Predicting Violations of QoS Requirements in Distributed Systems

Sandra Taylor and Hanan Lutfiyya

Department of Computer Science
The University of Western Ontario London, Ontario, Canada N6A 5B7
{taylor,hanan}@csd.uwo.ca

Abstract. A Quality of Service (QoS) requirement refers to a non-functional requirement such as performance. A QoS management system allocates and schedules computing resources. A dynamic QoS management system is one that dynamically allocates resources to an application during its lifetime. This is usually done when the application has a QoS requirement that is not being satisfied at run-time. Ideally, the QoS management system is able to predict when the QoS requirement will be violated before it is violated. This paper describes an approach to prediction and hows shows how this was applied to a case study.

Keywords: Quality of Service, Fault Management, Prediction, Multimedia

1 Introduction

There has been an increase in distributed applications that involve the exchange of data that is time-sensitive. Applications include video-on-demand, distance education, tele-medicine, tele-conferencing and electronic commerce. Users of these applications expect them to perform at acceptable levels, that is, they expect a high level of quality of service (QoS). Quality of service in this context refers to non-functional, run-time (or operational) requirements, such as the application's performance or availability. A possible QoS requirement for an application receiving a video stream is the following: "The number of video frames per second displayed must be 25 plus or minus 2 frames". A QoS requirement is *soft* for an application, if the application is functionally correct even though the QoS requirement is not satisfied at run-time. Otherwise, the QoS requirement is said to be *hard* (e.g., flight control systems, patient monitoring systems). This work primarily focusses on soft QoS requirements.

The term *QoS management* is used to refer to the allocation and scheduling of computing resources. QoS management techniques, such as resource reservation and admission control techniques (e.g., [4]), can be used to guarantee QoS requirements, but these techniques usually base the resource reservation on worst-case scenarios which leads to inefficient resource utilisation. This is important for applications that have hard QoS requirements but is not usually

A. Marshall and N. Agoulmine: MMNS 2003, LNCS 2839, pp. 355–367, 2003.

needed for applications with soft QoS requirements since these applications are still considered functionally correct if the QoS requirement is not satisfied. Other QoS management techniques focus on either having applications adapt their behaviour based on reduced resource availability (e.g., change of video resolution) or make adjustments to resource usage in the system.

We developed a QoS management system [6] that deals with soft and dynamic QoS requirements by providing management services (implemented by a set of management processes and resource managers that support the following: (1) Detecting that an application's run-time behaviour does not satisfy the application's QoS requirements. This violation of QoS requirements (also called a *symptom*) is a manifestation of a fault in the system. (2) Determine (based on a set of symptoms) a set of hypothesis identifying possible causes of a location of the fault causing the violation of the QoS requirements. (3) Adaptation which may take either the form of resource allocation adjustments or application behaviour adjustments.

Currently, detecting that an application's QoS requirement is violated occurs after monitoring shows that the application's attributes used in the specification of the QoS requirement does not satisfy the constraints specified in the QoS requirement. In the example of an application receiving a video stream, the QoS requirement is detected to be violated when the frame rate is less than 23 or more than 27. If this QoS requirement is not being satisfied, it is most likely that the user has noticed that the application is not executing as expected.

Ideally, the QoS management system will be able to predict when the QoS requirement will be violated, i.e., determine that the QoS requirement will be violated before it is actually violated, and make the necessary adjustments. It may not always to be possible to make the necessary adjustments before the QoS requirement is violated, but at the very least the management system is more responsive.

This paper is organised as follows. Section 2 defines configuration models and describes how configuration models are related to detection. Section 3 analyses an application's QoS requirements, and shows how prediction rules may be generated. Section 4 describes an architecture and implementation that is able to do the prediction. Section 5 provides an analysis of the effectiveness of the methods described in Section 3. Section 6 provides a discussion. Section 7 describes related work and Section 8 provides conclusions.

