Department of Computer Science
George Mason University Technical Reports

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/ 703-993-1530

Design and Experimentation of an Automated Performance Evaluation Testbed for Self-Healing and Self-Adaptive Distributed Software Systems

Jason Porter
jporte10@gmu.edu

Daniel A. Menascé
menasce@gmu.edu

Hassan Gomaa
hgomaa@gmu.edu

Emad Albassam
ealbassa@gmu.edu

Technical Report GMU-CS-TR-2017-2

Abstract

Evaluating the performance of distributed software systems is very challenging especially in the presence of failures and adaptation. Of particular interest to this paper is self-healing and self-adaptation middleware that detects failures of distributed software systems, analyzes their root causes, devises plans to recover from these failures, and executes these plans. Recovery plans may trigger software architecture adaptations, which may be also initiated by the need to maintain performance and availability goals. This paper focuses on the evaluation and testing of recovery and adaptation frameworks (RAF) for distributed component-based software systems. We present TESS, a testbed for automatically generating distributed software architectures and their corresponding runtime applications, deploying them to the nodes of a cluster, running many different types of experiments involving failures and adaptation, and collecting in a database the values of a variety of failure recovery and adaptation metrics. Queries can then be run against the database to provide a thorough and scientific analysis of the efficiency and/or effectiveness of a RAF. Additionally, this paper presents a case study of the use of TESS for the evaluation of a specific RAF, called DARE, developed by our group.

1 Introduction

Evaluating the performance of distributed software systems is very challenging especially in the presence of failures and adaptation. This challenge is exarcerbated by the lack of global state knowledge, by the possibility of multiple concurrent failures of networks and nodes, and by delays in message transmission. Of particular interest to this paper is the performance evalutation of self-healing and self-adaptation frameworks that detect failures of distributed software systems, analyze their root causes, devise plans to recover from these failures, and execute these plans, according to the well-known MAPE-K (Monitor, Analyze, Plan, and Execute based on Knowledge) model for autonomic computing [1]. In the context of software systems, self-healing is the capability of a software system to automatically detect failures and then recover to a consistent state so that it can resume normal execution. Self-adaptation is the capability of the software system to automatically adapt its architecture by adding, removing, or replacing components seamlessly at run-time in response to changes in operational environment or user requirements (see e.g. [2]). This paper deals with the complex problem of performance testing and measurement of distributed middleware frameworks that support failure recovery and adaptation of distributed software systems.

The work reported here was developed in the context of the Resilient Autonomic Software Systems (RASS) project [1] aimed at designing, developing, and evaluating a framework to support highly decentralized component-based software systems. As part of the work on the RASS project, our team developed DARE (Distributed Adaptation and REcovery middleware), an architecture-based, decentralized middleware that provides both self-configuration and self-healing properties to large and highly dynamic component-based software architectures [3]. We previously described the operation of DARE using an emergency response system application as an example. In the process of our validation, we felt the need for a testbed that would *automatically* generate distributed architectures and applications, deploy them in the nodes of a cluster, run many different types of experiments, and collect in a database the values of a variety of metrics. A distributed application

[1]www.cs.gmu.edu/~menasce/rass/

consists of a distributed software architecture, consisting of components and connectors, and its implementation. Subsequently, queries can be run against the database so that a thorough and scientific analysis of the efficiency and/or effectiveness of a recovery and adaptation framework, such as DARE, can be carried out.

We designed and implemented such a testbed, called TESS, a Testbed for Evaluation of Self-Healing and Self-Adaptive Distributed Software Systems. TESS was designed and developed so that it can be used by other recovery and adaptation frameworks (RAF) besides DARE. TESS interacts with a RAF through two event logs generated by a RAF and processed by TESS. One of the logs captures the value of core metrics on recovery and adaptation events that are common to any RAF and the second log captures the values of metrics that are specific to the operation of a given RAF. We decided to use logs as opposed to an API as a communication mechanism between a RAF and TESS to decouple as much as possible these two layers.

