Performance Analysis of a Hierarchical Failure Detector

Marin BERTIER[2]
marin.bertier@lip6.fr
[1]Laboratoire d'Informatique du Havre
University of Le Havre
25 rue Philippe Lebon
BP540 76058 Le Havre cedex, France

Olivier MARIN[1,2]
olivier.marin@univ-lehavre.fr
[2]Laboratoire d'Informatique de Paris 6
University Paris 6 - CNRS
4 place Jussieu
75252 Paris Cedex 05, France

Pierre SENS[3]
pierre.sens@inria.fr
[3]INRIA
Domaine de Voluceau
BP 105
78153 Le Chesnay, France

Abstract

We present a new failure detector implementation. This implementation, a variant of the heartbeat failure detector, is both adaptable and designed for scalability. Its first specificity lies in the fact that it is designed as a shared service among several applications by way of an adaptation layer. This layer adapts the quality of service according to application needs. The second specificity is the hierarchic organization of the detection service: it allows to decrease the number of messages and the processor load. Through an experimentation evaluation, we show that our implementation is adaptable to the environment characteristics and usable with large scale applications.

1 Introduction

The consensus problem is considered as a basic building block for distributed systems. Fisher, Lynch and Paterson [6] have shown that the consensus cannot be solved deterministically in an asynchronous system that is subject to even a single crash failure, because it is impossible to determine whether a process has actually crashed or is only "very slow". To circumvent this impossibility Chandra and Toueg introduce, *the unreliable failure detector* [3]. A failure detector can be considered as an oracle per process. Such an oracle provides the set of processes that it currently suspects of having crashed.

Many applications use a failure detection service in order to solve the consensus problem [13, 5]. Typically, applications use messages sent periodically between hosts [9, 12, 5]. The problem of this approach is the fast increase in cost as detection messages are sent by all applications. The aim of our approach is to design a shared and moreover scalable detection service between several applications. We present a performance evaluation to show that our detection layer provides a suitable quality of service with respect to application needs while ensuring low costs for the system.

Our implementation is based on a basic failure detection layer introduced in [2]. This layer is a variant of the heartbeat detector: it adapts the sending period of *"I am alive"* messages as a function of the network quality of service and of the application requirements.

For each application, an adaptor is plugged onto the failure detector layer. Its aim is to adapt the quality of service provided by the basic layer according to application needs. This architecture guarantees the required quality of service and avoids network overload. To reduce the overall amount of exchanged messages, we organize the detectors hierarchically [8]. Moreover, the constant flow of communications produced by the failure detection service can be exploited to piggy-back application messages and data.

The rest of this paper is organized as follows. Section 2 describes our system model. In Section 3, we briefly present the basic layer of our failure detector and Section 4 presents the adaptation layer. In Section 5, we present the architecture of our detection service. Section 6 presents the performance evaluation of this service. Finally, we conclude in Section 7.

2 Context and Model

2.1 Models and Environment

Let us consider a distributed system consisting of a finite set of n processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ that are spread throughout a network. These processes communicate only by sending and receiving messages.

We rely on the model of partial synchrony proposed by Chandra and Toueg in [3]. This model stipulates that, for every execution, there are bounds on process speeds and on message transmission times. However, these bounds are not known and they only hold after some unknown time (called *GST* for *Global Stabilization Time*). We denote by Δ_{msg}

the maximum time, after *GST*, between the sending of a message and the delivery and processing by its destination process, assuming that the destination process has not failed and the message is not lost.

We have experimented our implementation in a "real-life" environment: our laboratory network. In a second phase, we analyze the behavior of our implementation when the model constraints are weakened. We assume that processes can fail by crashing only. Our algorithm however does not need synchronized clocks, but there must exist a known upper bound on the drift rate of local clocks. We assume that the network supports IP-Multicast communication and is quasi-reliable: if process p sends an infinite number of messages m to process q, then q receives at least one message m. We assume that a host may "*recover* " after crashing. However application behavior in case of failure/recovery does not bring any interesting information in the scope of this paper. We only guarantee the correct restart of the detection service.

All processes are assumed to have a preliminary knowledge of the system organization. More precisely, a process knows its local group composition as well as the broadcast addresses of all the other groups.

2.2 Application Context

The failure detector implementation presented in this article is part of the DARX (Dynamic Agent Replication eXtension) project [10]. The goal of this project is to provide a framework for designing large-scale agent systems. Some systems may comprise thousands of agents; they are often developed in the field of distributed artificial intelligence. Many multi-agent platforms [1, 14] propose solutions to deploy agent applications over networks. However, to our knowledge, no platform provides the required characteristics in terms of fault tolerance for massive agent organizations running over asynchronous systems such as the world wide web.

