Evaluation of Software Architecture Alternatives Using Survivability and Software Rejuvenation Modeling

Gabriela Jacques-Silva*, Alberto Avritzer¶, Daniel S. Menasché‡, Anne Koziolek§, Lucia Happe§, Sindhu Suresh¶
*IBM T. J. Watson Research Center, USA
¶Siemens Corporation, Corporate Technology, USA
§Universidade Federal do Rio de Janeiro, Brazil
¶Karlsruhe Institute of Technology, Germany

Abstract—In this paper we apply survivability and software rejuvenation modeling to evaluate alternative software architectures. We analyze failure history in two large industrial projects and propose an unified failure model to be used for the assessment of system survivability at the software architecture phase. Our goal is to assess the mean time to repair a system, conditioned that it starts from a failure prone state. To this aim, we use the failure model as one of the components of an analytical survivability framework which yields the desired metric of interest. The framework comprises a phased-recovery model and a software rejuvenation model.

As a case study, we consider a distributed large industrial data streaming system which contains system-wide and host-specific monitors that are subject to failures. We instantiate the proposed framework to analyze the considered case study. Numerical investigation allows us to quantify the impact of architecture deployment alternatives, phased recovery and software rejuvenation on the mean time to repair.

I. INTRODUCTION

Careful development of software architectures has been recognized for some time as an important factor in the creation of successful software systems [1]. However, finding the correct tradeoff between software architecture alternatives continues to elude designers [2], as the likelihood of occurrence of software architecture related failure events is hard to estimate early in the project lifecycle [3].

In this paper, we propose to use a survivability modeling and software rejuvenation approach to help assess the tradeoffs incurred in the selection of software architecture deployment alternatives. We propose to use survivability modeling to assess the impact of hard failures and to use software rejuvenation to pro-actively recover from soft failures.

We define a hard failure as a failure that causes system unavailability, as it leads to a system crash. A hard failure recovery may take significant time to be implemented, as it may require system verification (e.g. disk check) and repair. A soft failure is defined as a failure that causes system performance degradation, as for example, due to excessive resource consumption. A soft failure recovery is usually fast and might involve the restart of a process or the release of a resource.

In this approach, the metric of interest to software architecture design is the time to recover from the specific software failures being modeled. Therefore, in this analysis, we focus on the evaluation of software architecture alternatives to enable recovery from specific failures as defined by the failure model. The failure model maps failures into hard failures and soft failures. In addition, the failure model contains failure modes that are amenable to pro-active software rejuvenation strategies.

The hierarchical modeling approach to support survivability and rejuvenation modeling is composed of several models including:

1) a failure model that provides the probability of different sorts of soft and hard failures occurring given that the system is prone to failure. In this paper we propose an unified failure model that was developed based on performance and reliability studies of several large industrial projects [4], [5], [6];

2) a survivability model that represents phased-recovery conditioned on the occurrence of hard-failures that are identified as being of interest to the architecture design. This model accounts for recovery from different types of failures, as for example, failures of processing elements, data elements and network failures;

3) a rejuvenation model that characterizes pro-active scanning for soft-failures that can be addressed using software rejuvenation strategies. Early detection of potential failures allows for quick fixes.

Once the proposed methodology comprising the three models above is deployed, software architects are able to use a survivability-oriented approach to systematically compare the cost/benefits of using alternative software architectures. To illustrate the applicability of the approach, we consider the architecture of InfoSphere Streams, which is a platform for building distributed stream processing applications, and involves system-wide and host-specific monitors. Then, we quantify how the mean time to recover from failures in InfoSphere Streams varies as a function of different parameters such as the rate at which the system is sampled in search for soft failures and the probability that the system is amenable to automated failover given that a hard failure occurred.
In summary, our main contributions are the following:

1) unified failure model: we categorize failures into six types and divide them into soft and hard failures, indicating different repair strategies that can be used in each case;

2) analytical survivability framework: we use the unified failure model as an input to an analytical survivability framework which combines a survivability model and a rejuvenation model, and yields the mean time to repair a system once it becomes prone to failure;

3) case study analysis: we progressively specialize our framework to capture distributed host monitoring available in InfoSphere Streams Architecture, numerically indicating the impact of different parameters on the mean time to repair.

The outline of this paper is as follows. In Section II we present a brief overview of the failure models for two complex industrial systems, and in Section III we present an unified view of the failure model by combining the models presented in Section II. Section IV presents the metric and models we introduce for survivability assessment of software architectures. In Section V we present the software architecture model of the target data streaming application considered in this paper. Section VI contains the numerical results obtained by our survivability analysis. In Section VII we place our research in the context of the reviewed literature. In Section VIII we present our conclusions and suggestions for further research.