2 Relationship between Configuration Models and Detection

This section describes the relationship between configuration models and diagnostics. This relationship can be used to derive rules for prediction. The relationship is illustrated by examining multimedia applications.

Defining Configuration Model

Fig. 1 illustrates the internal structure of the application and the underlying system from an end-to-end perspective.

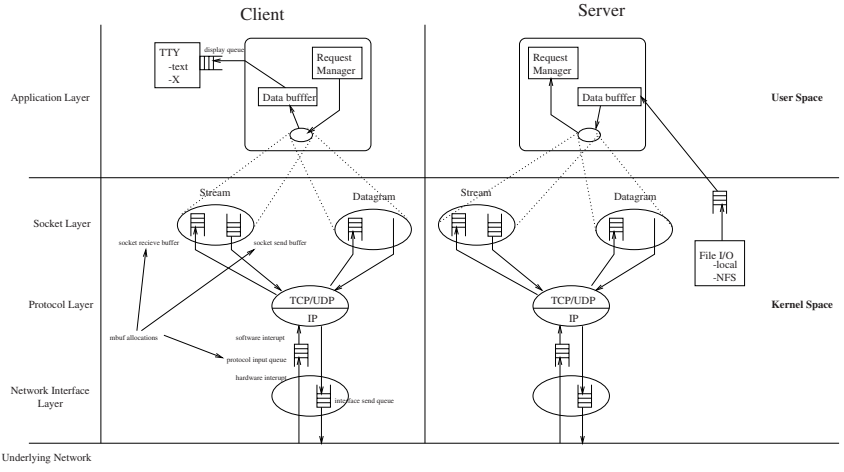

Fig. 1. Configuration Model of Multimedia Application

The data for the video image is retrieved from the disk into a buffer in user space. This gets copied into data buffers (*mbuf* for Unix System 4 kernels) that are used by the TCP/UDP protocol processes. This progresses through to the Protocol layer where the IP processing occurs. Each layer has data buffers as presented in Fig. 1.

The client side consists of a process (called the Request Manager) that has its own communication data buffer for transferring data. The client request manager sends the processed frames to the display process which is a text display or it can continue through an X server event queue to a graphical display system or to any other device.

Generally for any application the internal structure of the application and the underlying system can be represented as a graph $C = (V, E)$, where V represents the structural components of the application and system (in terms of the services provided) and for any $o_i, o_j \in V$ there exists $(o_i, o_j) \in E$ if o_i has a direct dependency on o_j i.e., o_i needs o_j in order to be functionally correct. This is refered to as a *configuration model*.

From the configuration model, *dependency chains* are identified. If an object o_i is dependent on o_{i+1} and o_{i+1} is dependent on o_{i+2} and so on until we have o_{i+j-1} is dependent on o_{i+j} (where $j \geq 1$) then o_i is said to be indirectly dependent on o_{i+j}. $o_i, o_{i+1}, ..., o_{i+j}$ is said to be *dependency chain* if for an l and m, where $i \leq l < m \leq i + j$, o_l is indirectly dependent o_m. A dependency chain does not have to have every single component possible in the chain. In the example presented in this section, a possible dependency chain is the following: client display process, client request manager and server request manager.

Identifying Measurements. At this point, we identify those attributes of the components identified in the previous step that characterise its behaviour and can be measured. These are referred to as *observable indicators*. The subset of

these observable indicators that are directly used in the specification of QoS requirements are referred to as *QoS attributes*. Assuming that we have a graph $C = (V, E)$ then for each $o_i \in V$, we can associate a set $\{a_{i1}, a_{i2}, ...a_{in}\}$ where a_{ij} is an observable indicator associated with o_i. For our example, observable indicators include the size of each of the incoming buffers. QoS attributes (which have an impact on the user's perception of the satisfaction of QoS requirements) include frames per second. These are observable indicators of the client display process.

Determining Correlations

In this context, a *fault* refers to the lack of a computing resource needed by an application to satisfy its QoS requirements.