The specific and unique contributions of this paper are: (1) a detailed design and implementation of TESS, (2) a definition of metrics to evaluate recovery and adaptation frameworks for distributed software systems, and (3) a description and discussion of the results of using TESS for the evaluation of DARE. It should be noted that while we use DARE as an example of a recovery and adaptation framework for our experimentation, TESS can be used by other RAFs as long as they generate the logs in the format expected by TESS.

This paper is organized as follows. Section 2 discusses the main functionalities assumed for recovery and adaptation frameworks and how a RAF interacts with TESS. Section 3 provides an overview of the three phases of TESS: architecture generation; application generation; and application execution and data collection. Sections 4 through 6 describe in detail each of these three phases. Section 7 describes our implementation of TESS in a computer cluster. Section 8 describes the DARE framework and Section 9 describes the experimental procedure used to evaluate DARE using TESS. The results of these experiments are described in section 10. Section 11 discusses related work and Section 12 provides concluding remarks.

2 Recovery and Adaptation Framework (RAF)

TESS is designed to work with recovery and adaptation frameworks (RAF) that provide the services described in this section and interface with TESS through two metric logs (see Fig. 1). The first log, called Core Events Log, stores data on (1) component and node failure events and (2) recovery and adaptations events. TESS reads these event data from this log in order to analyze and generate reports as described in Section 3.

The second log, called RAF-specific Events Log, records information about events specific to the RAF. Some examples of RAF-specific metrics may include the number of messages sent and received by the RAF to achieve its functionality as well as the time taken to perform specific tasks related to failure recovery and adaptation. To enable TESS to have access to the RAF-specific log, a RAF uses a file to be read by TESS to register the set of RAF-specific events and the format of this log. TESS processes and enters the information contained in the two logs into its Metrics DB, which is later used by TESS to provide detailed analysis of the experiments.

Entries in both logs have the same common prefix: timestamp, event type, event parameters. The core event types can be one of {component failure (CF), node failure (NF), component recovery (CR), node recovery (NR), adaptation start (AS), adaptation completion (AC)} and they have parameters associated with them that depend on the event type as illustrated in Table 1. For example, a component failure event has as parameters the id of the component that failed and the id of the node in which the component was running. Note that it is possible for a component to fail without the node on which it is running to fail. A node failure event generates one component failure event for each component running on the failed node in addition to the node failure event. All CF events generated by a NF event have the same timestamp. A component recovery event is generated by a RAF when a component is recovered and instantiated in the same node, if the node did not fail, or in another node, in case the node failed. The node id parameter for the component recovery event indicates the node where the failed component was re-instantiated after recovery. The adaptation start event requires the RAF to generate a unique number to be used as an adaptation id as well as the *adaptation goal*, which consists of a set of one or more components and their interconnections that need to be replaced by a set of one or more interconnected components. Finally, the adaptation end event indicates

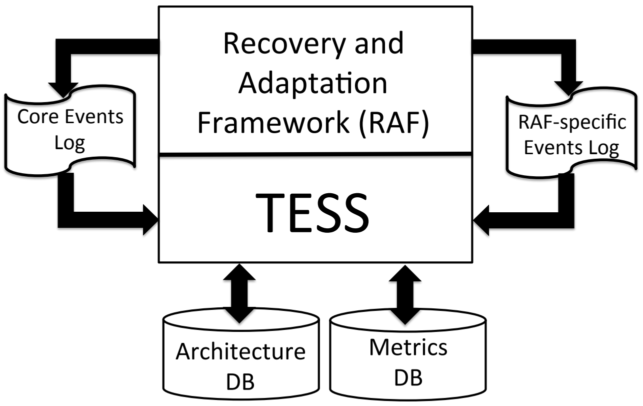

Figure 1: Architecture of a RAF and its Interaction with TESS

when a previously started adaptation ended.

All the events recorded by a RAF in the two logs are timestamped so that they can be properly merged by TESS and stored into its Metrics DB. As indicated in Fig. 1, TESS also keeps an Architecture DB that stores all the architectures to be used during an experiment.