II. Overview of Failure Reports

In this section we present a brief overview of failure reports for two large systems: Hadoop [5], [6] and a large industrial system [4].

Dinu and Ng [5] provided an analysis of Hadoop’s performance under failures. The authors have uncovered instances where a single failure had a significant impact on Hadoop’s performance. One of the major components associated with the performance slowdown was the time required for failure detection and recovery. Furthermore, two other major performance slowdown root causes were identified. First, Hadoop’s speculative execution algorithm was found to be influenced by fast tasks. Second, Hadoop’s tasks do not share failure information, so failure detection and recovery algorithms might be computed repeatedly, even if the failure cause is environmental, as for example a network congestion or database crash. The authors recommend the replacement of static thresholds and timeouts by using adaptive approaches that can dynamically react to environmental changes. In addition, the authors have identified a need to decouple overload recovery from failure recovery. In this paper, we present a failure model that provides a clear separation between soft failures related to environmental changes (e.g. transient issues, cpu overload conditions, network congestion), and hard failures that are related to architecture elements crashes (e.g. processing elements, data elements, and network related elements).

Rabkin and Katz [6] presented a failure taxonomy based on Hadoop’s ticket analysis. The authors have divided the root cause of the ticket report according to the following seven categories:

1) Misconfigurations were issues related to how Hadoop or operating system options were configured during installation.
2) Bug were issues that required a software change (e.g. patch) to be fixed.
3) Operational were issues related to how the system was being operated by the user.
4) System were issues related to the environment and not to Hadoop. Examples of System failures are networking or database issues.
5) Install were issues related to incorrect system installation.
6) User were issues related to user developed software.
7) Hardware were issues related to faulty hardware (e.g., network card failures).

The authors present recommendations to developers on how to avoid some of the problems they have identified.

A significant number of tickets were assigned to misconfiguration, which in this case represents: 1) instances of memory misconfiguration, and, 2) resource exhaustion as a result of mismanagement of files and sockets. One of the key areas for improvement recommended in [6] is the better management of resources like memory and thread allocation. Misconfigurations represented most of the problems analyzed in [6] and were responsible for several of tickets reported in [4] as shown in Table I.

Avritzer and Bondi [4] reported a failure analysis of several hundred tickets generated for a large industrial system. The authors have categorized the failure reports into eight categories as follows:

1) Crash event brings the system to a complete halt.
2) Malfunction event occurs when a component or a set of components do not work as intended.
3) Shutdown failure occurs when the system fails to shutdown correctly.
4) Initialization failure occurs when the system fails to initialize.
5) Incorrect documentation event occurs when the failure is a consequence of incorrect operations documentation.
6) Performance failure occurs when the operation execution does not meet its performance requirements.
7) Software upgrade needed failure occur when the failure can be corrected with a software upgrade.
8) Unhandled exceptions occur when an unplanned condition is encountered that causes an unhandled exception failure in the software.

In the next section, we provide an unified failure model that combines the experiences reported in Hadoop [5], [6] and in a large industrial system [4].

III. Failure Model

In this section we create a unified failure model by leveraging the two failure models presented in Section II. This unified failure model is used to support the creation of the survivability model presented in Section IV, where we associate each type of failure category with the corresponding failure recovery strategy.
The following types of failures are considered in the unified failure model:

1) Crash while in operation
2) Malfunction that does not lead to a crash
3) Crash on shutdown
4) Crash on initialization
5) Performance degradation
6) Unhandled exceptions

We have excluded from the unified failure model the following failure modes that were considered to be out of scope of this analysis or were deemed to be categorized using subjective judgement: hardware, software upgrade needed, all issues related to incorrect system operation, faulty installation, misconfigurations due to faulty documentation, and lack of proper operators training.

We further categorize the failures according to the required failure recovery into hard failures and soft failures.

Hard failures include crash while in operation, crash on shutdown, crash initialization and unhandled exception failure categories. All these failures require the system to go through a failure repair phase that could be manual, partially automated or fully automated.

Soft failures include malfunction that does not lead to a crash and performance degradation. Soft failures can be recovered using a variety of performance and reliability optimization strategies, including retry, software rejuvenation, and dynamic resource allocation.

Figure 1 illustrates the distinction between soft and hard failures, and shows the different actions that might be taken to counteract such failures.