To supply adequate support for large-scale agent applications, the DARX platform includes a hierarchical, fault-tolerant naming service. To provide a synchronous abstraction of the underlying network, this distributed service is mapped upon the failure detection service through the adaptation layer presented in Section 4.

3 The basic layer of failure detector

3.1 Unreliable failure detector

The aim of failure detectors is to provide information about the liveness of other processes. Each process has access to a local failure detector which maintains a list of processes that it currently suspects of having crashed. Since a failure detector is unreliable, it may erroneously add to its list a process which is still running. But if the detector later believes that suspecting this process is a mistake, it then removes the process from its list. Therefore a detector may repeatedly add and remove a same process from its list of suspect processes. Failure detectors are characterized by two properties: completeness and accuracy. Completeness characterizes the failure detector capability of suspecting every incorrect process permanently. Accuracy characterizes the failure detector capability of not suspecting correct processes. Two kinds of completeness and four kinds of accuracy are defined in [3], which once combined yield eight classes of failure detectors.

We focus on the $\Diamond P$ detector, named *Eventually Perfect*. This detector requires the following characteristics:

- **Strong completeness**: there is a time after which every process that crashes is permanently suspected by every correct process.
- **Eventual strong accuracy**: there is a time after which correct processes are not suspected by any correct process.

The demonstration found in [2] proves that the above properties hold, provided that a suitable adaptor is used. In parallel, [4] proposes a set of metrics that can be used to specify the Quality of Service (QoS) of a failure detector (see Figure 1). The QoS quantifies how fast a detector suspects a failure and how well it avoids false detection.

- *Detection time* (T_D): T_D is the time that elapses from p's crash to the time when q starts suspecting p **permanently**.
  The next metrics are used to specify the accuracy of a failure detector.
- *Mistake recurrence time* (T_{MR}): this measures the time between two consecutive mistakes.
- *Mistake duration* (T_M): this measures the time taken by the failure detector to correct a mistake.

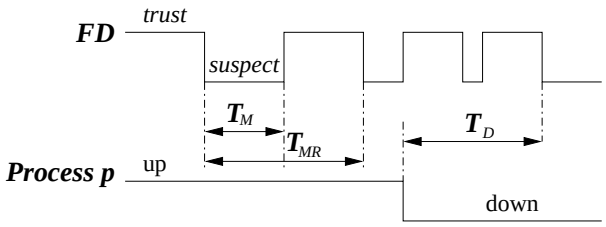

Figure 1. different metrics to specify quality of detection

3.2 Our implementation

In our implementation, we dissociate two layers: a basic layer, which provides a good detection time, and an adaptation layer, which adapts the QoS of the detection as well

as the interface provided by the basic layer. The adaptation layer is detailed in Section 4.

The basic layer uses the heartbeat strategy presented in [4]: every process p periodically (every Δ_i time units) sends an *"I am alive"* message $m_1, m_2, \ldots$ to the processes in charge of detecting its failure. To determine whether to suspect p, q uses a sequence $\tau_1, \tau_2, \ldots$ of fixed time units, called *freshness points*. The freshness point τ_i is an estimation of the arrival date of the i^{th} heartbeat message from p. If q later receives an *"I am alive"* message from p, then q removes p from its list of suspected processes (see Figure 2).

In our implementation τ_i is composed of an expected arrival date, called EA and a safety margin α. EA consists of an average of the η last arrival dates. EA provides for shorter detection times even though the probability of false detections may be increased.

Each process q considers the η most recent heartbeat messages, denoted $m_1, m_2, \ldots, m_\eta$. Let $A_1, A_2, \ldots, A_\eta$ be their receipt times according to q's local clock. When at least η messages have been received, $EA_{(k+1)}$ can be estimated by:

$$EA_{(k+1)} = \frac{1}{\eta} \sum_{i=k-\eta}^{k} \left(A_{(i)} - \Delta_i * i \right) + (k+1).\Delta_i$$

The safety margin $\alpha_{(k+1)}$ is calculated similarly to Jacobson's [7] estimation. It adapts the safety margin each time it receives a message according to network load. The adaptation of the margin α uses the $error$ in the last estimation. Parameter γ represents the importance of the new measure with respect to the previous ones. $delay$ represents the estimate margin, and var estimates the magnitude between errors. β and ϕ enable to weigh the variance; typical values are $\beta = 1$ and $\phi = 4$. The original algorithm is:

$$\begin{aligned} error_{(k)} &= A_k - EA_{(k)} - delay_{(k)} \\ delay_{(k+1)} &= delay_{(k)} + \gamma.error_{(k)} \\ var_{(k+1)} &= var_{(k)} + \gamma.(|error_{(k)}| - var_{(k)}) \\ \alpha_{(k+1)} &= \beta.delay_{(k+1)} + \phi.var_{(k+1)} \end{aligned}$$

