On-Line Failure Prediction in Safety-Critical Systems[*]

Roberto Baldoni and Luca Montanari

Cyber Intelligence and Information Security Research Center and

Department of Computer, Control, and Management Engineering Antonio Ruberti

"Sapienza" University of Rome, Italy

`baldoni@dis.uniroma1.it - montanari@dis.uniroma1.it`

Marco Rizzuto

Selex-ES, Rome, Italy

`mrizzuto@selex-es.com`

Abstract

In safety-critical systems such as Air Traffic Control system, SCADA systems, Railways Control Systems, there has been a rapid transition from monolithic systems to highly modular ones, using off-the-shelf hardware and software applications possibly developed by different manufactures. This shift increased the probability that a fault occurring in an application propagates to others with the risk of a failure of the entire safety-critical system. This calls new tools for the on-line detection of anomalous behaviors of the system, predicting thus a system failure before it happens, allowing the deployment of appropriate mitigation policies.

The paper proposes a novel architecture, namely CASPER, for online failure prediction that has the distinctive features to be (i) *black-box*: no knowledge of applications internals and logic of the system is required (ii) *non-intrusive*: no status information of the components is used such as CPU or memory usage; The architecture has been implemented to predict failures in a real Air Traffic Control System. CASPER exhibits high degree of accuracy in predicting failures with low false positive rate. The experimental validation shows how operators are provided with predictions issued a few hundred of seconds before the occurrence of the failure.

[*]A preliminary version of this paper appeared at SAFECOMP 2012.

Keywords: Failure Prediction, Complex Event Processing, Machine Learning, Complex Distributed Systems, Critical Infrastructures.

1 Introduction

A few years ago, safety-critical systems traditionally used in air traffic control, commercial aircraft, nuclear power, consisted of a monolithic (possibly proprietary) system provided by a single vendor. Such systems thus incurred in high cost of development and maintenance. To reduce such costs, systems have been unpacked in a set of applications/services (usually developed by different vendors) that interact through a set of well-defined interfaces.

Applications need to meet stringent Quality of Service (QoS) requirements in terms of availability in order to ensure, in their turn, the high availability of the whole safety-critical system. To achieve this objective, applications require to distribute and replicate data (e.g., flight routes in Air Traffic Control system) on a number of nodes connected through a WAN or a LAN. Due to the nature of such systems, the replicas of an application need to be strictly consistent in order that they keep the same state along the time, providing a client with the illusion that its request occurs instantaneously [21].

In such complex distributed systems, failures are a matter of life thus they have to be safely handled to ensure system survivability. Extensive testing during the design phase of an application cannot avoid the occurrence at operational time of failures that can lead to catastrophic consequences for the entire system functioning. Keeping a set of replicas strictly consistent in the presence of replica failures boils down indeed to solve the consensus problem [30]. Thus, if the distributed system has a good coverage of synchrony assumptions (i.e, network and computing nodes are working nicely), there are a number of fault tolerance mechanisms that can be used to overcome the failure and to keep replicas consistent (e.g., failure detection through heart-beating [9]). If a fault happens in a period when there is no coverage of synchrony assumption, replicas might manifest anomalous behaviors due to well-known Fischer-Lynch-Paterson (FLP) impossibility result stating that distributed consensus cannot be reached in an asynchronous system even in the presence of only one faulty process [15]. These behaviors could transitively affect other applications bringing in the worst case to a system failure such as, for example, an abnormal system shutdown. In this case, it might take a long time to resume a correct functioning, severely reducing system availability. The only way to avoid such abnormal system failures is to predict them, sensing the occurrence of anomalous behaviors. Accurate and timely predictions can help to mitigate the effect of failures by taking appropriate recovery actions before the failure occurs. Such actions can mitigate the loss of availability by reducing the time needed to resume a correct system behavior.

The industrial trend is to split a safety-critical system in at least two distinct sets of interconnected applications/services possibly developed by two or more distinct manufacturers. The first one handles the application logic (e.g. flight routes management in Air Traffic Control Systems) and the second set is devoted to control and supervision that, among the others, has the task of managing failures. Manufactures require that supervision services interfere as less as possible with the application logic. This is due to the fact that application logic has a long and complex deployment phase at customer premises and the performance of the application logic cannot be influenced in any way by the supervision mechanisms. This implies that for example application logic and supervision cannot be co-located on the same nodes.

For all these reasons, some prominent future challenges of next generation safety-critical systems relate to the capacity of keeping the same, or even increase, the availability of such systems while considering this open multi-manufacturers setting. In particular:

- High-level assurance of non-interference at run-time among distinct applications forming the safety-critical distributed system. Some of these applications should not have any interaction at all (e.g., supervision and application logic) in order to ensure the mutual correctness of their behavior;
- An application, which is a part of the safety-critical distributed system, would have a great value if agnostic with respect to the internals of all others interacting applications. As an example, supervision should have zero knowledge on the deployment and on the source code of the application logic.

These challenges translate into having a supervision that acts like a both *non-intrusive* and *black-box* observer. *Non-intrusive* means the failure prediction does not use any kind of information on the status of the nodes (e.g., CPU, memory) of the monitored system; *black-box* means no knowledge of the application internals and of the application logic of the system is exploited. Operationally, in safety-critical systems, a large amount of data, deriving from communications among applications and services, transits on the network; thus, the "observer" can focus on that type of data, only, in order to recognize many aspects of the actual interactions among the components of the system.

This paper presents a first attempt in the direction of designing next generation highly available safety-critical systems by introducing a novel non-intrusive and black-box failure prediction architecture, namely CASPER. It works *online* since the failure prediction is carried out during the normal functioning of the monitored system and it exploits only information traveling on the network interconnecting nodes of the supervised and monitored systems. Specifically, the aim of CASPER is to recognize any deviation from normal behaviors of the monitored system by analyzing symptoms of faults that might occur in the form of anomalous conditions of specific performance metrics. In doing so, CASPER combines, in a novel fashion, *Complex Event Processing* and machine learning algorithms designed through *Hidden Markov Models*. The latter are used to classify at run time nice states and anomalous states of the monitored system. Roughly speaking when an anomalous state is detected an alarm is sent to the operator. In this paper we show how CASPER can be effective also implementing the simplest version of HMM. We indeed deployed CASPER for monitoring a real Air Traffic Control (ATC) system developed by Selex-ES (a Finmeccanica Company)and conducted a 6 months long experimental evaluation. Results show CASPER timely predicts failures in the presence of memory and I/O stress conditions while keeping reasonably low[1] the number of false positives.