Table 2 provides an example of a few entries for the Core Events Log. The example shows that component C1 running at node N2 failed at time 101 and it was recovered at the same node at time 120. Then, node N4 failed at time 130 and components C2 and C3 that were running at that node also failed. Component C2 was recovered at time 135 at node N5 and component C3 was recovered at node N6 at time 137. The example also shows that node N4 recovered at time 152.

Table 1: Example of parameters for core recovery and adaptation events.

Event type	Parameters
Component Failure (CF)	ComponentId, NodeId
Node Failure (NF)	Node Id
Component Recovery (CR)	ComponentId, NodeId
Node Recovery (NR)	Node Id
Adaptation Start (AS)	AdaptationId, AdaptationGoal
Adaptation End (AE)	AdaptationId

Table 2: Exampe of a Core Events Log.

timestamp	Event type	Event Parameters	
101	CF	C1	N2
120	CR	C1	N2
...	...	....	
130	NF	N4	
130	CF	C2	N4
130	CF	C3	N4
...	...	....	
135	CR	C2	N5
137	CR	C3	N6
...	...	....	
152	NR	N4	

A RAF is assumed to exhibit the following functionalities:

- *Recovery from component failures*: creates a new instance of a failed component and logs event data on component failure detection and recovery events in the Core Metrics Log.

- *Recovery from node failures*: creates a new instance of each component that was executing on the failed node on a new node and logs event data on node failure detection and corresponding recovery events in the Core Events Log.

- *Adaptation*: adapts the software architecture by replacing one or more interconnected components with one or more interconnected components. Adaptation typically includes quiescing components to be disconnected from the application, removing these components, adding new components, and then interconnecting these components with the application.

To start a set of experiments a user must launch the *Start* script that interacts with the user to request (1) an id for the RAF (2) the name of a configuration file used by TESS to drive the process of generating architectures and conducting experiments (see Section 3), (3) the name of the file that contains the Core Metrics Log, and (4) the name of the file that contains the RAF-specific Metrics Log. Upon receiving these parameters, TESS starts the experiments.

3 Architecture of TESS

Figure 2 depicts the architecture of the TESS testbed, which consists of three stages: architecture generation, application generation, and application execution and data collection. During the first stage, TESS automatically generates a user-specified number of software architectures, which are stored in a database (step 1). Users can also add software architectures to the architecture database through a user interface. Each architecture consists of a number of components and connectors among the components. Each generated architecture specifies a set of static attributes for the components. These attributes are used at run-time to determine the behavior of components as explained later. For example, these attributes determine if a component is enabled to send and/or receive messages and of which type (synchronous or asynchronous). These attributes also specify the probability that the component sends a message of a given type at set points during its execution as well as the probability that a component fails at run-time.

The application generation step (step 2) uses a *universal component template*, discussed later in detail, and the static attributes of the components generated in step 1 to generate the application to be tested. The component template provides a probabilistic profile for the run-time behavior of components.

The generated application is then deployed according to a deployment configuration map that indicates how software components are mapped to nodes of a distributed system (see step 3).

The third stage of TESS monitors the execution of the distributed application (step 4), collects the values for a variety of metrics related to failures and their recovery, as well as adaptation, and stores these values in a