The next timeout $\Delta_{to_{(k+1)}}$, activated by q when it receives m_k, expires at the next freshness point:

$$\tau_{(k+1)} = EA_{(k+1)} + \alpha_{(k+1)}$$

A precise presentation of our estimation method is given in [2]. It is shown to be a useful compromise between a short detection time and the need to avoid false detections.

Each adaptor proposes an emission interval Δ_i which satisfies the application specified quality of service required by application. This computation method is presented in Section 4. Then the basic layer uses the smallest interval required. But if it detects that the network load can't withstand this interval, it changes Δ_i with respect to the network load. The basic layer tries to provide the shortest detection time with respect to the emission interval Δ_i. Hence the quality of the detection provided by the first layer is wholly derived from the emission interval and the safety margin α. The adaptor can consult these parameters in order to adapt the quality of service for every application.

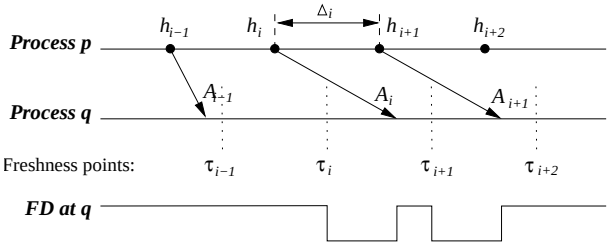

Figure 2. Failure detection with the heartbeat strategy

The second service provided by the basic layer is the relaying of information by piggy-backing on *"I am alive"* messages. In fact the initial role of heartbeat messages is solely to signal that the sender is still alive. We propose to piggy-back application information to reduce network overload. A message loss will be noticed by the receiving failure detector; piggy-backed retransmission might therefore be requested. This mechanism tolerates network failures, but the recovery delay may be too long to tolerate a high loss rate. These two points are detailed further in Section 4.

4 Adaptation layer

The adaptation layer adjusts the shared service provided by the basic layer to the specific requirements of every application. Hence this layer is included between the basic failure detection layer presented in the previous section and the user application.

Our adaptor provides higher-level algorithms to enhance the characteristics of the failure detectors.

Firstly, it adapts the quality of service provided by the basic layer to the application needs. In fact the aim of the first layer is to provide the shortest detection time. The adaptor can only delay the moment when a process is suspected as having crashed. The main advantage of this architecture is that we make no assumption on the adaptation algorithm: it can be different for each application. Using several adaptors on the same host allows to obtain different visions of the system. The basic layer maintains a *blackboard* to provide information to the adaptation layers. In this *blackboard* it publishes information about:

- the list of suspects,
- the current emission interval Δ_i,
- the current safety margin α,

- and system observation information.

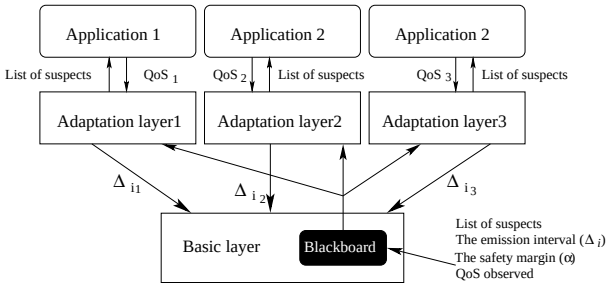

Figure 3. Quality of service

As seen in Figure 3, each adaptor informs the basic layer of the emission interval (Δ_i) required with respect to the quality of service they must provide. The basic layer selects the smallest interval required as long as the network load allows for it. Every application must provide to its adaptation layer the detection quality of service it requires. The scope of this detection quality of service is limited to the local group. It is expressed by means of the variable presented in Section 3.1: the upper bound on detection time (T_D^U), the lower bound on average mistake recurrence time (T_{MR}^L) and the upper bound on average mistake duration (T_M^U). The network characteristics, the message loss probability (P_L) and variance of message delays (V_D), are provided by the basic layer. The adaptation procedure is adapted from [4]:

- *Step 1*: Compute $\gamma = \frac{(1-P_L)(T_D^U)^2}{V_D+(T_D^U)^2}$ and let $\Delta_{i_{max}} = \max(\gamma.T_M^U, T_D^U)$.
  If $\Delta_{i_{max}} = 0$, then the QoS cannot be achieved
- *Step 2*: Let

$$f(\Delta_i) = \Delta_i . \prod_{j=1}^{[T_D^U/\Delta_i]} \frac{V_D+(T_D^U-j\Delta_i)^2}{V_D+p_L(T_D^U-j\Delta_i)^2}$$

  Find the largest $\Delta_i \leq \Delta_{i_{max}}$ such that $f(\Delta_i) \geq T_{MR}^L$.
- *Step 3*: Set the safety margin $\alpha = T_D^U - \Delta_i$.

