and accuracy of failure detections under varying computational loads. Even without related work for a direct performance comparison, we compared our approach with a traditional adaptive failure detector approach manually configured with different monitoring periods. The results showed that, in most cases, our autonomic failure detector performed better than such adaptive failure detector for each monitoring period considered.

Paper Organization. The remainder of this paper is organized as follows. In section 2, it is discussed related work and the theoretical context of the contribution of this paper. In section 3, it is presented the system model and the QoS metrics used to configure and evaluate the proposed failure detector. In section 4, it is described the design and evaluation of our autonomic failure detector approach. Lastly, in section 5, final remarks and future work are presented.

2 Related Work and Theoretical Context

Fault-tolerance mechanisms for distributed systems must guarantee the correct functioning of services even in the presence of faults. The design of these mechanisms considers models with hypotheses about the behavior of the components over faulty conditions. The most commonly used model is called crash model and assumes that failed (crashed) components do not respond to any request. Even when a crash model cannot be verified in real scenarios it can be emulated by masking and fault hierarchy techniques[12]. Detecting crash failures of system components is a basic issue for the working of many fundamental protocols and algorithms used to build dependable systems. For example, in a passive replication scheme, the failure of a primary replica has to be readily detected in order to allow a backup replica to take on the role of failed replica with minimal impact on the distributed application [13]. Crash failure detection usually considers monitored processes which periodically send their state using messages, named heartbeats, to a monitor process. The monitor determines a timeout interval used to define the instant of the arrival of heartbeat messages. If a heartbeat does not arrive in the timeout interval the monitor will believe that the monitored process has crashed. This detection model depends on timeliness constraints about transmission and processing of the monitoring messages.

In an asynchronous distributed system, time bounds for processing and transmission of messages are unknown which makes it impossible to solve certain problems of fault-tolerance in a deterministic way [14]. To address this, Chandra and Toueg (1996) [1] introduced the unreliable failure detectors approach. These failure detectors are termed unreliable because they may wrongly indicate the failure of a process that is actually correct and, on the other hand, may not indicate the failure of faulty processes. Chandra and Toueg demonstrated how, encapsulating a certain level of synchrony, unreliable failure detectors can solve fundamental problems of the asynchronous distributed systems (e.g. consensus and atomic broadcast). Despite the great importance of Chandra and Toueg‘s work for the understanding and solution of fundamental problems in distributed systems, the absence of timeliness bounds of the asynchronous model imposes hard practical challenges for the implementation of failure detectors. One of these challenges is deciding appropriate values for detection timeout. Long timeouts make detection slow and can compromise the system response time during faults of the system. However, short timeouts can degrade failure detector reliability and damage the system performance as many algorithms and protocols which rely on failure detector information can do additional processing and message exchange imposed by false failure suspicions. Therefore, many researchers studied the use of delay predictors in failure detectors implementation (e.g. [4–6]). These predictors suggest, at runtime, values for detection timeout to achieve quick detection with a minimal impact on the reliability of the detector. However, these researches do not consider the dynamic adjustment of monitoring periods which is another important factor for the performance of failure detectors.

The specification of Chandra and Toueg's failure detectors addresses properties which are difficult to evaluate in practice (e.g. "eventually every process that crashes is permanently suspected by some correct process"). Because of that, Chen (2002)[2] defined QoS metrics which have been used to evaluate the speed and accuracy of failure detector implementations. Then, the work of a designer is to use a monitoring period and a delay predictor which deliver a detection service with a QoS level suitable for the related application requirements. When the computing environment behavior does not change over time or when it is possible to estimate this behavior using some probability distribution function then we can adequately configure the failure detector so that it shows a satisfactory QoS level. For example, in [2] a procedure to make an offline configuration for the failure detectors is presented. Also, in the same paper the authors suggest that such procedure can be re-executed whenever the network traffic characteristics change. However, the effects of this re-execution on the detector performance have not been evaluated. In [3], the authors shortly comment on a consensus-based procedure to dynamically adjust the monitoring period when certain conditions occur. However, the authors have not detailed their solution, neither evaluated it in terms of QoS metrics. The authors of [9], [7] and [15] explored the runtime configuration of failure detectors. However, these researches consider that the environment behavior does not change and they do not demonstrate how to dynamically setup a failure detector using QoS metrics such as detection time, mistake duration and mistake recurrence time.

As far as we are aware, this is the first work to propose a failure detector able to dynamically self-configure by adjusting both the monitoring period and the timeout according to a specified QoS.