2 Related Work

A large body of research is devoted to the investigation of approaches to online failure prediction. The interested readers can find a comprehensive taxonomy of them in [35]. In this section, we discuss only those that are closer to our solution and that mainly inspired the approach we followed. We can distinguish between approaches that make use of symptoms monitoring techniques for predicting failures (as CASPER does), and approaches that use error monitoring mechanisms. In the latter category, Salfner in [34] presents an error monitoring failure prediction technique that uses Hidden Semi-Markov Model (HSMM) in order to recognize error patterns that can lead to failures. Hoffman et al. in [22] describe two non-intrusive data driven modeling approaches to error monitoring, the first based on a Discrete Time Markov Model and the second one on function approximation.

[1] False positive rate less then 12%, F-Measure more then 80%, see Section 6.2 for details.

In the context of symptoms monitoring, there exist failure prediction systems that use black-box approaches, namely ALERT [36] and Tiresias [38]. Both systems are intrusive in the sense that they interfere with hosts running the application logic by sensing internal parameters such as CPU consumption, memory usage, input/output data rate. Using such internal parameters allow to quickly sense bad behaviors and alert operators. Thus, one of the challenges of CASPER has been to get good accuracy and fast response time while being non-intrusive. A more recent work [2] that can be consider non-intrusive as it uses network traffic only, presents data-mining techniques that extract essential models of anomalous behavior sequences known to be precursors of system failure conditions, i.e., symptoms in our sights. The main difference is that is designed to address network failures only, while our work is more general in that sense. From a model point of view, in the field of failure prediction, [31] presents prediction sub-model for each component and combines them using component dependencies. This is different from other approaches as usually the same model is used for all the components. Note that [31] is intrusive as it requires the installation of monitoring probes to the observed components.

Out of the area of failure detection systems, CASPER got inspired by the following approaches: Aguilera et al. in [3] consider the problem of discovering performance bottlenecks in large scale distributed systems consisting of black-box software components. The system introduced in [3] solves the problem by using message-level traces related to the activity of the monitored system in a non-intrusive fashion (passively and without any knowledge of node internals or semantics of messages). The main difference respect to our work is that a bottleneck is not properly a failure, but can be consider as a performance degradation. Fu and Xu in [16] analyze the correlation in time and space of failure events and implements a long-term failure prediction framework named hPrefects for such a purpose. hPrefects forecasts the time-between-failure of future instances based on the correlations among failures. The difference, in this case, is that we perform short term online failure prediction: avoid the occurrence of an upcoming failure basing on monitoring of symptoms, caused by faults, that will lead to that failure while hPerfects addresses long-term failure prediction: it estimates the occurrence of future failures using data on past failure occurrences, without symptoms monitoring. A complementary work on how to react to the determination of an “unsafe state” that requires to transition to a new “safe state” can be done with functional composition techniques such as described in [19].

3 Background

The approach followed in this work combines Complex Event Processing (CEP) and Hidden Markov Models (HMM) to analyze symptoms of failures that might occur in the form of anomalous conditions of performance metrics identified for such purpose. We used HMM in order to recognize the state of the observed system starting from a set of timely changing performance metrics, computed by a CEP engine. In this section, some basic definitions about these two techniques are provided.

3.1 Hidden Markov Model

A Hidden Markov Model (HMM) is a statistical model (i.e., a formalization of relationships between variables in the form of mathematical equations) in which the system modeled is assumed to be a Markov process with hidden states. It can be considered as the simplest dynamic Bayesian network. Differently from regular Markov model, in hidden Markov model the state is not visible to the observer. The latter can just see the output(s) emitted but a state. Then the observer looking at the sequence of outputs can infer the sequence of hidden states. HMM allows thus to recognize the state of the modeled system starting from observations of its output, building thus a classification of the output symbols.

HMM are employed in several fields to detect temporal patterns, such as voice recognition [32], clustering in time-series data [4], intrusion detection [25], computational biology (e.g., [13, 27] just to name a few. In [39] a human action recognition method based on a Hidden Markov Models is presented. A set of time-sequential images is transformed into an image feature vector sequence, and the sequence is converted into a symbol sequence by vector quantization. The goal is to classify observed images sequences into human actions categories. The predicting capabilities of the HMMs are used by Dockstader et. all in [11]: the problem of the detection and prediction of motion tracking failures with application in human motion and gait analysis is presented. The approach defines a failure as an event and uses the output probability of a trained HMM to detect and a logarithmically transformed probability to predict such events. The vector observations for the model are derived from the time-varying noise covariance matrices of a Kalman filter that tracks the parameters of a structural model of the human body. A medical application of HMM is presented in [26] where the hidden Markov models are used to model ECG signals, [6] presents an original HMM approach for online beat segmentation and classification of electrocardiograms.

The HMM framework has been visited because of its ability of beat detection, segmentation and classification, highly suitable to the electrocardiogram problem. The same author published [5] which originally combines HMM and wavelets providing new insights on the ECG segmentation problem. P2P-TV traffic has been modeled using HMM in [17], by proposing a simple traffic model that can be representative of P2P-TV applications. HMM is often used for deviation detection and state diagnosis; [10] uses the Hidden Markov Model formalism considering three aspects involved in component's state diagnosis: the monitored component, the deviation detection mechanism and the state diagnosis mechanism.

The model In this paragraph the baseline of the model is provided. Interested readers can refer to [32] for further details. In this work we consider the simplest version of Hidden Markov Model, i.e., a bivariate stochastic process $\{s_t, o_t\}_{t>0}$, which consist of:

a) an unobserved sequence (hidden state process) $\{s_t\}_{t>0}$, which is an homogeneous first order Markov chain, with state space $\Omega = \{\omega_1, \ldots, \omega_N\}$, initial probability vector $\pi(0)$ such that the generic element $\pi_i(0)$ is defined as

$$\pi_i(0) := p(s_0 = \omega_i), \quad \forall i = 1, \ldots, N$$

and the transition matrix A whose elements are:

$$a_{i,j} := p(s_t = \omega_j | s_{t-1} = \omega_i), \qquad \forall i, j = 1, \ldots, N.$$

b) the observed sequence $\{o_t\}_{t>0}$, that is conditioned to$\{s_t, o_t\}_{t>0}$ forms a sequence of conditionally independent random variables, with values in $\Sigma = \{\sigma_1, \ldots, \sigma_M\}$, characterized by the matrix B, whose elements are:

$$b_k(\sigma_j) := p(o_t = \sigma_j | s_t = \omega_k), \quad \forall k = 1, \ldots, N, \, \forall j = 1, \ldots, M.$$

Those elements represents the probabilities to emit a symbol σ_j at time t, given that the state of the Markov chain is ω_k. Each symbol of the alphabet Σ can be emitted, according to a given probability, if the Markov model is in state ω_k.

This paper will target homogeneous first-order Markov chain that is completely characterized by the triple $(A, B, \pi(0))$.