From this information, the adaptation layer can alter the detection of the basic layer and change its emission interval so as to adjust the quality of service provided by the basic layer.

The adaptation layer can also change the interface of the detector. The basic layer has a push behavior. When it suspects a new process, every adaptor is notified. The interface with any particular application can therefore be altered; for example a pop behavior can be adopted, where the adaptation layer does not send signals to the application but leaves to the application the duty of interrogating the list of suspects. The typical adaptor works as follows (see Figure 4);

Every process $p \in \Pi$ performs :

{ Initially, every process q is suspected by process p }
$suspect_p \leftarrow \Pi - \{p\}$
$wait_p \leftarrow \emptyset$
for all $q \in \Pi - \{p\}$
  $\Delta_p(q) =$ Initial margin evaluation
    { $\Delta_p(q)$ refined detection margin}

q is suspected by FD because message n has not arrived:
  $wait_p \leftarrow wait_p \cup \{q\}$
  $delay =$ Margin evaluation$(\Delta_p(q))$
  wait for $delay$
  if $q \in wait_p$ **then**
    $suspect_p \leftarrow suspect_p \cup \{q\}$
    $wait_p \leftarrow wait_p - \{q\}$
q is no more suspected by FD because message n has arrived
  if $q \in wait_p$ **then**
    $wait_p \leftarrow wait_p - \{q\}$
  else
    $suspect_p \leftarrow suspect_p - \{q\}$
      {it is a false detection}
Periodically
  For all $q \in \Pi - \{p\}$
    $\Delta_p(q) =$ margin evaluation$(QoS_p(q))$
      {$QoS_p(q)$ is provided by failure detector}

Figure 4. The adaptor implementation

initially it suspects every process and initializes the refined margin according to QoS requirements. When the failure detector distrusts a process, it sets a timeout by toning down the margin applied in the first layer. At the expiration of the timeout, the adaptor checks if the process is still suspected by the failure detector. If this is the case, then the adaptor also begins to distrust the same process. If the failure detector stops suspecting a process, then it corrects this mistake. Periodically, the adaptor reevaluates its margin according to the network observation provided by the first layer.

5 Hierarchical architecture

5.1 Presentation

As part of the means to supply adequate support for large-scale applications, the detection service follows a hierarchical organization. It comprises two levels: a local and a global one, mapped upon the network topology.

The system is composed of local groups, mapped upon a LAN, bound together by a global group. Each group is a detection space, which means that every group member watches on all the other members of its group. Every local group elects exactly one leader which will participate to the global group.

For example, in Figure 5, the system is organized in three

local groups, one for each LAN. And these local groups are merged in one global group.

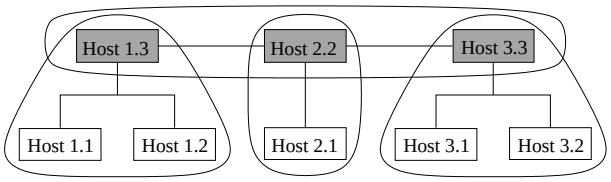

Figure 5. Hierarchical organization

This organization calls for two different failure detector types. This distinction is important since a failure does not have the same interpretation in the local context as in the global one. A local failure corresponds to the crash of a host, whereas in the global context a failure represents the crash of an entire local group. In this situation, the ability to provide different qualities of service to the local and the global detectors is a major asset of our implementation. Therefore a local group leader has two different failure detectors, one for the local group and one for global group

In a local group, the failure detector uses IP-Multicast for sending *"I am alive"* messages. In a LAN, IP-Multicast can be used with the broadcast property. Therefore a host only sends one message to communicate with all the other hosts. Failure detectors in a global group use UDP in order to be more compatible with the general network security policy.