3 Basic Issues: System Model and QoS Metrics

We consider a distributed system made up of a finite set $\Pi = \{p_1, p_2, ..., p_n\}$ of processes interconnected by unreliable channels, where corrupted messages are discarded in the receptor processes and messages may be lost. No upper bounds are assumed for message transmission delays and processing times. That is, it is assumed an asynchronous distributed system. Processes can fail by prematurely stop functioning (crash faulty model). Byzantine failures are not considered. In this work we develop a crash detection service using a pull monitoring style[16]. In this monitoring style a monitor process p_i periodically asks about the state of a monitored process p_j by sending a *"are you alive?"* (aya) message and so p_j must respond by using a message called *heartbeat* (hb) or *"I am alive!"*. The aya and the hb messages exchanged between one pair of processes p_i and p_j are sequentially marked on each monitoring period (τ). Thus, aya_k is the k^{th} *"are you alive?"* message sent from p_i to p_j and hb_k the corresponding k^{th} *heartbeat* message sent from p_j to p_i. We use s_k and r_k to denote the sending and receiving instants of aya_k and hb_k, respectively - according to the local clock of p_i. There are no assumed synchronized clocks. Let rto_k denote the estimate of the timeout for the receiving of hb_k. If a heartbeat does not arrive after timeout rto, p_i will insert p_j into its suspect list. If p_i receives a heartbeat with a timestamp greater than the last heartbeat received then p_i will remove p_j from its suspect list.

In the configuration and performance evaluation of the detection service we apply the QoS metrics proposed by [2], such as *Detection Time* (TD); *Mistake Recurrence Time* (TMR); and *Mistake Duration* (TM). The *Detection Time* represents the time interval between the crash of a monitored process (p_j) and the time when the monitor process (p_i) suspects p_j permanently. The *Mistake Recurrence Time* measures the time between two consecutive mistakes of the failure detector. Finally, *Mistake Duration* measures the length time that mistake remained. TM and TMR are metrics for failure detector reliability and can be compared to *Mean Time to Repair* ($MTTR$) and *Mean Time Between Failures* ($MTBF$), respectively. Thus, we have used in our work TMR and TM to estimate the failure detector availability (AV) by: $AV = (TMR - TM)/TMR$. The application requirements in terms of QoS detection are defined by TD^U, TM^U and TMR^L which represent the maximum TD, the maximum TM and the minimum TMR, respectively. The notation T_D^U, T_M^U and T_{MR}^L is used in [2] where U and L represent the upper and lower bounds, respectively - we are following this pattern of notation with some differences (TD, TM, TMR instead of T_D, T_M, T_{MR} as originally proposed in [2]).

4 Design and Evaluation of the AFD Approach

The autonomic failure detector (AFD) proposed is made of an autonomic manager (or controller) which observes the behavior of a basic failure detection service module (plant or managed element). Then, based on the previously defined QoS detection requirements, the autonomic manager calculates the monitoring period and the detection timeout which a monitor process p_i has to use to check the state of a monitored process p_j (see Figure 1).

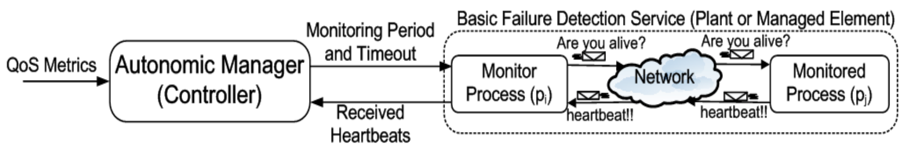
QoS Metrics
Autonomic Manager (Controller)
Monitoring Period and Timeout
Received Heartbeats
Basic Failure Detection Service (Plant or Managed Element)
Are you alive?
Are you alive?
Monitor Process (p_i)
Network
Monitored Process (p_j)
heartbeat!!
heartbeat!!

Fig. 1. General View of the Autonomic Failure Detector

The autonomic manager executes three basic tasks: (i) computing environment and detection service sensing (ii) timeout regulation and (iii) monitoring period regulation. These tasks are described in the following subsections.

4.1 Computing Environment and Detection Service Sensing

Computing Environment Sensing. In distributed systems, when a process (p_i) receives a heartbeat from a process (p_j) it cannot know if p_j is still working. This happens because the receiving of a heartbeat only carries past information about the state of p_j. Thus, if p_i receives a heartbeat at instant r_k it knows that p_j was working until $r_k - rtt_k/2$, where $rtt_k = r_k - s_k$ is the round-trip-time delay of the monitoring messages and $rtt_k/2$ is an estimate for the transmission delay for *hb*. Thus, the greater the interval between the arrival of heartbeat messages the greater the duration of the uncertainty of p_i about the state of the p_j. We can formalize such uncertainty in the following way: at a known instant t, the time interval which p_i is unaware of the state of p_j, which we name *uncertainty time interval (uti)*, can be computed by $uti(t) = t - (r_u - rtt_u/2)$, where r_u and rtt_u represent the time instant of the last heartbeat received and the last round-trip-time computed, respectively. If uti is measured when "are you alive?" aya_k is being sent then p_i will compute uti at each interval k by:

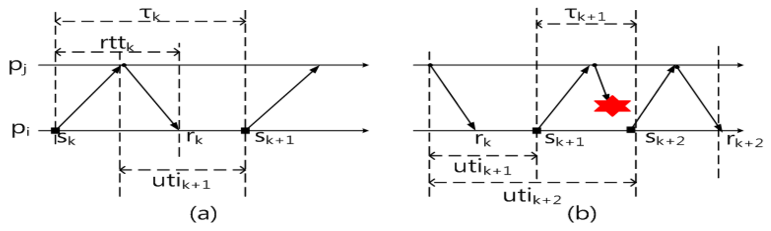

Fig. 2. *uti*: (a) without message losses; (b) with message loss in interval $k+1$

$uti_k = s_k - (r_u - rtt_u/2)$. Figure 2 illustrates the *uti* for (a) the case where messages are not lost and (b) the case when heartbeats are lost.

In order to represent the interaction delay from a monitored process to a monitor, we define the variable *delay* based on the last *uti* estimated. That is, $delay_k = |uti_k - \tau_{k-1}|$, where $\tau_k = s_{k+1} - s_k$ represents the monitoring period. Such a *delay* variable will be used later to calculate the monitoring period related to a given system load and a specified QoS (see section 4.3). If a monitored process p_j does not fail and messages are not lost then $|uti_k - \tau_{k-1}| = rtt_k/2$ (or $delay_k = rtt_k/2$), otherwise *delay* will proportionally increase to τ times the number of heartbeats which was not received by p_i since r_u.

We denote the minimum, maximum, maximum variation delays observed during the failure detector execution as $delay^L$, $delay^U$ and $jitter^U$, respectively. When the characteristics of the computing environment change, it is possible that the observed $delay^L$, $delay^U$ and $jitter^U$ change too. Thus, we consider a forgetting factor f in the computation of these variables, which is defined as: $f_k = max\{0, (TD^U - delay^L_k)\}/TD^U$, where $f_0 = 1$. If $TD^U \approx delay^L$ then $f \to 0$ and the autonomic manager does not take into account the values of these variables. On the other hand, if $TD^U >> delay^L$ then $f \to 1$ and the old values of these variables have the most impact on computing the new values. The algorithm below shows the steps to calculate the delay related variables.

1. Compute $delay_k = |uti_k - \tau_k|$;
2. If $k = 0$ then define $delay^L_k = delay_k$, $delay^U_k = delay_k$ and $jitter_k = 0$;
3. If $delay^L_k > delay_k$ then $delay^L_k = delay_k$, otherwise do $delay^L_k = f * delay^L_{k-1} + (1-f) * delay_k$;
4. If $delay^U_k < delay_k$ then $delay^U_k = delay_k$, otherwise do $delay^U_k = f * delay^U_{k-1} + (1-f) * delay_k$
5. Compute $jitter_k = |delay_k - delay^L_k|$, if $jitter^U_k < jitter_k$ then $jitter^U_k = jitter_k$ otherwise do $jitter^U_k = f * jitter^U_{k-1} + (1-f) * jitter_k$;

In order to handle occasional delay variations, the autonomic manager utilizes a filtered version of delay using the same forgetting factor f. Let $delay^F$ be the filtered version of *delay*, we compute: $delay^F_k = f * delay^F_{k-1} + (1-f) * delay_k$, with $delay^F_0 = delay_0$. The variables *delay*, $delay^L$, $delay^U$, and $delay^F$ represent past information obtained from received heartbeat messages. In order to predict current or expected values of delay (denoted $delay^E$), a safety margin (the *jitter*) is added to the filtered version of delay: $delay^E_k = delay^F_k + jitter^U_k$. Such $delay^E$ is then used in the monitoring period regulation mechanism (see section 4.3).

Detection Service Sensing. We compute the detection time considering the worst case – i.e., when the monitored process p_j fails immediately after it has sent a heartbeat. Thus, the autonomic manager on a monitor process p_i estimates that p_j supposedly crashed at instant $tcrash_k = r_{k-1} - rtt_{k-1}/2$, and p_i suspects p_j when it does not receive the heartbeat hb_k at $tsuspect_k = s_k + rto_k$ (where rto_k denotes the estimate of the timeout for receiving hb_k - see section 3). Consequently, TD can be computed by: $TD_k = tsuspect_k - tcrash_k$. For each monitoring period, we compute the number of false suspicions (nf) and the suspicion duration (sd) as follows. When a suspicion occurs, $nf_k = nf_{k-1}+1$ and $sd_k = r_k - (s_u + rto_u)$, where s_u represents the sending time of aya_u for which a corresponding hb_u was not received in the timeout interval (rto_u). Otherwise, if no suspicions occur, $nf_k = nf_{k-1}$ and $sd_k = 0$. Thus, we can compute the mean mistake duration and the mean mistake recurrence time by $TM_k = \sum_{l=0}^{k} sd_l/nf_k$ and $TMR_k = s_k/nf_k$, respectively. We compute the percentage of mistakes (PoM) and the detection service availability (AV) by: $PoM_k = nf_k/ne$ and $AV_k = (TMR_k - TM_k)/TMR_k$, where $ne = k + 1$ is the total number of estimations carried out by a monitor process.