3.2 Complex Event Processing

Detecting event patterns, sometime referred to as situations, and reacting to them are at the core of Complex Event Processing (CEP) [28]. Business intelligence, air traffic control, collaborative security, complex system software management are examples of applications built using CEP technology [14]. CEP is defined as an event processing that aggregates and correlates data from multiple (possibly heterogeneous) sources to infer high level events or patterns. In CEP, three fundamental concepts are defined: event streams, correlation rules, and event engine. An event is a representation of a set of conditions in a given time instant. Events belong to streams: events of the same type are in the same event stream. Correlation rules are commonly SQL-like queries (but also other types of queries are possible [12]) used by the event engine in order to correlate events: i.e., in order to discover temporal and spatial relationships between events of possibly different streams. There are several commercial products that implement the CEP paradigm. Among those currently available, we have chosen the well-known open source event engine ESPER [1]. The use of ESPER is motivated by both its low cost of ownership compared to other similar systems, its offered usability, and the ability of dynamically adapting the complex event processing logic by adding at run time new queries. In ESPER, while the events pass through memory, the query engine continuously sieves for the relevant events that may satisfy one of the queries. The queries are defined using an SQL-like query language named Event Processing Language (EPL). EPL can thus support all SQL's conventional constructs such as Group By, Having, Order By, Sum, etc. However, it adds further constructs (e.g., the `pattern`) that allows it to perform complex correlations among events. CEP engines are usually very optimized in order to grant a very high throughput (hundreds of thousands of events per second) even if deployed in common off-the-shelf computers. The interested reader can refer to [14] for additional detail on how to use, design, and build CEP-based applications.

3.3 Failure and Prediction Model

We model the distributed system to be monitored as a set of nodes that run one or more services. Nodes exchange messages over a communication network. Nodes or services are subject to failures. A failure is defined as an event that occurs when the delivered service deviates from correct service[7]. A failure is always preceded by a fault (e.g., I/O error, memory misusage); however, the vice versa might not be always true, i.e., a fault inside a system

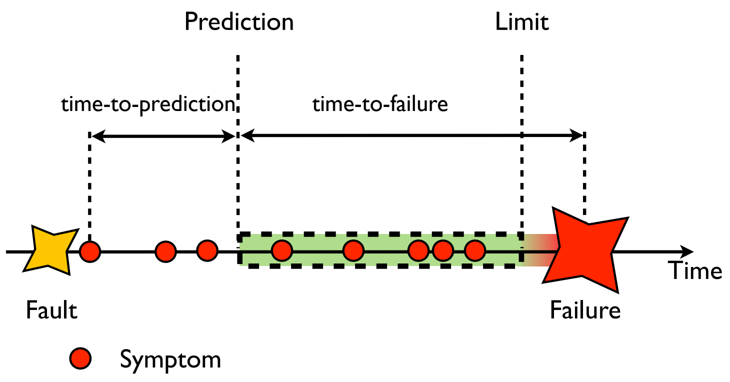

Fig. 1: Fault, Symptoms, Failure and Prediction

could not always bring to a failure as the system could tolerate, for example by design, such fault.

Faults that lead to failures, independently of the fault's root cause (e.g., an application-level problem or a network-level fault), affect the system in an observable and identifiable way. Thus, faults can generate side-effects in the monitored systems until the failure occurs (see Figure 1). Our work is based on assumptions similar to [23, 37, 38] i.e., (i) a fault generates unstable performance-related symptoms indicating a possible future presence of a failure, and (ii) the system exhibits a steady-state performance behavior with a few variations when a non-faulty situation is observed.

A failure prediction mechanism consists in monitoring the behavior of the distributed system looking for possible symptoms generated by faults, and in raising timely alerts regarding software failures if symptoms become severe. Proper countermeasures can be set before the failure to either mitigate damages or enable recovery actions. Figure 1 shows the *time-to-failure* as the distance in time between the occurrence of the prediction and the software failure. The prediction has to be raised before a *Limit*, beyond which it is not sufficiently in advance to take some effective actions before the failure occurs. Finally, the *time-to-prediction* represents the distance between the occurrence of the first symptom of the failure and the prediction. An ideal failure prediction system would produce zero false positives, i.e., no mistakes in raising alerts regarding failures, maximizing at the same time *time-to-failure*.

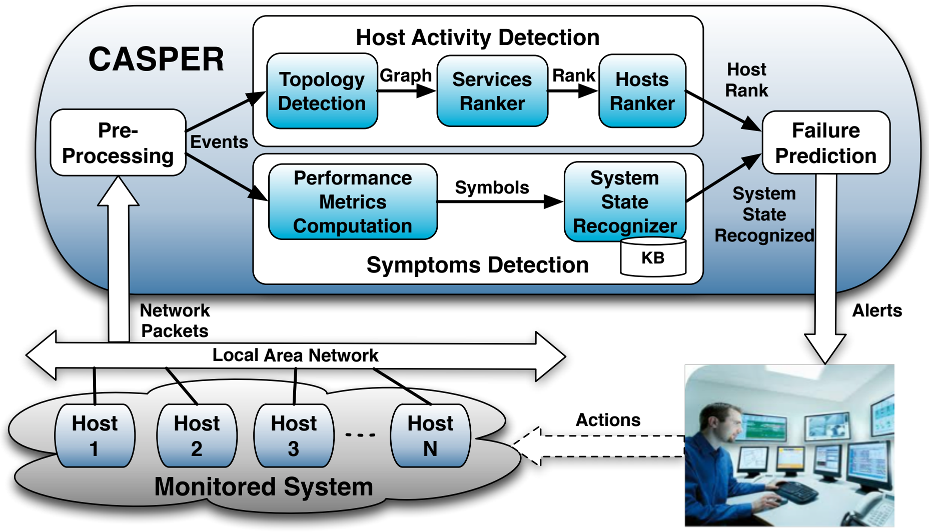

Fig. 2: The modules of the CASPER failure prediction architecture

4 Casper Architecture

The architecture we designed for online black-box non-intrusive failure prediction is named CASPER and it is deployed in the same subnetwork as the distributed system to be monitored. Figure 2 shows the main modules of CASPER. Next subsections provide a description of each of such modules. CASPER takes as input network packets flow, detects in parallel if the system is behaving correctly and if there are some hosts that is experiencing problems. Such information, coming from these two modules is correlated and alerts are sent to an operator. Activities of CASPER architecture are synchronized through a clock mechanism. At the beginning of the clock cycle events fed to the Host Activity Detection and Symptom Detection modules and alerts are generated at the end of the clock cycle. In the experimental validation we set the clock cycle to 800ms finding a good trade off between performances and accuracy[2].