5.2 Advantages and Disadvantages

The hierarchical organization makes it possible for every process to observe only a small part of the network. In a flat system, a process failure is detected by all the other processes, whereas in a hierarchical system, it is only detected by the other group members. We assume in our implementation that it is not essential to ensure the notification of every failure throughout the system. Therefore information propagation is left to the application, likewise to recovery.

In a flat system all of the n hosts send a message to the remaining $n-1$ hosts, whereas by IP-Multicast, each host sends only one message. However, if a message must be relayed to hosts outside the local LAN, then the message is duplicated. Anyway, all hosts will receive such a message. The complexity as a function of the number of messages is $n*(n-1)$.

In a hierarchical system where n hosts are divided equitably in g local groups. In each local group, n/g hosts send $n/g-1$ messages, and in the global group g hosts send $g-1$ messages. The complexity as function of number of messages is $n^2/g+g^2-g-n$. The hierarchical organization is therefore more profitable, without incidence from the number of hosts. Besides the high benefit in terms of the number of messages, the load imposed on leader does not appear too important. The leader load in terms of message processing is $k/n-1+g-1$. When $n > g^2$, the leader load is less important than that of any host in a flat group.

In Section 6, we compare the processor loads induced by the two organizations.

5.3 Implementation

This architecture needs more services than in a flat system, in particular the election of a leader, and the negotiation of the quality of service.

In a local group there must be a unique leader. When a leader crashes, the failure ought to be firstly detected by the local group [1]. To determine a unique local group leader, each group member has a list of all other the members of its group. This list is totally ordered by increasing identifiers. At the beginning the first group member in the list is appointed leader.

When a new leader is appointed, all other global group members must be informed. Since this may be costly, reelections caused by false detections must be avoided as much as possible. For this purpose, the election algorithm takes effect only if at least $(n+1)/2$ group members suspect the current leader.

Every process has a priori knowledge that if the leader is suspected then the next host in the list will become the new leader, provided is trusted.

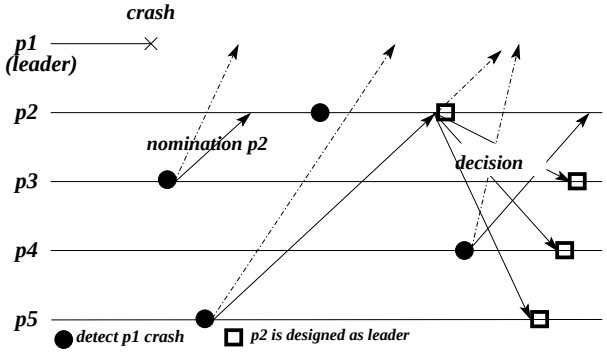

Figure 6. Successful leader election

Typically, this election algorithm works as follows: when a host suspects the leader, it chooses as the new leader the first unsuspected process in its ordered list. It also sends a *nomination* message with the name of this new leader to the current leader and the proposed one. When the proposed leader has received more than $n/2+1$ *nomination messages*, it becomes the new leader and it emits a *decision* message.

[1] Typically, the quality of service in a local group is better than in a global group

When a process receives a *decision* message, it updates the sender as the new leader.

This election algorithm can be applied with very few messages but it can't ensure that there is exactly one leader at a given time. If the old leader is suspected and yet just slow, then when it receives the *nomination* message, it launches another election in an attempt to cancel the current one. However for a short period of time, there are two leaders for the same group. To restrict this problem, as soon as a new leader joins the global group, the members of the global group ignore the messages from the old leader.

The second problem is how to insert a new leader in the global group. Shortly after its election, the new leader does not yet know the other group leaders, hence it can't communicate with them.

To solve this problem, there is a subscription mechanism, which allows a new leader to communicate with other group leaders. Each local group is located in a LAN, therefore a new leader periodically broadcasts a *registration message* to every LAN. When a leader receives a *registration message*, it replies to the new leader with an *identification message*. Upon reception of an *identification message*, the new leader stops sending broadcast messages to this LAN and adds the corresponding group leader to its list. If a given LAN fails to answer a registration message, a crash confirmation concerning this LAN is requested from the other leaders. This mechanism imposes that all the hosts in the system previously possess the broadcast address of every LAN.

6 Performances

We present various performance experiments to illustrate the way detection service works. First we present our test platform. We have implemented an application to illustrate how the detection service operates. This application uses the failure detection service to build a global vision of the system. Next we show how the adaptor works and how it can avoid false detections. Finally, we present measure to evaluate the performance of the hierarchical service mechanisms: leader election, quality of service adaptation according to network load.