4.2 Timeout Regulation

Failure detection for asynchronous distributed systems [2–6] requires timeout estimators which have: (a) fast convergence in such a way that it quickly learns the variations on the delay magnitude; (b) high accuracy so that it can meet the application requirements in terms of the detection time; and (c) an over biased estimate of the delay magnitude in order to prevent detection mistakes. These requirements are very hard to achieve because optimizing a criteria may have a negative impact on another one. If the characteristics of the environment change often, then it is a good idea to have a timeout estimator with a high learning rate and high accuracy. However, if short and spurious changes in environment behavior are considered an estimator with fast learning rate probably make mistakes. Nonetheless, over biased estimates is a good strategy to prevent mistakes, but overestimating of the detection timeout can lead to slow detections, compromising application responsiveness. To address fast convergence, high accuracy and minimum over biased estimate, we consider timeout estimators based on end-to-end delay observation as used in traditional literature of failure detection for asynchronous distributed systems. We then designed a novel strategy based on detection availability (AV) to suggest a safety margin (α) in such a way to decrease failure detector mistakes and to achieve the desired detection availability, as follows. If the detection service is inaccurate (i.e., AV is low), then the safety margin α is increased to improve detection accuracy; otherwise, if AV is high, then α is decreased to improve the detection speed. For a interval k, let AV^L and AV_k be the minimum and the observed detector availability, respectively, and let $\alpha_0 = 0$, the detection timeout is regulated by the following algorithm.

1. Compute $AV^L = (TMR^L - TM^U)/TMR^L$ and $AV_k = (TMR_k - TM_k)/TMR_k$;
2. Calculate $e_k = AV^L - AV_k$ and $\alpha_k = \alpha_{k-1} + \tau_{k-1} * e_k$, If $\alpha_k < 0$ then $\alpha_k = 0$;
3. Use a previously selected timeout estimator of the literature to suggest the timeout rto_k^C and so compute the detection timeout by $rto_k = rto_k^C + min[\alpha_k, TD^U - delay^L]$.

Any timeout estimator based on delay observation which considers the requirements discussed above can be used with our novel safety margin strategy.

4.3 Monitoring Period Regulation

The goal of the monitoring period regulation is to minimize the detection time without compromising the accuracy of the detector. To attain such an objective, we implement a feedback control loop embedded in the autonomic manager. This control loop carries out three activities: (i) characterization of the computing environment resource consumption (ii) definition of the control problem and (iii) design and tuning of the controller. Because we assume that the environment characteristics change and are unknown, we treat the computing environment as a black-box system and use *delay* variations to help us to estimate the resource consumption. In despite of unpredictable variations of *delay*, we use linear equations to model resource consumption and to estimate the relationship between *delay* and the monitoring period. These models based on linear equations are an approximation and do not appropriately address the problem of characterizing the environment behavior. Nonetheless, they are a good tool for helping us to describe the control problem issues, to define the desired dynamic behavior of the distributed environment and to design the control law. To overcome the limitations of the linear approach we designed an adaptation law which dynamically tunes the parameters of the model, thus enabling the considered model to self-adjust given the changes in the computing environment. Each one of the considered activities for the implementation of the period regulation control loop is described in greater details below. These activities result in the algorithm presented at the end of this section.

Characterization of the Computing Environment. As previously mentioned we abstract the environment as a Black-Box System (BBS). Thus, a monitor process submits a service request (i.e. a "are you alive" message) and receives a service response (i.e. a "heartbeat" message with the state of a monitored process) from the BBS. For the sake of estimating the execution BBS capability, we assume the minimum *delay* ($delay^L$) as an indication of BBS' service time. This service time is later used as a reference to estimate resource consumption in the system. Similarly, in this black-box modeling we take the expected delay $delay^E$ as an indication of the BBS' response time - such a response time will vary with the number of the service requests. The resource consumption (rc) can be estimated based on *delay* variations as:

$$rc_k = \begin{cases} 0 & if \;\; delay^U = delay^L \\ \dfrac{delay^E_k - delay^L}{delay^U - delay^L} & otherwise \end{cases} \tag{1}$$

Resource consumption can also be modeled as a function of the monitoring period. As such, shorter period will likely lead to more resource consumption. We use this fact later to design the plant model (i.e., the BBS behavior) by correlating both views of resource consumption : observed from *delay* variations and expected from the monitoring period.