We now define the concept of a *chain of failures*. A fault may occur in object o_n. This may cause an error to manifest itself in object o_{n-1}, where $(o_{n-1}, o_n) \in E$. This in turn may manifest itself as an error in object o_{n-2}, where $(o_{n-2}, o_{n-1}) \in E$. This results in a sequence of objects $o_n, o_{n-1}, .., o_1$ where o_n is where the fault occurs and o_1 is assumed to include QoS attributes and is assumed to have no other objects dependent on it. This sequence is a chain of failures. A fault may cause more than one chain of failures.

As an example, consider the dependency chain identified earlier: client display process, client request manager and server request manager. A CPU overload on the client machine impacts the rate at which the client request manager is able to process frames which, in turn, impacts the rate at which the client display process is able to display the frames. Thus, an example of a chain of failures is the following: server request manager, client request manager and client display process.

For each observable indicator a_{kl} we refer to the t^{th} measurement of a value of a_{kl} as a^t_{kl}. This is referred to as a *sample*. Let $S_{a_{kl}}$ denote the set of samples of a_{kl} taken during a run. Let g_k be a function of $S_{a_{kl}}$ and let x be the variable that takes values x_1, x_2, ..., x_n which are computed by $g_k(1)$, $g_k(2)$ and $g_k(n)$ respectively. Let y be a variable that takes values y_1, y_2, ..., y_n where $y_i = a^i_{ij}$. We say that there is a strong dependency between a_{ij} and a_{kl} if o_i and o_k can be found in the same dependency chain, there is a strong statistical correlation and if a relationship can be defined between y and x. If a_{ij} is a QoS attribute then it may be possible to use values of a_{kl} to predict values of a_{ij} (and thus predict when a QoS requirement is going to be violated), if g_k has the property that the value of x_i is contributed by a subset of those a^t_{kl} that represent samples that were taken before a^i_{ij}. In this paper, a_{ij} will be the frame rate and a_{kl} will be the size of the incoming communications buffer of the client's request manager. We will see that $g_k(i)$ results in the sum of the four previous buffer sizes i.e., $x_i = g_k(i) = b_{i-1} + b_{i-2} + b_{i-3} + b_{i-4}$.

It should be noted that it may be possible to use more than one attribute to predict a_{ij}. However, this paper focusses on the use of defining the relationship between two attributes: frame rate and size of the client request manager's communication buffer.

The concept of a configuration model is similar to that of a causality model (for a good overview of causality models and how they are used in fault management see [8]). The primary difference is that this work associates with each component a set of attributes characterising its behaviour.

3 Analysing the RTVC Application

This section describes the experiments and methods of analysis that were used to determine possible correlations for the Real Time Video Conferencing (RTVC) tool which is a part of the Sun Multimedia kit. The configuration model is similar to that of Fig. 2. The example QoS requirement referred to in the rest of this paper is the following: "The number of video frames per second displayed must be 25 plus or minus 2 frames". The QoS attribute of interest is the frame rate. Observable indicators include the size of each of the incoming buffers. We chose to use the size of the client's communication buffer (incoming buffer to the client's request manager).

Experiments were performed on laboratory machines that were isolated from other machines. Loads were generated based on a Poisson distribution.

3.1 Measuring Frame Rate and Communication Buffer Size

The frames per second rate is calculated as follows. Assume that f_i represents the i^{th} occurrence that the frames per second rate is reported and that a report is to be made after every ten frames that have been displayed. Let t_{f_n} represent the system clock time when the n^{th} frame is displayed and let $t_{f_{n+10}}$ represent the system clock time when the $(n+10)^{th}$ frame is displayed. The frame rate is calculated as $(10/(t_{f_{n+10}} - t_{f_n})$ The choice of determining the current frame rate after ten frames was received is based on experimentation. Anything less seemed to cause inaccurate predictions by reporting more false negatives. Anything after ten frames seemed to not be able to predict problems.