<table>
<thead>
<tr>
<th>Failure type</th>
<th>Correspondence in [4]</th>
<th>Correspondence in [6]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard failures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Crash while in operation</td>
<td>Shutdown</td>
<td>Misconfigurations</td>
</tr>
<tr>
<td>Crash on shutdown</td>
<td>Initialization</td>
<td>Misconfigurations</td>
</tr>
<tr>
<td>Unhandled Exceptions</td>
<td>Unhandled Exceptions</td>
<td>Bug</td>
</tr>
<tr>
<td>Soft failures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance degradation</td>
<td>Performance</td>
<td>System</td>
</tr>
<tr>
<td>Malfunction that does not lead to a crash</td>
<td>Malfunction</td>
<td>Bug</td>
</tr>
</tbody>
</table>

In Table I we show the unified failure model mapping into hard failures and soft failures and its correspondence to the failure models presented in [4] and [6].

In Section IV we introduce the software architecture survivability model that incorporates the unified failure model introduced in this section.

IV. FAILURE, REJUVENATION AND RECOVERY MODELS

Next, we introduce the failure, rejuvenation and recovery models used to assess how the system behaves from a failure-prone state up to full repair. The general framework is presented in Figure 2. System architecture information is used to parametrize the failure model, rejuvenation model and recovery model. The failure model yields the rates of different failure types, and is then used as input to the rejuvenation model and the recovery model, which together yield the metrics of interest.

The recovery model is a phased-recovery model, in which each state characterizes a step in the recovery process (Figure 3). The model accounts for different failure types and associated failure repair measures. The rejuvenation model includes pro-active software rejuvenation measures when the software is in the failure prone state. Notation is introduced in Table II.

In what follows, we discuss further details of the failure model, recovery model and rejuvenation model, before introducing the general model that comprises the three.

![Fig. 2. General failure, rejuvenation and recovery framework.](image)

A. Failure Model

The failure model characterizes how the system behaves after it becomes prone to failure, and is introduced in Section III. Let $p_h$ and $p_s$ be the hard and soft failure probabilities, respectively, $p_h + p_s = 1$.

Let $p_1$ be the probability that a hard failure is amenable to be recovered through a process start. Let $p_2$ be the probability that a hard failure is amenable to be recovered through automated failover (e.g., process migration) but not through a process restart, and let $p_3$ be the probability that the failure is not amenable to be recovered either through automated failover or process restart. Note that $p_1 + p_2 + p_3 = 1$.

Historical data about the rate at which hard failures and soft failures of different types occur allows us to parameterize $p_h$ as well as $p_1$ and $p_2$. 
TABLE II. TABLE OF NOTATION

<table>
<thead>
<tr>
<th>variable</th>
<th>description</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma$</td>
<td>failure rate after prone to failure</td>
<td>1</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>automated repair rate (phased-recovery model)</td>
<td>30</td>
</tr>
<tr>
<td>$\delta$</td>
<td>manual repair rate</td>
<td>0.25</td>
</tr>
<tr>
<td>$\beta_1$</td>
<td>fix rate when not amenable to</td>
<td>1</td>
</tr>
<tr>
<td>$\beta_2$</td>
<td>fix rate when amenable to</td>
<td>2</td>
</tr>
<tr>
<td>$p_{h}$</td>
<td>hard failure</td>
<td>0.9</td>
</tr>
<tr>
<td>$p_{s}$</td>
<td>soft failure</td>
<td>0.1</td>
</tr>
<tr>
<td>$p_{sr}$</td>
<td>successful rejuvenation</td>
<td></td>
</tr>
</tbody>
</table>

B. Recovery Model

A survivability model characterizes how the system evolves given that a failure occurred. Note that the model presented in Figure 3 contains a survivability model composed of states F1, F2, F3 and 0. Given that a hard failure occurred, let $T$ be the time until full repair. In the Appendix, we characterize $T$, which allows us to obtain the mean and the variance of the time until full repair as a function of $p_{1}$ and $p_{2}$.

C. Rejuvenation Model

The system is proactively rejuvenated at rate $\rho$. Rejuvenation is specially effective to prevent and counteract soft failures. With probability $p_{sr}$, rejuvenation is successful and the system transitions to state 0. With probability $1 - p_{sr}$, the system transitions back to the failure prone state. The success probability of rejuvenation must be determined using data from past experiences. Alternatively, one can inject failures at the system and identify how they are mitigated by rejuvenation.