6.1 Test platform

To emulate a large scale system, we use a specific distributed test platform, that allows to inject network failures and delays. We establish a virtual router by using DUMMYNET [11] and IPNAT. We use IPNAT, an IP maskering application, to divide our network into virtual LANs. DUMMYNET is a flexible tool originally designed for testing networking protocols. It simulates bandwidth limitations, delays, packet losses. In practice, it intercepts packets, selected by address and port of destination and source, and passes them through one or more objects called queues and pipes which simulate the network effects. In our experiment, each message exchanged between two different LANs passes through this specific host. We don't use DUMMYNET tools for intra-LAN communication because the minimum delay (around 100ms) introduced is too large.

The features of the test system are as follows:

- a standard "pipe" emulates the distance between hosts with a loss probability and a delay
- a random additional "pipe" simulates the variance between message delays.
- network configuration can be dynamically changed, thus simulating periods of alternate stability and instability.

6.2 Global vision building application

As shown in Section 3, the failure information is not directly communicated to the whole system. We consider that the reaction failure is the responsibility of the application. Hence the role of our particular application is to build, for every host in the system, a global vision of the hierarchical system.

The principle of this application is very simple:

- Each host is responsible for its group vision. Each host in a group has a failure detector which watches at the other group members.
- A leader is responsible for its local group. It communicates its local group composition to the other leaders, as well as the composition of the other groups to the members of its local group.

When a new leader joins the system, it sends to other leaders its local group composition and obtain in reply for them their composition, and communicates this information to the other local group members (see Figure 7). After initialization, every host in the system knows the system composition.

If one host is suspected by its group leader, the leader broadcasts this information to the other leaders, which in their turn broadcast it to their respective group members (see Figure 8). The same mechanism occurs at the end of suspicion or when a host reboots. When a leader is suspected by an other one, all the hosts in its group are suspected as having crashed until a new leader is elected.

6.3 Evaluation experiment configuration

In the rest of this section, we present several performance experiment.

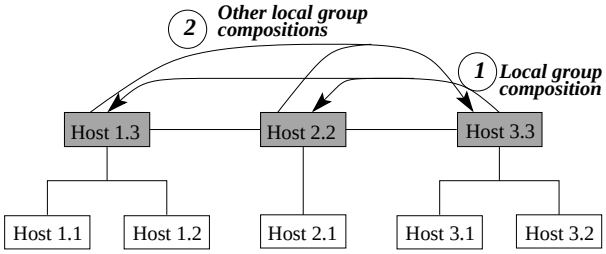

Figure 7. When a leader joins the rest of the system

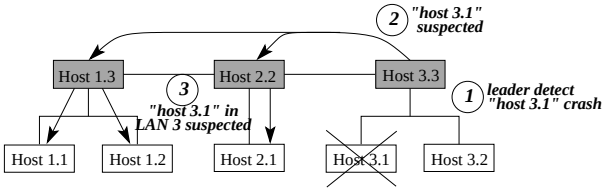

Figure 8. When the leader suspects a member of its group

The experiments were performed on a non dedicated cluster of 12 PCs. We consider a heterogeneous network composed of 8 Pentium III 600 MHz and 4 Pentium IV 1400 MHz linked by a $100\ Mbit/s$ switched Ethernet. This network is compatible with the IP-Multicast communication protocol. The algorithms were implemented in Java (Sun's JDK 1.3) on top of a 2.4 Linux kernel.

PCs are spread in 3 different local groups of four members. The organization is preliminarily known by every system member, as is the broadcast address for every local group.

Throughout the experimentation, the first layer is parameterized as follows:
in local groups $\gamma = 0.1$, $\beta = 1$ and $\phi = 2$, $n = 1000$.
in the global group $\gamma = 0.1$, $\beta = 1$ and $\phi = 2$, $n = 1000$.

In the global group, we use DUMMYNET to emulate Internet connections. The configuration between local group is inspired of real experiments between three hosts located in San Francisco (USA), Paris (France) and Toulouse (France) (see Figure 9). All communication channels are symmetric. Two pipes can applied, A first one is used to simulate the distance between two hosts with a loss probability and a delay.

And a second one is used randomly to introduce variance between message delays.

- group 1 to group 2: first pipe: delay 50ms, probability of message loss 0.012
  second pipe: delay 10ms with a probability of 0.1
- group 1 to group 3: first pipe: delay 10ms, probability of message loss 0.005
  second pipe: delay 4ms with a probability of 0.01
- group 2 to group 3: first pipe: delay 150ms, probability of message loss 0.02
  second pipe: delay 25ms with a probability of 0.3.