We measured the current size of the communication buffer (referred to as buffer throughout the rest of the paper), where b_i denotes the number of bytes occupying the buffer as measured in the ith sample, and the average number of bytes stored in the buffer. This value is calculated by adding 10 per cent of the current number of bytes in the buffer to 90 per cent of the average buffer value calculated in the previous sample. This was done every second.

3.2 Correlation Analysis

Graphing representative runs show that the the frames per second is negatively affected at or near the times when the buffer size spikes. There appears to be a good correlation between the buffer size and the frames displayed per second.

For an example representative run, the arithmetic mean of the sample buffer sizes is $\bar{b} = \sum_{i=1}^{599} b_i = 1996.633$ and the arithmetic mean of the sample frame rates is $\bar{f} = \sum_{i=1}^{599} f_i = 25.622$. The standard deviation for the buffer size, s_b is

calculated as follows: $s_b = \sqrt{\frac{1}{n-1}\sum(b_i - \bar{b})^2} = 3319.864$. The standard deviation for the frame rate, s_f is calculated as follows: $s_f = \sqrt{\frac{1}{n-1}\sum(f_i - \bar{f})^2} = 5.919$. These are relatively high measurements of standard deviation.

This then leads to the analysis of the correlation [2] between the two sets of variables. The standard calculation for correlation measures the strength of the linear relationship between two variables. When the two sets of variables being used are represented by x_i and y_i, and the means and standard deviations of the two variables are represented by $\bar{x}$ and s_x for the x values, and $\bar{y}$ and s_y for the y values, the equation is as follows: $r = \frac{1}{n-1}\sum\left(\frac{x_i - \bar{x}}{s_x}\right)\left(\frac{y_i - \bar{y}}{s_y}\right)$.

The measure of the relationship is reflected in a numerical result which lies between -1 and 1. The closer to zero the result is, the weaker the relationship is between the two variables. The strength of the relationship increases as r approaches either 1 or -1. If r is positive, the relationship between the variables is positive, whereas if r is negative, the relationship is negative. As we attempted to determine the best relationship between our two variables, we performed the correlation analysis using $s_x = s_f$, $s_y = s_b$. Table 2 presents values of r for different values computed for x_i.

x_i	**r**
b_i	-0.70258
b_{i-1}	-0.75709
b_{i-2}	-0.72961
b_{i-3}	-0.71012
$\frac{b_{i-1}+b_{i-2}}{2}$	-0.80955
$\frac{b_{i-1}+b_{i-2}+b_{i-3}}{3}$	-0.82667
$\frac{b_{i-1}+b_{i-2}+b_{i-3}+b_{i-4}}{4}$	-0.83506
$\mid b_i - b_{i-1} \mid$	-0.42106

Fig. 2. Correlation Table

Correlation values were computed for $x_i = b_i$, $x_i = b_{i-1}$, $x_i = b_{i-2}$ and $x_i = b_{i-3}$ to get -.70258, -0.75709, -0.72961 and -0.71012 respectively.

These negative values reflect the fact that there is a negative relationship between the values of the buffer size and the frames per second samples; i.e., as the buffer size increases, the frames per second figure decreases, rather than a positive relationship where they would both go in the same direction.

There were several more relationships tested moving the time differential further way, but r continued to decrease in the same pattern as it did after peaking at b_{i-1}.

We then tried correlations where x_i was defined using the average of measurements of buffer size of two samples $(b_{i-1}+b_{i-2})$ which represents the average buffer size in the previous two seconds, three samples $(b_{i-1}+b_{i-2}+b_{i-3})$ which

represents the average buffer size in the previous three seconds, and four samples $(b_{i-1} + b_{i-2} + b_{i-3} + b_{i-4})$ which represents the average of buffer sizes in the previous four seconds. This was examined since often there may be a spike in the buffer size that lasts briefly and not have a significant impact on frames per second. Each of these resulted in a higher correlation measurement than the initial set described. The results are summarised in Fig. 3.