[2] Details about the experimental validation for tuning the clock period (that is not considered in this work) can be found in the following technical report`http://www.dis.uniroma1.it/~midlab/articoli/MidlabTechReport3-2012.pdf`

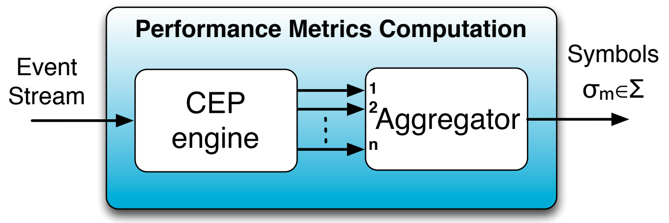

Fig. 3: Performance Metrics Computation component

4.1 Pre-Processing module

It is mainly responsible for capturing and decoding network data required to recognize symptoms of failures and for producing streams of events. The streams of events carry a large amount of information that is obtained from the entire set of network packets exchanged among the interconnected nodes of the monitored system. The Pre-Processing module receives as input network data exchanged by the hosts of the monitored system. More specifically, the pre-processing module extracts headers of network packets such as TCP/UDP headers, SOAP, GIOP, etc. The rationale is to extract from packets only the information that is relevant for the detection of specific symptoms (e.g., the timestamp of a request and of a reply, destination and source IP addresses). Finally, the Pre-Processing module transform protocol headers into an *events* stream that become input of both the Host Activity detection module and the Symptoms detection module.

4.2 Symptoms detection module

The stream of events received as input by the Symptoms detection module is used to discover specific performance patterns through complex event processing (i.e., event correlations and aggregations). The result of this processing is a system state that must be evaluated in order to detect whether it is a safe or unsafe state. To this end, we divided this module into two different components, namely a *Performance Metrics Computation* component and a *System State Recognizer* component.

4.2.1 The Performance Metrics Computation component

This component embodies a CEP engine and an Aggregator. It periodically (e.g., once per second) produces as output a representation of the system behavior in the form of *symbols* taking as input the event stream, (see Figure 2).

The CEP engine computes a vector $\mathbf{V} \in \mathbb{R}^N$ of *performance metrics*, i.e., a set of time-changing metrics whose combination indicates how the system is behaving (an example of network performance metric can be the round trip time). A vector $\mathbf{V}$ of performance metrics is produced per each clock cycle and is provided as input to the *Aggregator*. The Aggregator is a software component that takes in input a vector $\mathbf{V} \in \mathbb{R}^N$ and gives in output a single value in an interval $A \subset \mathbb{N}$, with $|A| = M$ (see Figure 3).

Aggregator internals The Aggregator works using a fix square grid of $\mathbb{R}^N$ constituted by $M = D^N$ $N-$dimensional intervals, where D is the number of 1-dimensional intervals per each component of $\mathbb{R}^N$. The N-dimensional intervals are numbered from 1 to M. Figure 4 represents an example of the defined square grid in $\mathbb{R}^2$ with $M = 16$. In order to perform its task,

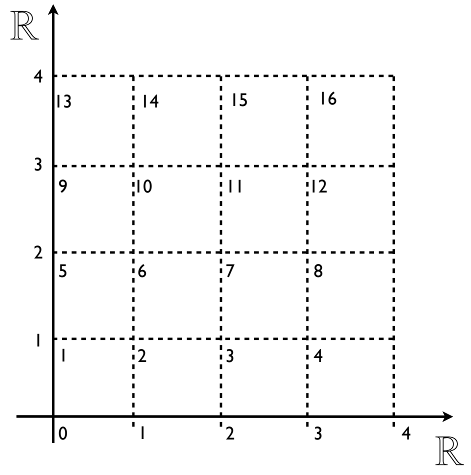

Fig. 4: Example of square grid in $\mathbb{R}^2$ with $D = 4$.

the aggregator needs to know a maximum and a minimum value of the components of $\mathbf{V}$. These values have to be provided during a configuration phase. Starting from $\mathbf{V} = (v_1, v_2, \ldots, v_N)$ the aggregator defines a new vector $\mathbf{V}_{normalized} = (v'_1, v'_2, \ldots, v'_N)$ as follows:

$$v'_i := \frac{D(v_i - v_i^{Min})}{v_i^{Max} - v_i^{Min}}, \quad \forall i = 1, \ldots, N. \tag{1}$$

The (1) simply normalizes each component v_i (according to its maximum and minimum values) in order to have $v'_i \in [0, D]$, for all i. This means that $\mathbf{V}_{normalized}$ identifies a point in the square grid defined before. Each component of $\mathbf{V}_{normalized}$ is now compared with the intervals of the grid in order to identify the $N-$dimensional interval $\mathbf{V}_{normalized}$ belongs to. The number of the interval will be the symbols σ_m.

Consider an example to better explain the Aggregator work: Assume an aggregator in $\mathbb{R}^2$ tuned with a square grid of $M = 16$ 2-dimensional intervals. This means that there are $D = 4$ 1-dimensional intervals per component. Assume also that $v_1^{Min} = v_2^{Min} = 0$ and $v_1^{Max} = v_2^{Max} = 100$ Consider an input vector $\mathbf{V} = [51.34, 58.22]$. Applying Equation 1, we have:

$$\mathbf{V}_{normalized} = (\frac{4(51.34 - 0)}{(100 - 0)}, \frac{4(58.22 - 0)}{100 - 0}) = (2.05, 2.034).$$

The vector $\mathbf{V}_{normalized}$ is shown in Figure 5. A comparison with the intervals boundary can easily identify the 2-dimensional interval which the point belongs to. In this case the interval is the number 11.

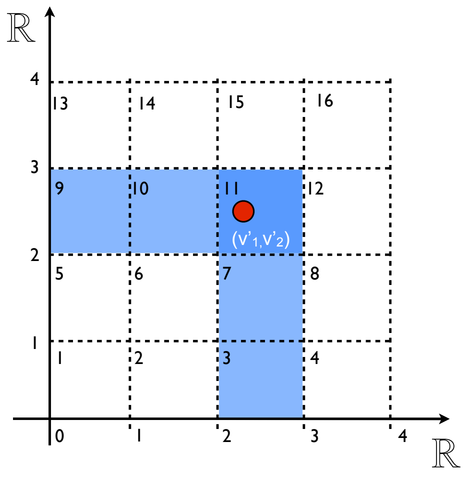

Fig. 5: Example of aggregator behavior with $N = 2$ and $D = 4$.

4.2.2 System State Recognizer component

This component receives a symbol from the previous component at each CASPER clock cycle and recognizes whether it represents a correct or an incorrect behavior of the monitored system. To this end, the component uses Hidden Markov Models. We recall that HMM consists of a *hidden stochastic process*, a set of *symbols* Σ and two probability matrices A and B as defined in Section 3.1. Figure 6 shows how we instantiated HMM in our architecture.