D. Combining Failure, Recovery and Rejuvenation Models

Figure 3 combines the failure, recovery and rejuvenation models. In state P, a node has a problem and is prone to failure. With rate $\gamma$, an actual failure occurs, and the system moves to states SF, F1, F2 or F3, depending on the failure level. State SF corresponds to soft-failures, which occur with probability $p_{s}$, whereas states F1, F2 and F3 correspond to hard-failures, which occur with probability $p_{h}$.

Let us analyze how the system behaves after a hard failure. After a hard failure occurs, the system transitions to one of states F1, F2 or F3. These three states correspond to increasingly severe failures:

1) with probability $p_{1}$, the system transitions to state F1 and the failure is amenable to recovery through process restart. At state F1, automatic restoration occurs at rate $\alpha$;

2) with probability $p_{2}$, the system transitions to state F2 and the failure is amenable to automated failover. At state F2, a quick fix adjust might make the failure amenable to recovery by process restart. A successful quick fix, e.g. process migration, takes mean time $1/\beta_{2}$;

3) with probability $p_{3}$, the system transitions to state F3. The failure is not amenable to automated failover or recovery through process restart. While at state F3, fine grained adjustments might make the system amenable to repair through process restart or automated failover. Let $\tilde{p}_{2}$ be the probability that, immediately after a fine grained adjustment, a node is amenable to automated failover but not process restart. The fine grained adjustment takes mean time $1/\beta_{1}$ and yield a transition to states F1 and F2 with probability $1 - \tilde{p}_{2}$ and $\tilde{p}_{2}$, respectively.

![Fig. 3. Failure, rejuvenation and recovery model. States marked in red comprise a phased-recovery model.](image)

V. TARGET APPLICATION ARCHITECTURE MODEL

To illustrate the applicability of our survivability modeling approach, we chose a telecommunications (telco) application that uses the pipes and filters architecture as a target [7]. In this scenario, network equipments continuously generate Call Data Records (CDRs) based on calls that are going through the system. These CDRs are then written into files, which are consumed by the application as they are generated. The application parses the incoming CDRs and do several transformations and data enrichment operations. The output of the data processing pipeline is written into new files.
Our target application is architected as a stream processing pipeline and implemented in IBM’s InfoSphere Streams real-time processing platform [8]. By using such platform, the application can run continuously, be automatically parallelized, and be distributed over a cluster. It can also leverage the platform fault-tolerance features, which can automatically restart the different components of the application [9].

As the application does file-based processing, it can use file systems designed to achieve fault-tolerance in commodity clusters. One such example is the Hadoop Distributed File System (HDFS) [10], which is available in several Apache Hadoop enterprise distributions [11], [12]. Another example is the MapR file system [13], available in MapR’s Hadoop distribution. This file system provides the same API as HDFS, but has a different architecture than HDFS.

Figure 4 shows the simplified analytics pipeline architecture of our target application. The first module of the application is the File ingestion. This module does periodic directory reads in the file system to find new files to be processed. As it finds a new file, the ingestion module dispatches the file’s name to one parallel copies of the Reader module. The Reader module opens the incoming file, reads its content and sends it to the Transformer. The transformer module is responsible for doing in-memory lookups and analytic-specific transformations on the incoming data (e.g., number of dropped calls in a given cell). The Sink module persists the analytic output to the file system and forwards statistics to the Statistics module. The level of parallelism of the Reader, Transformer, and Sink modules can be provisioned according to load the system needs to handle. A more detailed view of the application architecture can be found at [14].

![Telecommunications application architecture](image)

Fig. 4. Telecommunications application architecture

In Section V-A, we describe the stream architecture of the platform that our application runs on. The alternative file system architectures are described in Section V-B. Understanding the architecture of the underlying platforms enables us to do accurate application modeling, as it allows to understand how failures affect the application and its survivability characteristics.

A. InfoSphere Streams Architecture

IBM’s InfoSphere Streams is a platform for developing and executing high-performance and highly available stream processing applications. When writing the application, developers use the Streams Processing Language (SPL) [8]. In SPL, developers compose an analytics by specifying a data flow graph, where each vertex is an operator, and each edge is a stream connection. An operator does a transformation on the data flowing through its input stream connection and further produces new data, which is submitted to its output stream connections.

At the language level, developers can specify configurations related to the physical deployment of the application. For example, two or more operators can run on a single processing element (PE), which map to an operating system process. When operators reside in the same PE, they exchange tuples through in-memory queues and function calls. When they are in different PEs, the communication happens via TCP/IP. Different PEs of the same application can run in different hosts. Splitting the data flow graph into PEs and hosts depends on the resources and performance requirements of the application.