We use the following equation to model resource consumption as a function of the monitoring period: $rc_k = (\tau^U - \tau_{k-1})/(\tau^U - \tau^L)$ where $\tau^U = TD^U - delay^L$ and $\tau^L = delay^L$ represent the maximum and the minimum monitoring period, respectively. Relating this equation with equation 1 we can compute $delay^E_k = [(\tau^U - \tau_{k-1})/(\tau^U - \tau^L)] * (delay^U - delay^L) + delay^L$. Calculating the derivative of $delay^E$ in function of τ, we have: $\upsilon = -\Delta delay^E_k/\Delta\tau_{k-1} = (delay^U - delay^L)/(\tau^U - \tau^L)$ where $\Delta x_i = x_i - x_{i-1}$.

With respect to a monitor p_i, if we just observe the behavior of the delay variation in BBS with respect to the variation of τ, then it will be possible to compute the relationship between the input ($u = \Delta\tau$) and output ($y = \Delta delay^E$) of BBS using a linear ARX model, a commonly applied technique [17]:

$$y_{k+1} = y_k + \upsilon * u_k \tag{2}$$

Feedback Control Problem. The control problem is to regulate τ to obtain lower detection times using available resources. We name the maximum resource consumption with respect to p_i as $rc^U \in (0, 1]$. Therefore, we compute the difference between rc^U and the resource consumption rc_k (equation 1) by: $e_k = rc^U - rc_k$. When $e > 0$, rc is lower than expected, so τ is decreased to enable a fast detection. When $e < 0$, rc is greater than expected so τ is increased to consume less resources and to avoid failure detector mistakes.

Design and Tuning of the Controller Period regulation is carried out using a Proportional-Integral (PI) controller[17]. This controller produces a control action (the required variation on τ) considering the difference (e_k) between rc^U and rc_k. A PI controller considers the following control law:

$$u_k = K_P * e_k + K_I * \Delta t * \sum_{i=0}^{k-1} e_i \tag{3}$$

where Δt is the time interval between the last and the current activation of the controller and K_P and K_I are the proportional and integral gains of the controller, respectively [17]. The tuning of the controller entails finding the values of the gains K_P and K_I which have to address the following performance requirements [17]: stability; accuracy; settling time (K_s); and maximum overshoot (M_p). As previously discussed, if the environment characteristics change then these performance requirements will not be guaranteed with the linear controller (PI). Additionally, the design of such linear controller requires the choice of constant time intervals, named sampling period (h), in which the controller observes the resource consumption and actuates. The sampling period depends on the magnitudes of the environment delays which are unpredictable so the sample period cannot be constant either. Moreover, traditional design based on the z-transform transfer functions (see [17]) is a valid tool only if sampling period is constant so, in principle, we could not make use of z-transform also. To address these limitations, we consider the initial setup of the PI controller, using z-transform transfer functions, and use an adaptation law for the gains K_P and K_I.

Our initial setup uses Equations 2 and 3 to obtain the z-transform transfer functions $P(z) = \frac{\upsilon}{z-1}$ and $C(z) = K_P + K_I * \Delta t * \frac{z}{z-1}$ which represent the BBS behavior and the PI control law, respectively. We then use $P(z)$ and $C(z)$ to define the closed loop transfer function $F_r(z) = \frac{C(z)*P(z)}{1+C(z)*P(z)}$. With these definitions, we assume that the closed loop reaches the desired output in a settling time $K_s = delay^U$. Moreover, under moderate load conditions (i.e. mean delay greater than standard deviation), we define the maximum output overshoot as 10% (i.e. $M_p = 0.1$). Thus, we use the *pole placement technique*[1] to estimate the complex poles (cp) of the $F_r(z)$ as: $cp = m * exp(\pm j\theta)$, where $m = exp(-4/K_s)$ and $\theta = \pi * log(m)/log(M_p)$ are the magnitude and angle of the poles cp, respectively. Thus, we define K_P and K_I using the following adaptation law:

1. verify if $delay^L_k = delay^U_k$ then $\phi = 0$ else $\phi = 1/(delay^U_k - delay^L_k)$;
2. compute $\psi = 1/(\tau^U_k - \tau^L_k)$, $K_P = (\phi - m^2)/\psi$ and $K_I = (m^2 - 2 * m * cos(\theta) + 1)/\psi$

As previously discussed, $delay^L$ and $delay^U$ change over time so the closed loop poles cp change also. The proposed adaptation law adjusts K_P and K_I so as to handle these variations and to improve the control loop performance.

Period Regulation Algorithm. The algorithm below, used to regulate the monitoring period, follows from the previous discussion.