There were further combinations of buffer values used in our attempts to come up with the best relationship between the buffer values and the frames per second values. For instance, we looked at the absolute difference in the buffer size values $(\mid b_i - b_{i-1} \mid -\bar{b})$ from one sample to the next to see what relationship existed between those values and the frames per second values. The resulting correlation value was -0.42106. This result when compared to the previous results showed that this was the wrong direction in which to proceed.

3.3 Regression Analysis

The correlation analysis shows that there is a linear relationship between the buffer size and the frames per second values that are reflected between three and six samples later. We used a linear regression model to describe this relationship. More formally, consider a first-order linear regression model of the following form: $y_i = \mu + \alpha x_i$ where $y_i = f_i$ and $x_i = (b_{i-1} + b_{i-2} + b_{i-3} + b_{i-4})/4$. Least-squares regression was used to determine the value of α and μ. The predicted value is denoted by $\hat{f}$. The equation corresponds to a trendline and is called the trendline equation. Least squares regression was used to determine the values of the parameters of the linear regression model. This resulted in the following: $\hat{f} = 28.9653 - (1.701309E^{-3})((b_{i-1} + b_{i-2} + b_{i-3} + b_{i-4})/4$

The value of α is a measure of the change in the fames per second for every increment of one in $(b_{i-1} + b_{i-2} + b_{i-3} + b_{i-4})/4$. That is, every time the buffer occupancy level increases by one byte, the frames per second rate will decrease by $1.701309E^{-3}$. If the model fits the measured data well, then α is a direct measure of the average of the last four sampled buffer sizes on the frame rate. Specifically, if α is statistically non-zero(at say the 95% confidence level), then we conclude that the frame rate depends on the average of the last four sampled buffer sizes with a strength of α. We used t-statistics and P-value to confirm this.

We used the equation $\hat{f} = 28.9653 - (1.701309E^{-3})((b_{i-1} + b_{i-2} + b_{i-3} + b_{i-4})/4)$ to not predict the frames per second, but whether the frames per second is going to fall below or fall above a specific threshold value as defined by the QoS requirements.

If the target frame rate is 25 then for $25 = 28.9653 - (1.701309E - 3)((b_{i-1} + b_{i-2} + b_{i-3} + b_{i-4})$ we have that $(b_{i-1} + b_{i-2} + b_{i-3} + b_{i-4})/4$ should be equal to 2315 (this is rounded). For a target frame rate of 23 the value of $(b_{i-1} + b_{i-2} + b_{i-3} + b_{i-4})/4$ should be 3525 and for a target frame rate of 27 the value of $(b_{i-1} + b_{i-2} + b_{i-3} + b_{i-4})/4$ should be 1191.

Thus, for the QoS requirement "The number of video frames per second displayed to the user must be at least 25 plus or minus 2 frames", if the buffer

size is less than 1191 or greater than 3525 then this is an indicator that the QoS requirement will be violated.

4 Architecture

Monitoring of the application is needed to collect values of the attributes (e.g., current_fps). One monitoring mechanism is through instrumentation of the application which is briefly described here.

An attribute is associated with a sensor. A sensor is a class with variables for representing threshold and target values. Sensors are used to collect, maintain, and process a wide variety of attribute information within the instrumented processes. The sensor's methods (*probes)* are used to initialise sensors with threshold and target values and collect values of attributes. Probes are embedded into process code to facilitate interactions with sensors. The *coordinator* is the instrumentation component that is between the process and the management system. The coordinator is implemented as a thread of the application process.