An administrator deploys an application by submitting it to the Streams Application Manager (SAM). SAM is then responsible for starting and managing the application. PEs are started by communicating with a host controller (HC). The HC is responsible for starting up the process and for periodically reporting the health of all PEs running on the host to SAM.

![Instantiations of failure, rejuvenation and recovery model](image)

Fig. 5. Instantiations of failure, rejuvenation and recovery model

When an operator fails (e.g., due to an unhandled exception), the PE also crashes. When a PE crashes, the HC reports
the failure occurrence to SAM, and SAM automatically restarts it. If an HC or SAM crashes, the application continues to run without interruption. HC and SAM can automatically restart without administrator intervention. If a PE crashes while SAM is offline, the PE is restarted only after SAM comes back online. If a PE crashes while HC is offline, Streams tries to restart HC first. If HC does not come back online for a period of time, SAM migrates the PE to a different host. PE host migration also happens when there is a host failure.

Figure 5 illustrates two instances of the general failure, rejuvenation and recovery model described in Section IV. Figures 5(a) and 5(b) progressively specialize the model presented in Figure 2 by considering (a) generic distributed applications involving system and host monitoring and (b) the InfoSphere Streams Architecture, as described above. In Section VI we evaluate how different model parameters impact metrics of interest, e.g., mean time to recover.

B. File System Architectures

In this section, we describe two alternative file system architectures: HDFS and MapR-FS. The illustrative target application can use HDFS or MapR-FS. These two alternative and competing architectures allow us to appreciate the need of quantitative analysis when making software architecture decisions. While HDFS is an open-source free solution, MapR-FS is a proprietary solution. We envision that each alternative is associated to a different set of parameters in the proposed analytical model (the parameterization of the model being subject for future work). In Section VI we numerically investigate the proposed analytical model under different set of parameters, so as to quantitatively assist practitioners in the choice between different system architectures.

Figure 6 shows the different layers considered. In this work, we focus on layer three, and indicate how layers one and two impact layer three. The analysis of layer four is subject for future work.

1) HDFS Architecture: In the HDFS architecture, an application communicates with the HDFS daemons to do directory reads, file reads and file writes.

As described in [10], the architecture of HDFS has two main components: the NameNode and the DataNode. The NameNode is a central component of the system and is responsible for holding all the file system metadata. The DataNodes hold the files themselves. Files are split into blocks, which can then be stored by different DataNodes.

To tolerate failures, DataNodes maintain replicas of a file block. The number of replicas is defined by a replication factor, which is specified on a per-file basis. The NameNode is responsible for keeping track of the liveness of DataNodes and how many replicated blocks are currently available per file. If the number of available replicated blocks go down due to a DataNode failure (e.g., host failure or file block corruption), the NameNode will automatically replicate the unavailable block into another DataNode.

The NameNode has its own fault tolerance features, such as maintaining a transaction log and multiple copies of the metadata in its local disk. If the NameNode host fails, then the system becomes unavailable and needs administrator intervention.

Our target telco application can survive DataNode failures by restarting the operators using the file system. With NameNode failures, the application cannot make progress until there is administrator intervention.

2) MapR-FS Architecture: Similar to HDFS, the MapR File System (MapR-FS) also aims to achieve fault tolerance in a commodity cluster. It achieves so with a different architecture than HDFS. Unlike HDFS, MapR-FS does not have a single service that is responsible for responding file system metadata requests. File data and metadata are spread around the cluster in a similar fashion.

The data stored in a node is split into many parts called containers. Containers are then replicated evenly into all other nodes of the system. If a node crashes, its containers can be reconstructed from the copies that are spread out in the cluster. As the copies are spread in all other cluster nodes, the reconstruction can happen in parallel for different containers. This process is triggered automatically in case of failures.

If a node in the MapR-FS cluster fails, our target telco can detect such error, throw an exception, triggering an automatic restart by the InfoSphere Streams platform. On a new directory and file read attempt, the application will be routed to a an available node containing metadata and the file data.