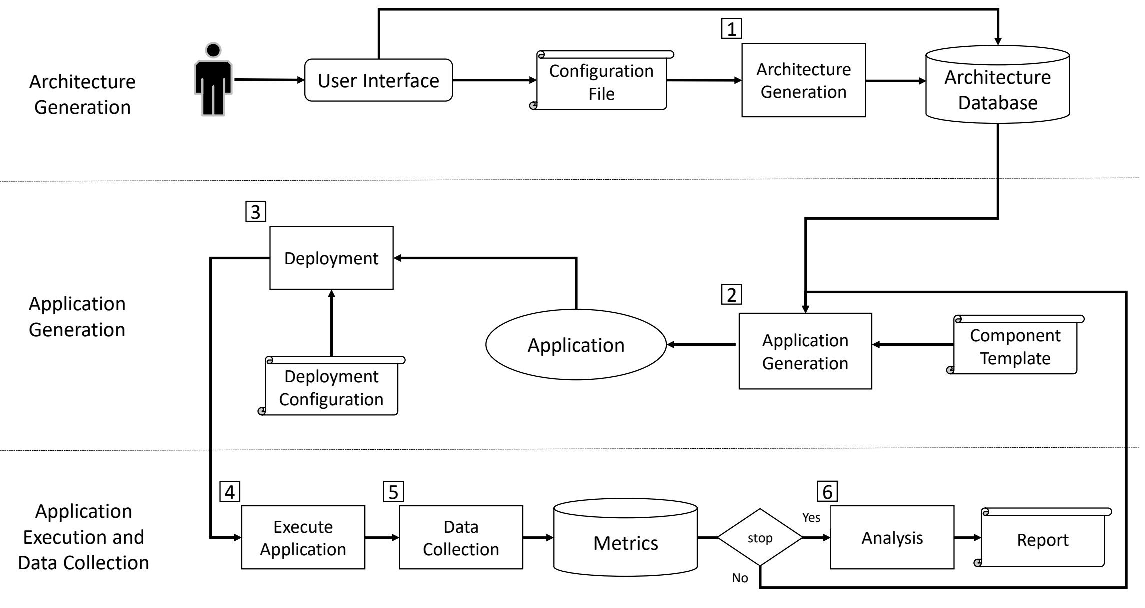

Figure 2: TESS Design

relational database (step 5). This database is analyzed during this stage and produces results based on all applications executed during the experiment but also for specific clusters of architectures based on their complexity (step 6). However, because, as it is usually the case, a user may request that more than one architecture be generated. So, after metrics are collected in step 5 for one of the architectures, TESS checks if other applications need to be generated. In the affirmative case, TESS goes back to step 2. After all experiments are run for the generated architectures, TESS proceeds to step 6 to perform a complete statistical analysis of the results.

TESS was designed to run on a cluster of distributed nodes and uses a MySQL database for storing architectural information and experimental results. Additionally, TESS was designed to be a general purpose environment for the thorough evaluation and testing of self-healing and self-adaptive distributed applications. Details of each stage of TESS are discussed in the next three sections.

4 Architecture Generation

The architecture generation stage of TESS involves the dynamic generation of random architectures represented as labeled directed graphs. Nodes are associated with component types; edges correspond to connectors and indicate the types of communication between components. We consider the following types of communication patterns between components: (1) A sends a synchronous (SY) message to B; this implies that A blocks while waiting for a reply from B. (2) A sends an asynchronous (AS) message to component B; component A can continue processing after sending the message and no reply is expected. We call it a unicast asynchronous message. (3) A sends an asynchronous (AS) message to multiple destinations (MD); component A can continue processing after sending these messages and no reply is expected from any of the recipients. We call this a multicast asynchronous message. Thus, the three possible labels for an edge are: (SY, SD), (AS, SD), and (AS, MD).

During initialization, TESS reads from a configuration file several user-specified parameters such as the number of architectures to be generated, the minimum and maximum number of components in the architectures, as well as other parameters necessary for the experimental process as discussed later.

The generation of architecture random graphs uses an adjacency matrix A whose cell $A[i, j]$ has a null value if components i and j do not communicate and has a communication pattern label L if i and j communicate. The possible values for the label L are: (AS,SD) for asynchronous single destination messages, (AS,MD) for asynchronous multi-destination messages and (SY,SD) for synchronous single destination messages. For further details on representing architectures as labeled directed graphs see [4]. For the graph to be connected we ensure that $N - 1 \leq E \leq N(N - 1)$ where N is the number of nodes and E the number of edges of the graph.

Although the architecture generation process has significant randomness, there are constraints that are enforced to avoid generating architectures that are nonsensical. Examples of such constraints include: (1) each component must communicate with at least one other component, (2) there must be some servers present in the architecture, (3) multicast messages can only be sent as asynchronous and (4) a component can only send a multicast message after it receives a message.