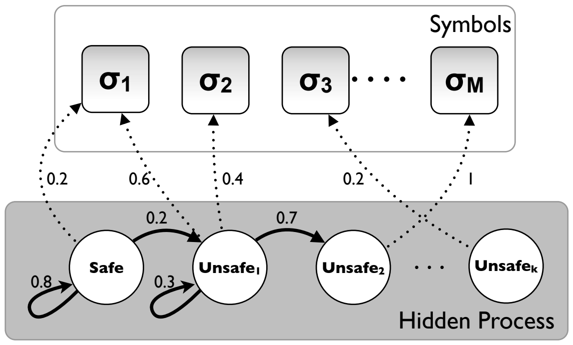

Fig. 6: Hidden Markov Models graph used in the System State Recognizer component

We model the state of the system to be monitored by means of the *hidden process*. We define the states of the system (see Figure 6) as:

- *Safe*: the system behavior is correct as no *active fault* ([7]) is present;
- *Unsafe$_i$*: a fault of type i, and then related symptoms, is present. We assume a finite number of k types of faults (e.g., memory stress, disk overload, cpu overload).

The number of states N is the sum of the k *Unsafe* states and the *Safe* state: $N = k + 1$. We assume that initially the system is in the Safe state:

$$\pi_i(0) = 1.$$

Since the state of the system is not known a priori, we can observe it only looking at the *emissions* of symbols. Figure 6 represents the emitted symbols as the set of $\{\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_M\}$. In addition, Figure 6 shows labeled edges among the vertices of the hidden process; these represent the values of the A matrix. In contrast, the edges that connect the states of the hidden process to the symbols are the probabilities to emit a given symbol σ_m, that is, the B matrix of HMM. Note that the values in Figure 6 are example values, according to them, if we assume that the system is in *Unsafe*$_2$ state, the probability to see the σ_M symbol emitted is 1.

4.3 Hosts Activity Detection Module

The symptoms detection module analyzes the observed distributed system as a single component. Therefore, the recognized system state represents the state of the whole distributed system: nothing can be stated regarding the single node. The Host Activity Detection Module (HADM) handles this problem by providing a periodic snapshot of the nodes health status. In particular, the HADM allows to:

- disclose the network topology of the observed system in a completely non-intrusive fashion;
- create a ranking among the network level services based on their regularity in terms of network activity, i.e. the mean number of messages produced in a given temporal window;
- create a ranking of the network nodes based on the *per-host* services ranking.

In order to perform these actions, the HADM uses as input the event stream produced by the pre-processing module. The HADM architecture embodies three components namely *Topology Detector Component, Service Ranker* and *Hosts Ranker*. A description of these components is now provided.

4.3.1 Topology Detector Component

Since each event received by HADM represents a network packet exchanged by two hosts using a given port number, it is easy to represent all the interactions among the hosts using a graph. The aim of this component is to provide a representation of the network topology of the system observed that is updated in real-time. Each host is represented as a vertex of a graph. Each logical link between two hosts is represented as a directed edge between two vertices. A logical link (edge) represents a connection between two hosts using a source port and a destination port. This idea is similar to the one used by iLand middleware [18] that supports also monitoring of resource consumption and deadline fulfillment of time sensitive operations. The graph takes the time dimension into account by considering an edge has two states, namely active and inactive. An active edge is logical link that experienced at least a message exchange during last CASPER clock cycle. Otherwise it is inactive. As far as nodes is concerned, we consider a sink node, a node that only received messages during the last CASPER clock cycle. A node is called source if it only sent messages in the last cycle.

Figure 7 shows the network topology of the SELEX-ES Air Traffic Control system realized through JUNG library [3]. Black nodes represent sink nodes, white nodes sources, while green nodes are neither sink or source nodes. Dotted edges are inactive network links.

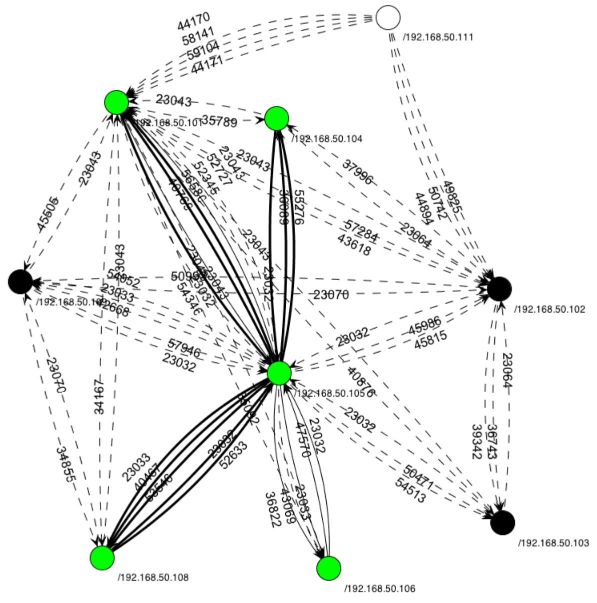

Fig. 7: An example of graph representing a 9-nodes Air Traffic Control system. At the given CASPER clock cycle, the system has 3 sink (black) nodes, 1 source (white) node and 4 green nodes.

[3] A video of the output produced in real-time by this module during its functioning is available at the following link `www.cis.uniroma1.it/projects/casper.php`

4.3.2 Services Ranker component

The Service Ranker Component takes as input a live updated network topology graph generated by HADM and extracts information about services running at each host looking at each graph node and the label of each outgoing edge. Then the component assigns periodically to each service a vote based on its regularity in message exchange. The Services Ranker component outputs a real-time ranking of the liveness of all services of the monitored system.

The vote is based on a mathematical function of the service outer message rate. CASPER computes if the average message rate of a service is near its historical rate. In the affirmative the vote is decreased, otherwise it is increased. To compute the historical rate CASPER uses the exponential moving average, also known as *exponentially weighted moving average* (E.M.A.) defined as follows:

$$E.M.A. := \begin{cases} S_1 = Y_1 \\ S_t = \alpha Y_t + (1-\alpha)S_{t-1} \quad t > 1. \end{cases}$$

where the coefficient α represents the degree of weighted decrease, a constant smoothing factor between 0 and 1. A higher α discounts older observations faster. Y_t is the observation at a time period t. S_t is the value of the E.M.A. at any time period t.

There are two adjustable parameters in this ranking feature: (i) the deviation (how much the Y_t is far from the average) and (ii) α. Both parameters can be tuned in real-time using the CASPER GUI[4].

4.3.3 Hosts Ranker Component

Host Ranker component takes the output of the Services Ranker component as input and computes a ranking among the hosts. This ranking is used as an input by the Failure Prediction Module that correlates this stream with the one emitted by the Symptoms Detection module.