C. Illustrative Parameterization of Target Telecommunications Application

Next, we make some additional simplifying assumptions to further illustrate how to parameterize the infrastructure services used by our target telecommunications application. We first consider the InfoSphere Streams services (Figure 5(b)), accounting for a) the probability that migration is successful and b) the probability that SAM and HC are up.

| TABLE III. MODEL PARAMETERS FOR INFRASTRUCTURE SERVICES ACCOUNTING FOR FILE SYSTEM ARCHITECTURE |
|-----------------------------------------------|-----------------------------------------------|
| parameter | description |                                             |
| s          | probability that SAM is up                    |
| c          | probability that HC is up                     |
| t          | probability that fix of SAM does not impact neither PE nor HC |
| N          | number of data nodes in the network           |
| d          | probability that data node is up              |
| e          | probability that name node is up              |
| μ          | rate of migration trials                     |

To appreciate how file system information can be integrated in our model, let us consider the rate at which the process transitions from state F2 to state F1, referred to as $\beta_2$. In this section, we assume that this transition occurs exclusively due to successful migrations. Let $\mu$ be the rate of migration trials. Let us further assume that, as far as there is at least one data node up, migration is successful. Then, $\beta_2^{(HDFS)}$ is given as a function of $\mu$ as follows,

$$\beta_2^{(HDFS)} = \mu(1 - dN^{-1})e$$  \hspace{1cm} (1)

where $N$ is the number of data nodes in the network, $d$ is the probability that a given data node is up and $e$ is the
probability that the name node is up. Table III summarizes the terminology used for the parameterization of the InfoSphere Streams Architecture.

As in MapR-FS there are no name nodes, \( \beta_2^{(\text{MapR-FS})} \) is given by

\[
\beta_2^{(\text{MapR-FS})} = \mu (1 - dN^{-1})
\]  

(2)

where \( N \) is the number of MapR-FS nodes in the network and \( d \) is the probability that a given MapR-FS node is up.

In order to further illustrate how to simplify the parameterization of the model in Figure 5(b), let us now consider the additional assumption that the events of SAM being up and HC being up are independent. Let \( s \) be the probability that SAM is up and \( c \) be the probability that HC is up. Then, due to independence,

\[
p_1 = sc, \quad p_2 = s(1-c), \quad p_3 = 1-s
\]  

(3)

Recall that \( \tilde{p}_2 \) is the probability that HC is down immediately after a fix of SAM. Let \( t \) be the probability that a fix of SAM does not impact neither PE nor HC. Then,

\[
\tilde{p}_2 = (1-c)t
\]  

(4)

In the numerical evaluation of the proposed model presented in the following section, in order to reduce the parameter space and simplify presentation we will consider the special case where \( p_2 = \tilde{p}_2 \), which corresponds to \( s = t \) under the parameterization above.

VI. ANALYTICAL MODEL EVALUATION

In this section we report numerical results obtained with the proposed analytical model. Our goals are to show the applicability of the model to a) compare systems based on their reliability properties, b) understand the impact of different parameters on how the system behaves from failure up to recovery and c) assist practitioners in setting sampling rates to detect failures and proactively rejuvenate the system. To this aim, we explore the parameters of the failure, rejuvenation and recovery model (Figure 3), showing the impact of these parameters on the metrics of interest.

A. Impact of Probability of Automated Failover on Mean Time to Recovery

Next, we consider the reference setup with parameters given by their default values as described in Table II. Then, we investigate how the mean time to repair the system varies as a function \( p_1 \) and \( p_2 \).

Figure 7 allows us to compare different solutions as a function of \( p_1 \) and \( p_2 \). Note that when \( p_1 \) and \( p_2 \) are close to 0, the mean repair time is roughly 1 hour. This is because the system remains on average 30 minutes at state P, and then takes on average roughly 30 additional minutes to transition to state 0. By increasing \( p_1 \) and \( p_2 \), the mean repair time can be reduced by up to 20 minutes, to roughly 40 minutes. The decision of whether to invest in strategies that increase \( p_1 \) or \( p_2 \) depends on the costs and on the envisioned gains. For instance, when \( p_1 \) and \( p_2 \) are close to 0, increasing \( p_1 \) by 0.3 yields a reduction in the mean time to repair of roughly 50 minutes. In
contrast, increasing $p_2$ by the same amounts barely perturbs the mean repair time.

Figure 8 shows how $p_1$ and $p_2$ affect the mean time to repair when $p_{sr} = 0.2$. In this case, the contour plots indicate that the mean time to repair is equal to 1.6 hours if $(p_1, p_2) = (0, 0)$ but reduces to 1 hour when $p_1 = 1$. When $p_2 = 1$, the mean time to repair equals 1.4 hours.

**B. Survivability Model Analysis**

Next, we numerically evaluate the behavior of the survivability model comprising states F1, F2, F3 and 0 in Figure 3. Figures 9 and 10 are obtained using equations (8) and (9) in the appendix, respectively. Figure 9 shows the mean time to repair from hard failures as a function of $p_1$ and $p_2$. The mean time to repair is more sensitive to $p_1$ than $p_2$ when $p_1$ and $p_2$ are small. As $p_1$ and $p_2$ increase, the mean time to repair from hard failures, after such failures occur, reduces from 0.8 hours to roughly 0.1 hours.