5 Application Generation

The application generation phase of TESS uses a universal component template, shown in Algorithms 1 through 3 and described in what follows, to drive the behavior of the components of the architecture. The static attributes of the components for a given architecture are obtained from the Architecture DB (see previous section). Because many of these attributes are probabilities, components exhibit different behaviors at run time according to the random numbers generated as explained below. We now describe Algorithm 1 from the point of view of a component *S* that receives a message *m* from component *C*. Depending on the component type attribute obtained from the Architecture DB for component *S*, that component may be classified as a sender (lines 3-6), receiver (lines 7-17), sender-receiver (lines 18-30), or receiver-sender (lines 31-43).

If a component is a sender, it potentially sends an asynchronous message and potentially sends a synchronous message (lines 4-5). We say "potentially sends a message" because algorithms 2 and 3, to be explained below, only send messages based on a randomly selected value that is compared with a static probability of that component sending asynchronous or synchronous messages. It should also be understood that sending a synchronous message is a blocking primitive.

If component *S* is a receiver, it receives message *m* (line 8) and may fail with a probability determined by a static attribute of the component as determined in the Architecture DB (see lines 9-12). If the component does not fail, it waits for a random amount of time to simulate the time taken by the component to process the message and act on it (line 26). If message *m* is synchronous, component *S* replies to component *C* (lines 27-29).

If component *S* is a sender-receiver, it potentially sends an asynchronous message and potentially sends a synchronous message (lines 19-20), receives a message *m* and, as explained above, may or may not fail, processes the received message and replies to it if it is a synchronous message (lines 21-29).

Finally, if *S* is a receiver-sender, it receives a message and, as above, may or may not fail (lines 32-36), processes the received message (line 37), potentially sends an asynchronous message and potentially sends a synchronous message (lines 38-39), and replies to message *m* if it is a synchronous message (lines 40-42).

Algorithm 2 shows the algorithm used for potentially sending an asynchronous message. This algorithm is used in lines 4, 19, and 38 of Algorithm 1. A uniformly distributed random number *p* between 0 and 1 is generated (line 1). If this number is less than or equal to the

Algorithm 1: Universal Template

```
   Input  :Component attributes from the
           Architecture Database
1  while application running do
2      switch Component.type do
3          case Sender do
4              SendAsyncMessage
5              SendSyncMessage
6          end
7          case Receiver do
               /* a message is received          */
8              Receive message m from component C
               /* should component fail?         */
9              p ← rand(0..1)
10             if $p \leq$ Component.FailureProbability then
11                 the component fails
12             end
               /* msg. processing time           */
13             wait (AvgMsgProcTime)
14             if m.type = SYN then
15                 Send Reply Message to C
16             end
17         end
18         case Sender-Receiver do
19             SendAsyncMessage
20             SendSyncMessage
21             Receive message m from component C
               /* should component fail?         */
22             p ← rand(0..1)
23             if $p \leq$ Component.FailureProbability then
24                 the component fails
25             end
               /* msg. processing time           */
26             wait (AvgMsgProcTime)
27             if m.type = SYN then
28                 Send Reply Message to C
29             end
30         end
31         case Receiver-Sender do
32             Receive message m from component C
               /* should component fail?         */
33             p ← rand(0..1)
34             if $p \leq$ Component.FailureProbability then
35                 the component fails
36             end
               /* msg. processing time           */
37             wait (AvgMsgProcTime)
38             SendAsyncMessage
39             SendSyncMessage
40             if m.type = SYN then
41                 Send Reply Message to C
42             end
43         end
44     end
45 end
```

probability that the component sends an asynchronous message (line 2), then the component will either send a unicast message (lines 5-6) or a multicast message (lines 8-9). When sending a unicast asynchronous message, a random destination component *D* consistent with the generated architecture is selected and the message is sent. In the case of multicast messages, the message is sent to the multicast group prescribed by the architecure.