There are two sensors used in this work: fps_sensor and socket_buffer_sensor. Fig. 3 illustrates the use of fps_sensor in example psuedo-code for a video playback application that has the QoS requirement "The number of video frames per second displayed to the user must be at least 25 plus or minus 2 frames". This QoS requirement suggests an upper threshold of 27 frames per second and a lower threshold of 23 frames per second. Sensor fps_sensor includes probes such as the following: (1) An initialisation probe (line 3 of Fig. 3) that takes as a parameter the default threshold target value, and the lower and upper bounds. When the coordinator is instantiated (line 2 of Fig. 3) it communicates with the QoS management system to get the application's QoS requirements in the form of a condition list which represents a condition by an attribute identifier, the identifier of a sensor that monitors that attribute, a comparison operator and value that the attribute is to be compared to using the comparison operator. Thus when the sensor requests this information, the coordinator will already have retrieved it. (2) A probe, probe_framerate(), that (i) determines when ten frames have been displayed. For every ten frames displayed, it calls a function that returns the system clock time. This and the system clock time returned ten frames ago is used to calculate the current frame rate and checks to see if this time falls within a particular range defined by the lower and upper acceptable thresholds. Unusual spikes are filtered out; and (ii) informs the coordinator through the report method if the frames per second fall below the lower threshold or is higher than the upper threshold. The report method is implemented as a thread.

The socket_buffer_sensor monitors the length of the communication buffer. This is done as follows. A socket provides for interprocess communication. In UNIX, a socket is a file descriptor. The kernel allocates an entry in a private table in the process area, called the user file descriptor table and notes the index of this entry. The index is the file descriptor that is returned to the process. The entry allocated is a pointer to the first *mbuf* structure. The memory associated with

Given: Video application *v*.
QOS expectations *e*.

---

1. Perform initialization for *v*.
2. Initialise coordinator *c*.
3. Execute *fps_sensor* → *init_probe(e)*
4. **while** (*v* **not done**) **do:**
5. Retrieve next video frame *f*.
6. Decode and display *f*.
7. Execute *fps_sensor* → *probe_framerate()*
8. **endwhile**

Fig. 3. Instrumentation Example

the *mbuf* structures of the process is the buffer. The sensor, socket_buffer_sensor, implements a function getBufferStats() that given a file descriptor for a socket returns buffer statistics including the length of the buffer.

Both the fps_sensor and the socket_buffer_sensor have read and report methods. When these sensors are instantiated, the report methods are initiated as threads through a function call so that they can communicate with the coordinator on an ongoing basis, separate from the application.

The report method of the socket_buffer_sensor calls at regular intervals of one second the read method which in turn calls getBufferStats() which returns the buffer statistics. In the i^{th} interval, the read method computes $(b_{i-1}+b_{i-2}+b_{i-3}+b_{i_4})/4$ and compares it to a threshold value. Based on the trendline equation computed in the previous section, it determines if the calculated value falls below 1191 or rises above 3525. If it does then a *warning* indication is generated. Otherwise, a *no warning* indication is generated. These are indicated by 0 and 1 respectively and are sent to the coordinator.

For the purposes of analysis, the fps_sensor was subclassed with the probe_framerate() rewritten so that it reports all calculated frame rate values and not just the frame rate values that violate the QoS requirements. The coordinator was subclassed so that it takes reports from the fps_sensor and the buffer_sensor and puts those reports into a logfile.

5 Analysis

To analyse the effectiveness of the approach we had to find a way to transpose the log file reports of frames per second into a second-by-second calculation that can be used in our analysis. This is denoted by fps_{t_i} where t_i represents the time at i seconds into the run.

We let fps_j represent the j^{th} occurrence (between time t_i and t_{i+1}) that the frames per second rate is reported. The equation used for averaging is the following: $fps_{t_n} = \frac{fps_1 + fps_2 + ... + fps_j}{j}$

After extracting the warnings from the log report, a file of warnings would look something like this: 0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,1,0,0,0,0,0,1,1,1,1,1,....

We correlated these warnings with the frames per second rate. We only needed to know when the warning was turned on and when it was turned off. Repetitions of a warning value were filtered.