1. if $k = 0$ then do $ui_0 = 0$;
2. obtain $e_k = rc^U - rc_k$, adapt K_P and K_I, compute the integral control action $ui_k = ui_{k-1} + e_k * K_I * \Delta t$ and compute the proportional control action $up_k = K_P * e_k$;
3. define $\tau^L_k = delay^L_k$ and $\tau^U_k = TD^U - delay^L_k$ and obtain $\tau_k = \tau^L_k + (ui_k + up_k)$, if $\tau_k > \tau^U_k$ then $\tau_k = \tau^U_k$ else if $\tau_k < \tau^L_k$ then $\tau_k = \tau^L_k$;

4.4 Performance Evaluation

Setup of the Simulations. The simulations were carried out in Matlab / Simulink / TrueTime[18]. The simulated environment has three computers named c_1, c_2 and c_3 that are connected by a Switched Ethernet with a nominal transfer rate of $10Mbps$ and memory buffer size of $8Mbits$. Messages have a fixed size of $1518bits$ and when a *buffer overflow* occurs the messages are discarded. A process in c_1 monitors failures of a process in c_2. A process in c_3 generates a random burst of messages in the network. This burst is generated in such a way that the mean utilization of the bandwidth is increased by 10% at each $1000ms$ and this utilization returns to zero after reaching 90%. The burst generation allows us to evaluate the detectors under different load conditions. The simulation is executed until we have transferred 10^4 monitoring messages (or $40000ms$), approximately. The experiments compare the performance of our autonomic detector (AFD) with an adaptive detector (AD), both using the Jacobson algorithm[19] for timeout prediction. In the performance evaluation we manually configured AD with $\tau = 1ms$ and $\tau = 5ms$. We configured AFD as follows: $rc^U = 0.5$, $TD^U = 50ms$, $TM^U = 1ms$ and $TMR^L = 10000ms$. The failure detectors were evaluated by TD, TM, TMR, AV and PoM metrics.

[1] see [17] for a discussion

Simulation Results. Figures from 3 to 7 present the performance of the failure detectors in terms of the considered performance metrics, for varying network loads. In these figures, the x-axis and y-axis of the graphics represent the time in milliseconds and the metric which is being considered, respectively. In Figures 3, 4 and 6, TD, TM and the TMR are represented in milliseconds.

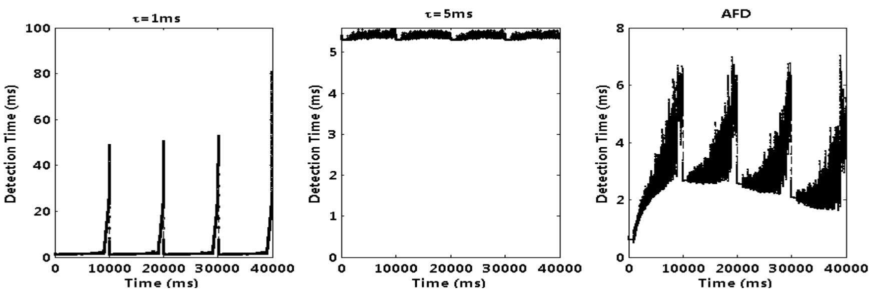

Fig. 3. Performance in terms of the Detection Time

In terms of the TD metric (Figure 3), the AD with $\tau = 1ms$ presents the lowest TD under low network load conditions but TD under high network load conditions increases significantly (the highest values of the curve). The AD with $\tau = 5ms$ presents mean TD equal to $5.5ms$. The AFD varies TD with the variation of the network load but, in this case, the observed TD is always lower than $7ms$ and its mean is equal to $5ms$.

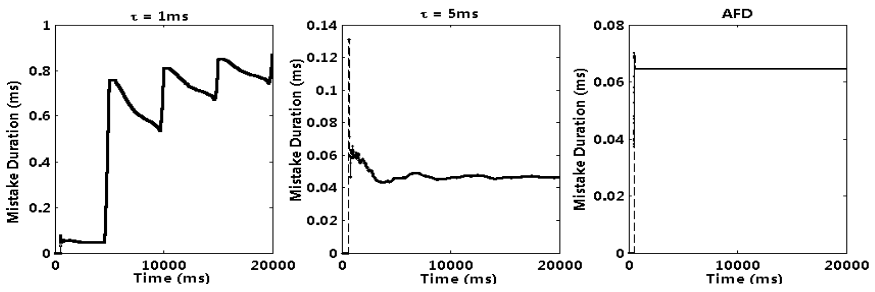

Fig. 4. Performance in terms of the Mistake Duration

In terms of the TM metric (Figure 4), the AD with $\tau = 1ms$ presents mistake durations around $10^{-1}ms$ and has mean equal to $0.70ms$. the AD with $\tau = 5ms$ and the AFD have TM around $10^{-2}ms$ where the mean mistake durations are $0.05ms$ and $0.06ms$, respectively. However, the TM showed by AFD are always lower than 0.07 and vary less than in AD.