The hosts ranker component first of all computes a per-host vote from input received by the service rank component by summing up votes of all services of a host. This vote is then multiplied by the *request over replies* ratio forming the final host's vote. This ratio represents, for each host, the number of requests sent to that host during a CASPER clock cycle over

[4] CASPER GUI in action can be seen at the following link `www.cis.uniroma1.it/projects/casper.php`

the number of replies produced by the host in the same cycle. The ratio greater than 1 means there is a number of requests not served by the host (probably the host is overloaded). If the host is working well, we expect a ratio that will be very close to 1, thus the final host's vote will be equal to the one obtained by summing up votes of services managed by that host. If the ratio is greater than 1, the final host's vote will get higher, amplifying the host non-regularity.

4.4 Failure Prediction module

This module is mainly responsible for correlating the information about the state received from the System State Recognizer component and from Host Ranker component. It takes as input the recognized state of the system at each CASPER clock-cycle and the host ranking. The recognized state can be a safe state or one of the possible unsafe states $Unsafe_i \dots Unsafe_k$. Using the CEP engine, this module counts the number of consecutive $Unsafe_i$ states and produces a failure prediction alert when that number reaches a tunable threshold. The alert will also contain the ranking of the hosts so the operator can have an overview of the single hosts, in the moment of the prediction.

5 Training of the model

The knowledge base (see Figure 2) concerning the possible safe and unsafe monitored system states is composed by the matrices A and B (see Section 3.1) of the System State Recognizer component. This knowledge is built during an initial training phase. If the A matrix represents how the system behaves, the B matrix represents what we can see about the system behavior. Adjust the entries of these matrices is the solution to the known *training problem* of the HMM. There is no known way to solve for a maximum likelihood model analytically; moreover solve this problem without knowledge about the path of system states is a NP Complete problem and it can only be approximate using complex heuristics. Nevertheless, if the path of hidden states that generates the observations is known, the parameters of the matrices can be computed using the maximum likelihood re-estimation technique [32].

We followed a supervised learning approach, that is, we considered the case in which the sequence of hidden states is known.This choice is motivated by two main reasons: The first one is performance of the learning phase: the system has been designed in order to be deployed in a critical environment, in which a new anomalous condition has to be learnt in near real time, so unsupervised algorithms (e.g., Baum-Welch or Viterbi algorithm), usually computationally too slow, have not been considered. The second reason is that, due to the complexity of the observed system, the effects of faulty conditions can be unpredictable, thus it is very important to recognize the type of fault that caused the deviation. This can be effectively obtained with a supervised learning approach, where the system is trained with sequences of faulty observations, that we extracted from the testing environment. Thus, in our case, parameters of the HMM can be computed as follows, according to methodology presented in [33]:

- the probabilities π_i are determined by the relative frequency of sequences starting in state ω_i:

$$\pi_i = \frac{\text{number sequences starting in } \omega_i}{\text{total number of sequences}};$$

- the elements of the A matrix are determined by the number of transitions from ω_i to ω_j divided by the total number of transition from the ω_i state:

$$a_{ij} = \frac{\text{number of transitions}(\omega_i, \omega_j)}{\text{number of transitions}(\omega_i, \omega_k)\forall\omega_k \in \Omega};$$

- the elements of the B matrix are determined by the number of times the process has generated symbol σ_j (i.e. an emission of σ_j) being in the state ω_i divided by the number of time the process has been in the state ω_i:

$$b_i(o_j) = \frac{\text{number of } \sigma_j \text{ emissions}}{\text{number of times the process has been in state} \omega_i}.$$

Having the sequences of states known, what described is sufficient to build the A and B matrices and the π_i as well.

In order to have the hidden state sequences, during the training CASPER is fed concurrently by both recorded network traces and a sequence of pairs `<system-state,time>` that represents the state of the monitored system (i.e., $Safe, Unsafe_1, \ldots, Unsafe_k$) at a specific time. Since the training is offline, the sequence of pairs `<system-state,time>` can be created offline by the operator using network traces and system log files. No training is required for the other components of CASPER.

6 Results

We deployed and tested CASPER failure prediction capabilities in a real Air Traffic Control system owned by Selex-ES that manufactures and manage ATC systems. We worked on an ATC system in production and on a perfect copy of the system deployed in a testing environment.

We first collected a number of network traces from the ATC representing steady state performance behavior and stress conditions leading to software failures. The former traces have been collected in the production environment. The latter ones have been collected in the testing environment where we injected memory and I/O stress in one of the nodes of the ATC system.

After the collection of the traces, we trained CASPER and once the training phase was over we deployed CASPER on the testing environment of the ATC system in order to conduct our experiments campaign. We studied (i) the CASPER accuracy in detection of the state of the monitored system and (ii) the CASPER capability to predict a failure caused by these conditions. The implementation of all CASPER components has been done in Java.

6.1 HMM and Aggregator parameters

During the experimental campaign we had to choose the number of hidden states of the HMM model (and what each of them represents) and to tune parameters of the Aggregator (see Section 4.2.1), i.e., number of symbols $M = D^N$, the possible values per each performance metric D, and the number of performance metrics N.

In particular, for the HMM, we considered a total of three hidden states: $Safe$, $Unsafe_1$, $Unsafe_2$ according to the two stress condition (memory and I/O) that we injected in the system.

We considered $N = 3$ performance metrics, in particular: round trip time (average time between the requests and the relative replies within a clock cycle), message rate (average number of messages per second of the clock cycle) and number of requests without reply (number of requests that are still waiting for a reply within a clock cycle). These metrics are computed by the CEP engine and produced one time per CASPER clock cycle, as vector $\mathbf{V}$ entries (see Section 4.2.1).

The number of emissions is $M = |\Sigma| = 216$ symbols and is due to the number of performance metrics considered $N = 3$ and the possible values per each performance metric D. In this campaign we chosen $D = 6$. These parameters have been chosen empirically, after a campaign of experiments[5].

6.2 Results of CASPER failure prediction

We have run two campaigns of experiments once CASPER was trained and tuned. In the first one, we injected faults in the ATC testing environment and we carried out 8 tests for each type of fault. In the second one, we observed the accuracy of CASPER when monitoring for 24h the ATC system in operation. From the point of view of the accuracy, we found during the tests of the first campaign the mean results reported in Tab. 1, where N_{tp} (number of true positives) means the system state is unsafe and the recognized state is "system unsafe"; N_{tn} (number of true negatives): the system state is safe and the recognized state is "system safe"; N_{fp} (number of false positive): the system state is safe but the recognized state is "system unsafe"; and N_{fn} (number of false negatives): the system state is unsafe but the recognized state is "system safe".

Tab. 1: Accuracy metrics.