This information was used in our analysis that focussed on determining the effectiveness of our approach to prediction. These results are now presented and are only with respect to frame rates being too low. We have the following categories of warnings:

- On-Time Warning – This warning is issued between one and five seconds in advance of the time frame where the frames per second rate drops below the minimum threshold.
- Late Warning – This is a warning which is issued, but is only issued less than one second in advance of the drop in frames per second rate below the minimum threshold.
- False Positive Warning – This is a warning that is issued where there is no instance of a frames per second reading which is below the minimum threshold.
- No Warning – This is an occurrence of the frames per second rate dropping below the minimum standard where no warning at all is received.

In Fig. 4, we show an example of the warnings that have been received in a specific run of our experiments. Series 1 represents the values of the frames per second as calculated above, and Series 2 represents the warnings given and filtered as previously discussed. We can readily see in this graph that the warnings received appear to arrive, in most cases, just at the beginning of a decrease in frames per second values.

The portion of correct early warnings is represented as P_p. The portion of false predictions is represented by P_f and the portion of late warnings or no warnings is represented by P_l (these were combined since late and no warnings have the same end result).

Warning percentage calcuations for three case studies of video each of which is approximately ten to fifteen minutes in length was calculated. The value of P_p for each of the case studies was 0.63, 0.62, and 0.59 respectively. The value of P_f for each of the case studies was 0.013, 0.027 and 0.068 respectively. The value of P_l for each of the case studies was 0.35, 0.37 and 0.36 respectively. The results are consistent and demonstrate that is possible to correctly predict in advance the violation of the QoS requirement approximately 60 per cent of the time. The excellent results though are reflected in the paucity of false positives that we received. This is an important feature since reacting to a false positive warning could be more costly if the QoS management system is forced to allocate increased system time and resources to the application that would go to waste.

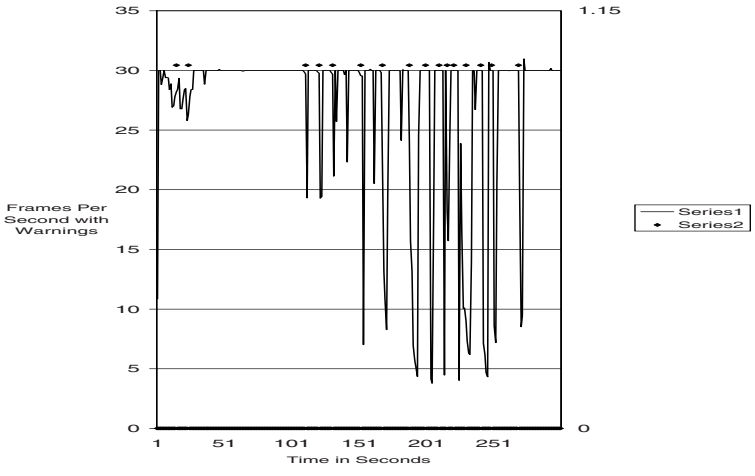

Fig. 4. Graphic Display of Warnings Relative to FPS Rates

6 Discussion

1. The sensors presented here can be used in any application that receives a multimedia stream. Only the parameters of the trendline equation and the threshold values would change. These could be read from a configuration file by the coordinator upon application start-up.
2. We initially tried to correlate the observable indicators of frame loss and jitter with the frame rate. The statistical correlation computed was so low that it was not deemed necessary to continue along these lines. We would like to improve the percentage of on-time warnings without increasing false positive warnings by examining the use of multiple attributes to predict that a QoS requirement will be violated.
3. In this work, the relationship between attributes of the different objects of dependency chains is represented using linear regression. More case studies are needed to determine other ways to relate information. There is the possibility that different parameters could be used in a vector relationship or a matrix of vectors or possibly in a weighted combination of parameter statistics.
4. This paper assumed that the fault will a heavy CPU load that causes the CPU to be a resource lacking by the application. However, it may be the case that fault is a lack of several computing resources. This should be taken into account.