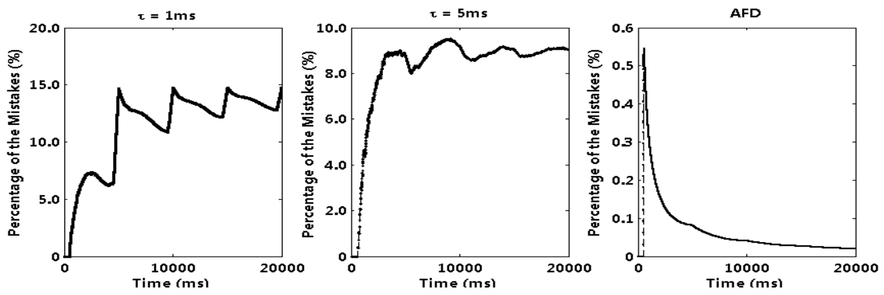

Fig. 5. Performance in terms of the Percentage of Mistake

In terms of the *PoM* metric (Figure 5), the AD presents mean *PoM* equal to 12.0% and 9.0% for with $\tau = 1ms$ and $\tau = 5ms$, respectively. The AFD always presents a *PoM* lower than 0.6% and has a mean *PoM* equal to 0.1%. Such better accuracy performance of AFD is due to its safety margin, and the use of longer periods for high network loads - thus contributing to more stable delays.

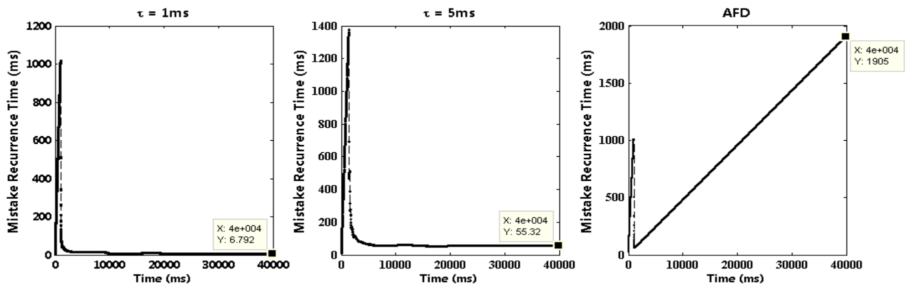

Fig. 6. Performance in terms of the Mistake Recurrence Time

In terms of the TMR metric (Figure 6), the AD presents an initial peak and stabilizes around $6.8ms$ and $55.3ms$ for $\tau = 1ms$ and $\tau = 5ms$, respectively. The AFD has an initial oscillation in TMR but then it increases with time reaching $1900ms$. This better performance of AFD is also due to its safety margin and longer periods for high network loads.

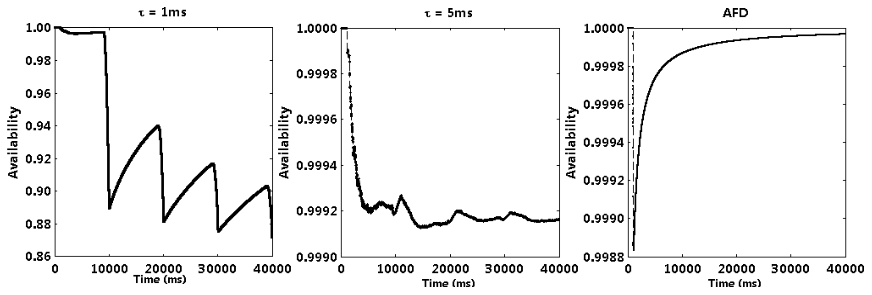

Fig. 7. Performance in terms of the Detection Availability

Lastly, in terms of the AV metric (Figure 7), the AD with $\tau = 1ms$ presents a mean AV around 0.9 but it shows very oscillatory behavior. The AD with $\tau = 5ms$ has an AV around 0.999. The AFD has an AV around 0.9999 and varies less than AD. Observe that AV is directly derived from TM and TMR, thus the better performance of AFD - which also performed better for TM and TMR. From the performance data presented above we can conclude that AFD has better performance in terms of detection time, mistake recurrence time, percentage of mistakes and availability, and it has a performance similar to AD in terms of mistake duration. We carried out similar experiments considering the comparison between AFD and another AD (using the Bertier approach [3]) and AFD also performed better. These results are available in a technical report at LaSiD website (http://www.lasid.ufba.br) and are omitted here due to space constraints.