Figure 10 shows that the standard deviation of the time to repair from hard failures also decreases as $p_1$ and $p_2$ increase, which indicates that recovery becomes more predictable in the event that hard failures occur.

**C. Sampling Rate and Rejuvenation**

Figure 11 shows the mean time to repair the system as a function of the sampling rate $\rho$ and rejuvenation success probability, $p_{sr}$. Note that when $p_{sr} = 0$, the mean time to repair is insensitive to $\rho$. Therefore, increasing $\rho$ might increase costs and bring no benefits. As $p_{sr}$ increases, rejuvenation is beneficial. For $p_{sr} = 1$, slightly increasing $\rho$ yields a significant reduction in the mean time to repair.

**D. Model Applicability**

The proposed model is applicable to the analysis of applications running on InfoSphere Streams, and to compare the impact of failures in the different file system architectures on our target telecommunications application, as discussed next.

1) **InfoSphere Streams Architecture Analysis:** The discussion presented in the previous sections is applicable to the analysis of the InfoSphere Streams Architecture. Noting that Figure 5(b) is a special instance of Figure 2, the results reported in Sections VI-A, VI-B and VI-C immediately apply to the InfoSphere Streams Architecture, after translating terms to the specifics of the domain as discussed in Section V-A.

2) **File System Architecture Comparison:** The proposed model can be used to compare two file-systems, HDFS and MapR-FS, used to support a target application. As discussed in Section V-B, while HDFS is an off-the-shelf free solution, MapR-FS is a proprietary solution, which might be associated to better survivability but higher costs. Assuming that each solution is associated to a different $(p_1, p_2)$ pair, and letting the other parameters be fixed to their default values (Table II), the discussion presented in Section VI-A, VI-B and VI-C immediately applies to the comparison of different file system architectures. The parameters presented in Table II can be obtained, for instance, through controlled experiments in a
testbed. While the detailed parameterization of the analytical model based on experimentation is subject for future work, in this section we focused on an exploratory study of the different model parameters and their sensitivity analysis.

VII. Related Work

There is a vast literature on analytical models to quantify software architecture reliability [15], [16], [17]. These works consider a fine grained division of the system into modules to account for failures, which are usually modeled using the probability of failure on demand paradigm [18], [19]. In this paper, we take a bird’s-eye survivability modeling view using a high-level failure model categorization of different types of failures by taking advantage of experimental failure categorizations [4], [5], [6] and their impact on the system as a whole.

A. Software Architecture Reliability and Optimization

Lyu [15] surveyed software reliability engineering methodologies. The approaches presented are related to 1) fault prevention, removal, tolerance and forecasting, 2) software reliability models, 3) operational profile estimation, and 4) measurement and analysis of software reliability. Architecture-based software reliability models have also been reviewed and compared by Goseva et al. [16]. The presented models are designed to help the architect understand the relationship between overall system reliability, individual component reliabilities, under the failure on demand paradigm, and the expected interactions between components. Goseva et al. showed empirical results validating the estimations provided by architecture-based reliability models. Cortellessa et al. [20] propose an approach for software architecture modeling after the deployment phase, to address software architecture evolution and to respond to requirements churn under reliability constraints.

Additionally, software architecture optimization approaches (survey in [21]) aim to automatically search for the best design alternatives in a given subset of the design space, combining the aforementioned quantitative assessment techniques and optimization techniques. Martens et al. [22], for instance, apply multi-criteria optimization to a software architecture model to improve performance, reliability, and costs and identify required trade-offs between them. The approach builds on top of previous work by Brosch et al. [23] to evaluate reliability. However, to the best of our knowledge, no approach has targeted the survivability of software architectures yet.

B. Architecture Styles

Klein et al. [24] introduce the attribute based architecture style (ABAS). ABAS enabled principled reasoning about architecture styles by adding three components to the style: 1) component topology, 2) quality attributes, 3) systematic analysis using models (e.g. queuing models). In this paper we present a systematic methodology for high-level software architecture optimization using architecture styles and model, inline with the analysis paradigm introduced in [24]. We extend the approach introduced in [24] by using a survivability and rejuvenation modeling approach.