Precision: $p = \frac{N_{tp}}{N_{tp}+N_{fp}}$	88.51%
Recall (TP rate): $r = \frac{N_{tp}}{N_{tp}+N_{fn}}$	76.47%
F-measure: $F = 2 \times \frac{p \times r}{p+r}$	82.05%
FP Rate: $f.p.r. = \frac{N_{fp}}{N_{fp}+N_{tn}}$	11.26%

In order to evaluate the ability of CASPER of predict failures due to memory stress, we injected this kind of stress in one of the node of the ATC system until a service failure. Figure 8 shows the anatomy of this failure in one of the tests. In Figure can be appreciated how CASPER have run with some wrong recognized state (false positive) until the time the memory stress starts at second 105. The sequence of false positives starting at second 37 is not sufficiently long to create a false prediction. After that memory stress starts, the failure prediction module outputs a prediction at second 128; thus, the time-to-prediction is 23s. The failure occurs at second 335, then the time-to-failure is 207s, which is satisfactory with respect to ATC system recovery requirements. We can also see a little burst of system state inference component false negatives starting at second 128, successfully ignored by the failure prediction module.

[5] Details about the accuracy of CASPER varying D are available in the technical report: `http://www.dis.uniroma1.it/~midlab/articoli/MidlabTechReport3-2012.pdf`

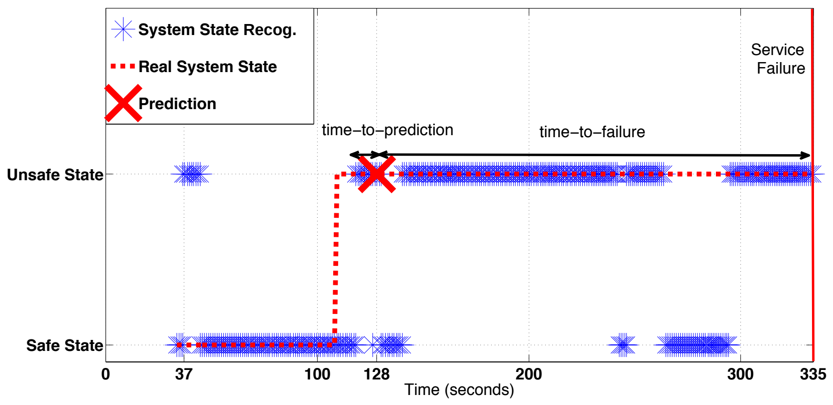

Fig. 8: Failure prediction in case of memory stress starting at second 105.

Figure 9 shows the anatomy of the failure in case of I/O stress in one test. A failure caused by I/O stress happens after 408 seconds from the start of the stress (at 190s) and has been predicted at time 222 after 32s of stress, with a time-to-prediction equal to 376s before the failure. There is a delay due to the false negatives (from 190s to 205s) that the system state inference component produced at the start of the stress period. The time-to-prediction is 21s.

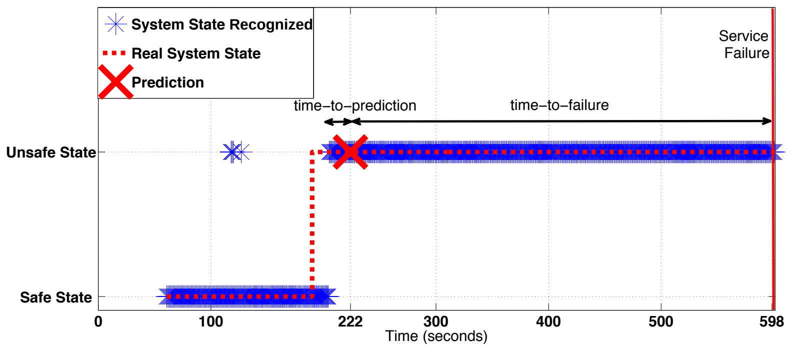

Fig. 9: Failure prediction in case of I/O stress starting at second 408.

In general, we obtained that in the 8 tests we carried out, the time-to-failure in case of memory stress varied in the range of [183s, 216s] and the time-to-prediction in the range of [20.8s, 27s]. In case of I/O stress, in the 8 tests, the time-to-failure varied in the rage of [353s, 402s] whereas the time-to-prediction in the range of [19.2s, 24.9s]. Note that the time-to-prediction can be influenced by the stress tools used, thus they have been accurately tuned to match the real faulty situations the ATC system usually suffers.

6.3 Results of CASPER hosts ranking

The host ranking is a real time ranking among the hosts. Figure 10 represents a situation in which, at second 128, a memory stress application has been run in host 102 until its failure. The regularity of this host is highly affected by the stress. The host ranking module recognizes this fact and assigns, to host 102, a higher and higher vote. Also the other hosts have suffered the misbehavior of host 102 thus implying the increase of the related votes. At 270s host 102 halted and its vote starts to increase linearly. The failure prediction has been triggered at 191s.

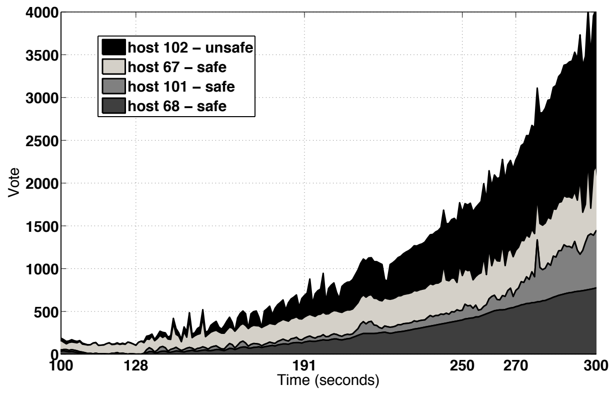

Fig. 10: Hosts ranking behavior. At second 128 a memory stress starts on host number 102.

7 Conclusion

We presented an architecture to online predict failures of safety-critical distributed systems. The failure prediction architecture, namely CASPER, provides accurate predictions of failures by exploiting only the network traffic of the monitored system. In this way, it results non intrusive with respect to the nodes hosting the safety-critical system and it executes a black-box failure prediction as no knowledge concerning the layout and the logic of the safety-critical distributed system is used. To the best of our knowledge, this is the first failure detection system exhibiting all these features together. Such failure prediction mechanisms sense anomalous behaviors that propagate through software applications and services forming the safety-critical systems. Results have shown that CASPER exhibits pretty good accuracy and it is able to generate predictions with a margin of time that allows recovery actions to mitigate the upcoming occurrence of a failure of the system.

In the last couple of years, researchers started investigating how to manage and predict failures in complex infrastructures like cloud computing ones [24, 20, 29]. We are currently studying how to specialize CASPER to cloud computing infrastructures. Specifically, we are using network traffic and power consumption data, taken from a datacenter enclosure, as input, for showing how the prediction accuracy of failures, occurring inside the enclosure, can be improved through the correlation of those inputs. Preliminary results have been published in [8].