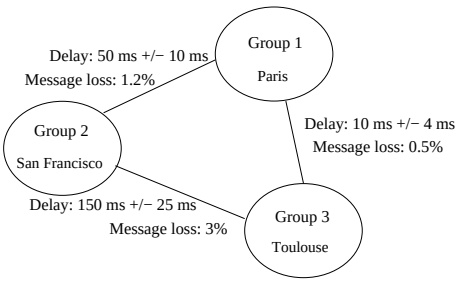

Figure 9. Network configuration

6.4 Global vision application evaluation

Figure 10 presents the time to build the global vision confronted to special events. Every result presented here is an average obtained by repeating the same experiment 10 times.

- The detection time is the time between a host crash and the moment when all the other local group members suspected it.
- Initialization begins with the starting of every host at the same time and ends when all the hosts have the vision of the entire system.
- A host crash delay is the time between a host crash and the moment when all the other hosts are aware of this crash.
- a leader crash delay is quite the same thing as a host crash, yet with a leader election. However the election time (approximately 10ms) is very small compared to the detection time and the propagation time.

The results of this experimentation would be predictable. This application only broadcasts information provided by the failure detectors. The propagation time depends directly upon the "*I am alive*" message emission interval.

Here the detection time average is small compared to the emission interval, because it's the detection time in a local group, where message propagation is quite constant and message losses are very rare.

	Emission Interval Δ_i			
	500	1000	1500	2000
Local Detection Time (ms)	520	1012	1532	2052
Host crash delay (ms)	1062	2133	3104	3924
Leader Crash delay (ms)	1181	2091	3052	4012
Initialization (ms)	1698	3212	4010	8012

Figure 10. Time to build global vision

The theoretical time of the crash event delay is equal to the local detection plus two communications: One emission by the leader of the local group to the other leader, and a second by every group leader to its local group members. Statistically, a message is sent with the next "*I am alive*" then the duration is equal to half the emission time.

6.5 Hierarchical organization evaluation

We experiment to compare processor load in hierarchical organization versus that in a flat system.

The calculation computation collaborates the theoretical calculation in Section 5.2. For this last experimentation, we apply the network configuration of the section 6.3 but in order to compare the generated loads, the emission delay (Δ_i) is the same everywhere: $700ms$. Figure 11 describes the organization according to the number of hosts. Figure 12 illustrates that every message received induces some processing. In a flat organization a host receives more messages, hence the processor load is more important.

total number of hosts	4	6	9	12	16
number of hosts in a local group	2	3	3	4	4
number of local groups	2	2	3	3	4

Figure 11. Experimental topologies

6.6 Adaptor Comparison

The inclusion of an adaptation layer between the failure detector and the application allows to choose between different failure detection strategies. The example presented in Section 4 is that of a QoS negotiation, but it is possible to use the adaptor for other purposes.

For instance, in this performance evaluation we use two different adaptors. The former implements the adaptation algorithm presented in Section 4 based on periodic reevaluation (QoS Adaptor). While the latter is a new adaptor (TD Adaptor) which tones down the detection provided by the basic layer, introducing a *trust degree* (TD) in the first

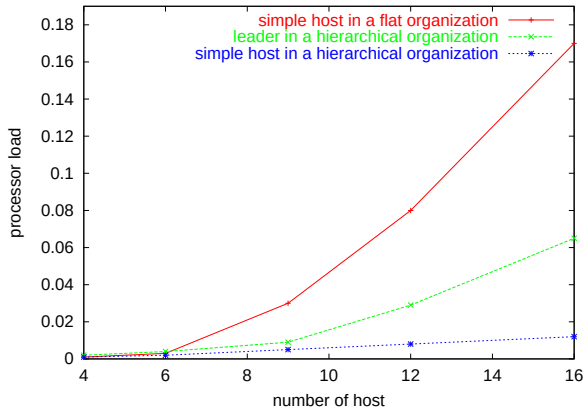

Figure 12. Impact of hierarchical organization in the processor load

layer detection. The trust degree is determined by the number of false detections that occur during a period t. When the first layer suspects a process to have crashed, it computes a refined margin according to this trust degree. The refined margin is increased for each false detection in the period t by half the false detection duration. This solution is not based on a formal quality of service computation, but on the fact that the system configuration is subjected to variations between long periods of alternate stability and instability.

These experiments have been performed over 32 hours, in the conditions described in Section 6.3 with an emission delay equal to $1000ms$. The hosts which take part in the detection are submitted to a normal use by the laboratory staff. The adaptors are configured for a "perfect system" with the variance of message delays: $V_D = 0$ and the message loss probability: $P_L = 0$ for the first adaptor and $TD = 0$ for the second one. The aim of this configuration is to observe how the provided Quality of Service is adapted with respect to network observation. Figure 13 compares theses two adaptors according the metrics presented in Section 3.1.