7 Related Work

The earliest work found in predictive fault detection is found in [5]. These papers characterise normal network behaviour. They developed models that seem rather ad-hoc to estimate weekly patterns. This takes into account that the definition of normal changes over time. This is experimentally validated through a case study where the introduction of backups (this caused heavy network traffic) causes a change in normality. Changes in the statistics of traffic data such that they do not conform to the definition of normality can be used to detect anomalies. However, this work does not particularly focus on prediction.

The work in [9] describes how to detect changes in network behaviour that are closely correlated with service interruptions. Some approaches to fault detection have been developed based on Bayesian networks (e.g., [3]). In [7], the basic approach is to model the threshold metric at two levels: non-stationary behaviour as in workload forecasting and stationary behaviour with time-serial dependencies, in order to compute the probability of violations. This apparently worked well with two caveats: if the actual values of predictions were far enough from the threshold and if the prediction horizon was not too far in the future.

There is a good deal of work on intrusion detection that is somewhat relevant. A great deal of research in intrusion detection is based on the hypothesis that network attacks cause abnormal behaviour. Much of the work in this area models normal network behaviour so that behaviour that does not satisfy the constraints of normal network behaviour is assumed to be caused by intrusion. Examples of this work include [1].

Most of the work described focusses on characterising network traffic. The work in [10] focusses on characterising application behaviour. They use the number of requests processed per second as a measurement of web performance. This was measured every 15 minutes. We will refer to each measurement as an observation. The methodology used discarded all observations that were known to correspond with a fault. The rest of the observations were used to create distribution models for "normal" behaviour.

As can be seen, there is an area of research that is being done in proactive or predictive detection of failures, but most of the work is being done using network parameters and statistics. Work that is directed towards proactive failure detection at the application is not abundant.

8 Conclusions and Future Work

The work in this paper shows that an approach to prediction based on dependency chain can provide good results. Future work includes addressing the questions brought up in the discussion section. The long-term goal is to develop a set of tools that facilitate the development of trendline equations and sensors that make use of these equations.

References

1. M. Bykova, S. Ostermann, and B. Tjaden. Detecting network intrusions via a statistical analysis. In *Proceedings of the 33rd Southeastern Symposium on System Theory*, pages 309–314, 2001.
2. R. Carter Hill, William E. Griffiths, and George G. Judge. *Undergraduate Econometrics*. John Wiley and Sons, Inc., 2000.
3. C. Hood and C. Ji. Probabilistic network fault detection. In *Global telecommunications Conference, 1996: Communications: They key to Global Prosperity*, pages 1872–1876, 1996.
4. M. Jones, D. Roşu, and M. Roşu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. *Proceedings of the Sixteenth ACM Symposium on Operation Systems Principles*, St. Malo, France, October 1997.
5. R. Maxion and F. Feather. A case study of ethernet anomalies in a distributed computing environment. In *IEEE Transactions on Reliability 39(4)*, pages 433–443, October 1990.
6. G. Molenkamp, H. Lutfiyya, M. Katchabaw, and M. Bauer. Resource management to support application-specific quality of service. *IEEE/IFIP Management of Multimedia Networks and Services (MMNS2001)*, October 2001.
7. D. Shen and J. Hellerstein. Predictive models for proactive network management: Application to a production web server. In *Proceedings of the 2000 IEEE/IFIP Network Operations and Management Symposium*, pages 833–846, 2000.
8. M. Steinder and A. Sethi. The present and future of event correlation: A need for end-to-end service fault localization, 2001.
9. M. Thottan and C. Ji. Fault prediction at the network layer using intelligent agents. In *Proceedings of the 1999 IEEE/IFIP International Symposium on Integrated Network Management*, pages 745–759, 1999.
10. A. Ward, P. Glynn, and K. Richardson. Internet service performance failure detection. In *1998 Web Server Performance Workshop*, pages 38–43, June 1998.