5 Final Remarks

Traditional failure detection approaches for distributed systems do not support the self-configuring of the failure detector by QoS metrics. However, when the computational environment characteristics are unknown and can change, self-configuring is a basic ability to guarantee the tradeoff between response time and availability. A self-configuring ability requires the modeling of the dynamic behavior of the distributed system which is a great challenge when the computing environment can change. To address these challenges, this paper presented the design and implementation of a novel detector based on control theory and which is able to dynamically configure the monitoring period and detection timeout following the observe changes in the computing environment and according to user-defined QoS constraints. We developed a series of experiments in a simulated environment to verify the performance of the autonomic failure detector (AFD) in terms of the speed and accuracy. These evaluations allowed us to observe how fast and how accurate was the detector in different scenarios of network loads. Even without similar works for a direct comparison, the evaluations considered comparisons with different configurations of an adaptive failure detector (AD) available in literature and manually configured with different monitoring periods. The simulations demonstrated that the AFD indeed could dynamically regulate the monitoring period to achieve the desired QoS and, in most cases, our approach performed better than the AD considered.

Since the primary goal of our research is to provide a mechanism to dynamically adjust the monitoring period, AFD was designed to encapsulate any available AD not just the one used in the implementation and evaluation presented. This makes our solution general enough to take advantage of any other AD with better performance.

Finally, the evaluations have considered only local networks which are usual in cluster computing environments. In future works, we are going to evaluate our approach in WAN scenarios and apply it to other mechanisms related to the management of autonomic applications under development at LaSiD [20].

References

1. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal Of The ACM **43**(2) (mar 1996) 225–267
2. Chen, W., Toueg, S., Aguilera, M.K.: On the quality of service of failure detectores. IEEE Trans. On Computer **51**(2) (may 2002) 561–580
3. Bertier, M., Marin, O., Sens, P.: Performance analysis of hierarchical failure detector. In: Proc. Of Int. Conf. On DSN, USA, IEEE Society (Jun 2003) 635–644
4. Lima, F.R.L., Macêdo, R.J.A.: Adapting failure detectors to communication network load fluctuations using snmp and artificial neural nets. In: Second Latin-American Symposium on Dependable Computing (LADC2005), Lecture Notes in Computer Science, Volume 3747, Pages 191 - 205. (Oct 2005)
5. Nunes, R.C., Jansch-Pôrto, I.: Qos of timeout-based self-tuned failure detectors: the effects of the communication delay predictor and the safety margin. In: International Conf. On DSN, IEEE Computer Society (Jul 2004) 753–761
6. Falai, L., Bondavalli, A.: Experimental evaluation of the qos failure detectors on wide area network. In: Proc. of Int. Conf. on DSN, IEEE CS (Jul 2005) 624–633
7. Xiong, N., Yang, Y., Chen, J., He, Y.: On the quality of service of failure detectors based on control theory. In: 20th Int. Conf. on Advanced Information Networking and Applications. Volume 1. (Apr 2006)
8. Satzger, B., Pietzowski, A., Trumler, W., Ungerer, T.: A lazy monitoring approach for heartbeat-style failure detectors. In: 3rd Int. Conf. on Availability, Reliability and Security, 2008 (ARES2008). (Mar 2008) 404–409
9. Mills, K., Rose, S., Quirolgico, S., Britton, M., Tan, C.: An autonomic failure-detection algorithm. In: WOSP '04: Proc. of the 4th int. workshop on Software and Performance, New York, USA, ACM (2004) 79–83
10. IBM: Autonomic computing: Ibm's perspective on the state of information technology. Technical report, IBM Coorporation, USA, New York (2001)
11. Ogata, K.: Discrete-Time Control Systems. 2nd edn. Prentice-Hall, Upper Saddle River, NJ 07458, USA (1995)
12. Cristian, F.: Understanding fault-tolerant distributed systems. Communication of the ACM **34**(2) (1991) 56–78
13. Birman, K.: Replication and fault-tolerance in the ISIS system. ACM SIGOPS Operating Systems Review **19**(5) (1985) 79–86
14. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2) (1985) 374–382
15. So, K.C.W., Sirer, E.G.: Latency and bandwidth-minimizing failure detectors. In: EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, New York, NY, USA, ACM (2007) 89–99
16. Felber, P.: The Corba Object Group Service : A Service Approach to Object Groups in CORBA. PhD thesis, École Polytechnique Fédérale De Lausanne (1998)
17. Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D.M.: Feedback Control of Computing Systems. Wiley-Interscience, Canada (2004)
18. Henriksson, D., Cervin, A.: Truetime 1.13 - reference manual. Tech. Report TFRT-7605-SE, Dep. Of Automatic Control, Lund Institute Of Technology (Oct 2003)
19. Jacobson, V.: Congestion avoidance and control. ACM CC Review; Proc. Of The Sigcomm '88 Symp. In Stanford, Ca, August, 1988 **18, 4** (1988) 314–329
20. Andrade, S.S., Macêdo, R.J.A.: A non-intrusive component-based approach for deploying unanticipated self-management behaviour. In: Proc. of IEEE ICSE 2009 W. Software Engineering for Adaptive and Self-Managing Systems. (May 2009)