This experimentation does not aim at comparing the quality of service provided by the adaptors, but to show that in a same host, two applications may have different views of the system.

To show this result we separate the experimentation results from those of the initialization period. We consider the first hour of runtime as the initialization. We can see that both adaptors take a long time to converge to the system characteristics and that during this time a lot of mistakes occur. During the initialization, they represent 83% of the total mistakes for the first adaptor and 50% for the second one. After the initialization, the adaptors provide a regular

		First layer	QoS Adaptor	TD Adaptor
Experimentation	Detection Time	$1216,6$	$2089,9$	$2311,9$
	Number of false detection (ms)	43	6	8
	Mistake duration average (ms)	$1100,6$	$272,5$	$101,4$
Initialization	Detection Time	$1113,2$	$1749,3$	$1351,9$
	Number of false detection (ms)	8	5	4
	Mistake duration average (ms)	$953,2$	$339,3$	$529,1$

Figure 13. Comparison of the two adaptors

detection quality.

7 Conclusion

Our failure detection service provides an adaptable quality of detection, and minimizes network and processor overload. The failure detection service presented in this paper is a part of the DARX (Dymanic Agent Replication eXension) project. This service provides information to the hierarchical naming service and allows the observation service to compute network statistics and to piggy-back its information flow.

This implementation is a shared service between several applications, suitable for large-scale environments. This implementation split into two layers: the basic layer provides a short detection time and adapts the emission interval to the network conditions. The adaptation layer customizes the quality of service provided by the first one according to application needs. This architecture allows to have a detection service customized for each application, yet with a single communication flow. For each new application, the only part of our implementation which requires code rewriting is the adaptor. Another important specificity is the hierarchical organization of the detection service, mapped upon the network topology. This organization can be used with a considerable amount of hosts and concentrates communications in local area networks.

Presently, we are studying how to upgrade the failure detection service so as to detect network partitioning. In the current implementation, detectors consider every silent process as having crashed. If the network is partitioned then two sub-systems may coexist independently. In some cases, to avoid the loss of global consistency, all but one partition must suspend their activity. However, it is crucial to avoid false partitioning detections as they can cause the self-termination of the entire failure detector service. The aim of our actual research is to be able to differentiate several process failures from a network partitioning. We intend to analyze both the simultaneity between failures and the network topology knowledge so as to avoid a maximum of false decisions.

References

[1] *IBM Aglets homepage*. http://www.trl.ibm.com/aglets/.

[2] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proc. of Int'l Conf. on Dependable Systems and Networks*, Washington D.C., USA, June 2002.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 1996.

[4] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. In *Proc. of the First Int'l Conf. on Dependable Systems and Networks*, 2000.

[5] B. Devianov and S. Toueg. Failure detector service for dependable computing. In *Proc. of the First Int'l Conf. on Dependable Systems and Networks*, pages 14–15, juin 2000.

[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, apr 1985.

[7] N. W. Group. Rfc 2988 : Computing tcp's retransmission. *http://www.rfc-editor.org/rfc/rfc2988.txt*, 2000.

[8] S. Haddad, F. Nguilla, and A. E. F. SEGHROUCHNI. A consensus protocol for wide area networks. In *Workshop on Distributed and Parallel Systems (DAPSYS'98)*, Budapest, Hongrie, 1998.

[9] M. Larrea, A. Fernández, and S. Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *Proc. of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC-00)*, pages 334–334, NY, July 16–19 2000. ACM Press.

[10] O. Marin, P. Sens, J.-P. Briot, and Z. Guessoum. Towards adaptive fault-tolerance for distributed multi-agent systems. In *Proc. of European Research Seminar on Advances in Distributed Systems*, pages 195–201, May 2001.

[11] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[12] I. Sotoma and E. Madeira. Adaptation - algorithms to adaptative fault monitoring and their implementation on corba. In *Proc. of the IEEE 3rd Int'l Symp. on Distributed Objects and Applications*, pages 219–228, september 2001.

[13] P. Verissimo, A. Casimiro, and C. Fetzer. The timely computing base: Timely actions in the presence of uncertain timeliness. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 533–542, New York City, USA, june 2000. IEEE Computer Society Press.

[14] N. Wijngaards, B. Overeinder, M. van Steen, and F. Brazier. Supporting internet-scale multi-agent systems. *Data and Knowledge Engineering*, 41:229–245, June 2002.