Acknowledgments

The authors would like to thank Luca Iocchi for comments and suggestions on the HMM internals and the anonymous reviewers for their insightful comments that greatly improved content and presentation of this paper. This work has been partially supported by the TENACE project (MIUR-PRIN 20103P34XC), by the CINI-SELEX-ES project "Iniziativa Software" and by the EU project PANOPTESEC.

References

[1] Esper project web page, 2012. `http://esper.codehaus.org/`.

[2] Hesham J Abed, Ala Al-Fuqaha, Bilal Khan, and Ammar Rayes. Efficient failure prediction in autonomic networks based on trend and frequency analysis of anomalous patterns. *International Journal of Network Management*, 23(3):186–213, 2013.

[3] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 74–89, New York, NY, USA, 2003. ACM.

[4] Jonathan Alon, Stan Sclaroff, George Kollios, and Vladimir Pavlovic. Discovering clusters in motion time-series data. In *In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 375–381, 2003.

[5] Rodrigo Varejão Andreão and Jérôme Boudy. Combining wavelet transform and hidden markov models for ecg segmentation. *EURASIP J. Appl. Signal Process.*, 2007(1):95–95, January 2007.

[6] R.V. Anreão, B. Dorizzi, and J. Boudy. Ecg signal analysis through hidden markov models. *In IEEE Transactions on Biomed. Eng.*, 53(8):1541–9, August 2006.

[7] A. Avizienis, J. Laprie, B.Randell, and C.E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[8] Roberto Baldoni, Adriano Cerocchi, Claudio Ciccotelli, Alessandro Donno, Federico Lombardi, and Luca Montanari. Towards a non-intrusive recognition of anomalous system behavior in data centers. In *Computer Safety, Reliability, and Security*, pages 350–359. Springer, 2014.

[9] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

[10] A. Daidone, F. Di Giandomenico, A. Bondavalli, and S. Chiaradonna. Hidden Markov models as a support for diagnosis: Formalization of the problem and synthesis of the solution. In *Proceedings of 25th IEEE Symposium on Reliable Distributed Systems (SRDS 2006)*, pages 245–256, Leeds, UK, October 2006.

[11] Shiloh L. Dockstader, Nikita S. Imennov, and A. Murat Tekalp. Markov-based failure prediction for human motion analysis. In *Proceedings of the Ninth IEEE International Conference on Computer Vision - Volume 2*, ICCV '03, pages 1283–, Washington, DC, USA, 2003. IEEE Computer Society.

[12] Bugra Gedik Henrique Andrade Kun-Lung Wu Philip S. Yu MyungCheol Doo. Spade: The system s declarative stream processing engine. In *Proceedings of ACM SIGMOD international conference on Management of data*, Vancouver, BC, Canada, June 9–12 2008.

[13] Sean R. Eddy. Profile hidden markov models. *Bioinformatics*, 14(9):755–763, 1998.

[14] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[15] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[16] Song Fu and Cheng zhong Xu. Exploring event correlation for failure prediction in coalitions of clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC07)*, 2007.

[17] Maria Antonieta Garcia, Ana Paula Couto da Silva, and Michela Meo. Using hidden markov chains for modeling p2p-tv traffic. In *GLOBECOM*, pages 1–6, 2010.

[18] M. Garcia Valls, I.R. Lopez, and L.F. Villar. iland: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems. *Industrial Informatics, IEEE Transactions on*, 9(1):228–236, Feb 2013.

[19] Marisol García Valls and Pablo Basanta Val. A real-time perspective of service composition: key concepts and some contributions. *Journal of Systems Architecture*, 59(10):1414–1423, 2013.

[20] Qiang Guan, Ziming Zhang, and Song Fu. Ensemble of bayesian predictors and decision trees for proactive failure management in cloud computing systems. *Journal of Communications*, 7(1):52–61, 2012.

[21] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles &Amp; Practice of Parallel Programming*, PPOPP '90, pages 197–206, New York, NY, USA, 1990. ACM.

[22] Günther A. Hoffmann, Felix Salfner, and Miroslaw Malek. Advanced Failure Prediction in Complex Software Systems. Technical Report 172, Department of Computer Science, Humboldt-Universität zu Berlin, Germany, 2004.

[23] C.S. Hood and C. Ji. Proactive network-fault detection. *In IEEE Transactions on Reliability*, 46(3):333 –341, september 1997.

[24] Ravi Jhawar, Vincenzo Piuri, and Marco Santambrogio. Fault tolerance management in cloud computing: A system-level perspective. *Systems Journal, IEEE*, 7(2):288–297, 2013.

[25] R. Khanna and Huaping Liu. Control theoretic approach to intrusion detection using a distributed hidden markov model. *Wireless Commun.*, 15(4):24–33, August 2008.

[26] Antti Koski. Modelling ecg signals with hidden markov models. *Artif. Intell. Med.*, 8(5):453–471, October 1996.

[27] Anders Krogh, Michael Brown, I. Saira Mian, Kimmen Sjölander, and David Haussler. Hidden markov models in computational biology: applications to protein modeling. *Journal of Molecular Biology*, 235:1501–1531, 1994.

[28] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[29] Reza Farrahi Moghaddam, Fereydoun Farrahi Moghaddam, Vahid Asghari, and Mohamed Cheriet. Cognitive behavior analysis framework for fault prediction in cloud computing. In *Network of the Future (NOF), 2012 Third International Conference on the*, pages 1–8. IEEE, 2012.

[30] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, April 1980.

[31] Teerat Pitakrat. Hora: Online failure prediction framework for component-based software systems based on kieker and palladio. In *KPDAYS*, pages 39–48, 2013.

[32] L. Rabiner and B. Juang. An introduction to hidden markov models. *ASSP Magazine, IEEE*, 3(1):4 – 16, jan 1986.

[33] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *Proceedings of IEEE*, volume 77, pages 257–286. IEEE, 1989.

[34] F. Salfner. *Event-based Failure Prediction: An Extended Hidden Markov Model Approach*. PhD thesis, Department of Computer Science, Humboldt-Universität zu Berlin, Germany, 2008.

[35] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Computing Surveys (CSUR)*, 42, 2010.

[36] Yongmin Tan, Xiaohui Gu, and Haixun Wang. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 173–182, New York, NY, USA, 2010. ACM.

[37] Marina Thottan and Chuanyi Ji. Properties of network faults. In *Proceeding of IEEE/IFIP Network Operation and Management Symposium (NOMS 2000)*, pages 941–942, 2000.

[38] Andrew W. Williams, Soila M. Pertet, and Priya Narasimhan. Tiresias: Black-box failure prediction in distributed systems. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Los Alamitos, CA, USA, 2007.

[39] J. Yamato, J. Ohya, and K. Ishii. Recognizing human action in time-sequential images using hidden Markov model. *Proceedings 1992 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 379–385, 1992.