In this paper, we illustrate the applicability of the proposed approach using a streaming application (Section V). In particular, the streaming application presented in Figure 4 is built using the pipe and filter software architecture defined by Garland et al. [17]. In the pipe and filter architecture style, software components have defined inputs and outputs. Each component processes and transforms its input datastream sequentially to generate the associated output datastream. In this software architecture style, software components are called filters and the connectors between the components are called pipes. Specialized versions of this architecture style are implemented using Unix filters connected by Unix pipes. Software architectures built in the pipe and filter architecture style lend themselves to throughput analysis and optimization, but may suffer from long end-to-end delays depending on the number of filter components used to build the software pipeline. Modern streaming applications can be analyzed using the pipes and filters architecture style. Examples of high-throughput streaming applications are large data set processing to support big data analytics, and the monitoring and alarming of large number of components in critical infrastructures, such as gas, water, power and telecommunications.

C. Survivability Modeling

Heegaard and Trivedi [25] study the survivability of the Internet and computer networks. In [26], a Continuous Time Markov Chain (CTMC) is used to model the actions taken in reaction to a failure in a telecommunication network, evaluating an extension of the System Average Interruption Duration Index (SAIDI) that accounts for variations of energy demand and supply during a multi-step recovery process. The approach is extended in [27] to quantify the Energy Not Supplied (ENS) in the presence of multiple failures under specific independence assumptions. These methodologies introduced in [26], [27] are related to the research presented in this paper as they consider the survivability of critical infrastructures. Nonetheless, the work presented here significantly differs from previous work as we (1) apply a survivability modeling approach to software architecture analysis, (2) apply survivability modeling to data streaming architectures, and (3) introduce survivability metrics for the assessment of architecture deployment alternatives for a complex data streaming application.

D. Software Rejuvenation

In this paper, we consider rejuvenation as one of the proactive strategies to prevent failures [28], [29], [30]. In [31] a software monitoring approach to decide when to apply software rejuvenation to recover from soft failures was introduced. The approach used leaky-bucket concepts and was designed to distinguish between hard failures and soft failures. The leaky-bucket based approach was later extended in [32] to distinguish between soft failures and security intrusions.

VIII. Conclusion

In this paper we have presented a novel approach for the assessment of software architecture deployment alternatives. Our approach takes advantage of historical failure data to create a failure model. The failure model is used to categorize the failures into soft failures and hard failures. The phased recovery from hard failures is modeled using a survivability
model, while soft failures are pro-actively avoided using a software aging detection and rejuvenation approach.

The proposed failure, rejuvenation and recovery analytical model can be used to compute different metrics of interest to software architecture designers of reliable systems. In particular, in this paper we focused on the mean time from a failure prone state to the failure recovery state. We called this metric the mean time to repair. We have applied our methodology to a large industrial telecommunication system and we were able to indicate how different parameters of deployment alternatives have an impact on the mean time to repair metric.

In future work, we will study the detailed parametrization of the analytical model based on experimentation. Additionally, we plan to integrate this approach into the architecture modeling approach Palladio [33]. Thus, we will enable survivability assessment directly on a design representation of the software architecture by providing a transformation form the Palladio Component model to the survivability model presented in this paper. Furthermore, we will enable multi-criteria optimization that also considers performance and costs criteria and supports well-informed decisions accounting for the trade-off between cost and reliability.

APPENDIX

Given that a hard failure occurred, let $T$ be the time until full repair. The distribution of $T$ is characterized by the following Laplace transform.

$$T^*(s) = E[e^{-sT}] = p_1T_1^*(s) + \left( p_2T_2^*(s) + p_3T_3^*(s) \right) \delta + \frac{\beta_1 p_2}{\beta_2 + \delta} \times \left( \frac{\delta}{\delta + \beta_2} T_1^*(s) + \frac{\delta}{\delta + \beta_1} + \frac{\beta_1 (1 - p_2)}{\beta_1 + \delta} T_1^*(s) \right)$$

where

$$T_1^*(s) = \frac{\alpha + \delta}{\alpha + \delta + s}$$  \hspace{1cm} (5)

$$T_2^*(s) = \frac{\beta_2 + \delta}{\beta_2 + \delta + s}$$  \hspace{1cm} (6)

$$T_3^*(s) = \frac{\beta_1 + \delta}{\beta_1 + \delta + s}$$  \hspace{1cm} (7)

The mean and variance of the time to repair from hard failures are given by

$$E[T] = -\frac{dT^*(0)}{dt}$$  \hspace{1cm} (8)

$$V(T) = \frac{d^2T^*(0)}{dt^2} - (E[T])^2$$  \hspace{1cm} (9)

REFERENCES