Algorithm 2: SendAsyncMessage

```
Input :Component attributes from the
       Architecture Database
1  p ← rand(0..1)
2  if $p \leq$ Component.ProbSendAsync then
3      p ← rand(0..1)
4      if $p \leq$ Component.ProbSendUnicast then
5          Select random destination D consistent with
             the architecture
6          Send asynchronous message to D
7      else
8          Send messages to all components in the
             multicast group specified in the architecture
9      end
10 end
```

Algorithm 3, used in lines 5, 20, and 39 of Algorithm 1, describes how synchronous messages are potentially generated. A uniformly distributed random number p between 0 and 1 is generated (line 1). If this number is less than or equal to the probability that the component sends a synchronous message (line 2), a random destination component *D* consistent with the architecture is selected and a synchronous message is sent to that destination (lines 3-4).

Algorithm 3: SendSyncMessage

```
Input :Component attributes from the
       Architecture Database
1 p ← rand(0..1)
2 if $p \leq$ Component.ProbSendSync then
3     Select random destination component D
        consistent with the architecture
4     Send synchronous message to D
5 end
```

As described above, different components behave differently from each other because they have different static attributes (e.g., Component.type, Component.ProbSendAsyn, and
Component.ProbSendSync) generated during the architecture generation phase and because of the random values of the probabilities generated at run-time.

6 Application Execution and Data Collection

The various components of the application are deployed in the various nodes of a distributed system (a computer cluster in our experiments). Once the application starts to execute, the RAF records core events and RAF-specific events in the logs described above. These logs are then merged at the end of each experiment into a single log at a master node that controls the experiments and stores it into the databases used by TESS. From this merged log, metrics are gathered and stored in the Metrics DB for later analysis.

As mentioned previously, metrics are classified into core metrics and RAF-specific metrics. The core metrics gathered during experimentation include:

- Component Recovery Time: time elapsed from when a component failure was detected to when the failed components were recovered.
- Node Recovery Time: time elapsed from when a node failure was detected to when the failed components located at that node were recovered to a new node.
- Adaptation Time: time elapsed from the initiation to the completion of an adaptation procedure.

These metrics allow for an effective evaluation of the performance of the self-healing and self-adaptation process of any RAF.

Figure 3 depicts the UML class diagram that models the Architecture, Component and Metrics classes, which are mapped to the TESS database(s). The attributes of the Architecture class describe the main elements of a software architecture, which is captured as a directed graph. The Architecture class has 1-to-many associations with the Component and Experiment classes. A given architecture consists of one or more instances of the Component class, the attributes of which describe the characteristics of a given component. Since a given architecture can be mapped to one or more experiments, each instance of the Experiment class describes the results of running an experiment for the specific architecture with which it is associated. The Experiment class has a subclass called RAF-specific Experiment that describes experimental results associated with user defined RAF-specific metrics

6.1 The Architecture DB

The Architecture DB consists of two tables: *Architecture* (contains information for the generated architectures) and *Components* (contains information for the individual components of each architecture). which are described below.

The *Architecture* table has the following columns:

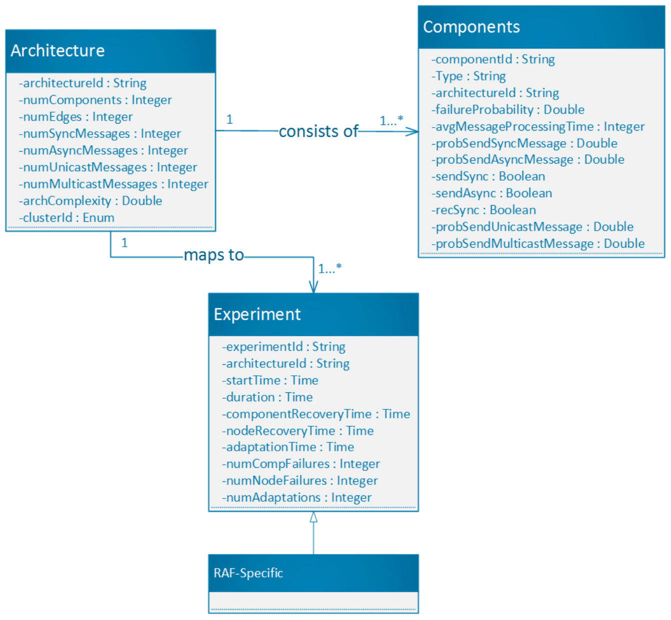

Figure 3: UML Class Diagram for TESS databases

- *ArchitectureId*: unique id for the architecture (primary key).
- *NumComponents*: number of components of the architecture.
- *NumEdges*: number of edges (connections) of the architecture.
- *NumSyncMessages*: total number of synchronous message interfaces for this architecture.
- *NumAsyncMessages*: total number of asynchronous message interfaces for this architecture.
- *NumUnicastMessages*: total number of unicast messages interfaces for this architecture.
- *NumMulticastMessages*: total number of multicast message interfaces for this architecture.
- *ArchComplexity*: architecture complexity computed as:

$$\begin{aligned}\text{Complexity} &= \#\,\text{components} + \#\,\text{edges} + \\ &\quad \#\,\text{edges}/\#\,\text{components} + \\ &\quad \#\,\text{synchronous messages}/\#\,\text{edges} + \\ &\quad \#\,\text{multicast messages}/\#\,\text{edges}.\end{aligned}$$

  This complexity metric is inspired by the cyclomatic complexity for computer programs [5] and was adapted by us to software architectures. Here we consider component-type as opposed to component-instance architectures. Other complexity metrics can be used. We used this metric in our experiments to cluster architectures into simple, moderate, and complex, using K-means clustering with K = 3.

- *ClusterId*: id of the cluster in which this architecture belongs.

The columns of the *Component* table are:

- *ComponentId*: unique id of a component (primary key).
- *Type*: identifies the type of component (for e.g. sender, receiver, sender-receiver, receiver-sender).
- *ArchitectureId*: unique id for the architecture (foreign key).
- *FailureProbability*: probability that a component fails after receiving a message.
- *AvgMessageProcessingTime*: average time elapsed after a component receives a message and the component sends a message in reply.
- *ProbSendSyncMessage*: probability that a message sent by a component is synchronous. This is used to determine the frequency with which a component sends a synchronous vs asynchronous message.
- *ProbSendAsyncMessage*: probability that a message sent by a component is asynchronous. This is used to determine the frequency with which a component sends an asynchronous vs synchronous message.
- *SendSync*: Boolean expression identifying whether a component sends synchronous messages.
- *SendAsync*: Boolean expression identifying whether a component sends asynchronous messages.
- *RecSync*: Boolean expression identifying whether a component receives synchronous messages.
- *ProbSendUnicastMessage*: probability that a message sent by a component is unicast. This is used to determine the frequency with which a component sends a unicast vs multicast message.
- *ProbSendMulticastMessage*: probability that a message sent by a component is multicast. This is used to determine the frequency with which a component sends a multicast vs unicast message.

6.2 The Metrics DB

The Metrics DB consists of a single table, called *Experiment*, which contains the values of the metrics gathered from each run of an experiment. The columns of this table are:

- *ExperimentId*: unique id of the experiment (primary key).
- *ArchitectureId*: unique id for the architecture (foreign key).

- *StartTime*: start time of the experiment.
- *Duration*: duration of the experiment.
- *ComponentRecoveryTime*: the component recovery time metric.
- *NodeRecoveryTime*: the node recovery time metric.
- *AdaptationTime*: the adaptation time metric.
- *NumCompFailures*: number of component failures during the experiment.
- *NumNodeFailures*: number of node failures during the experiment.
- *NumAdaptations*: number of adaptations during the experiment.

Whilst the Architecture table consists of a single entry for each generated architecture and the Components table consists of a single entry for each component of a particular architecture, the Experiments table consists of multiple entries for metrics associated with a given architecture. In other words, for each architecture, multiple experiments would be conducted and numerous values for each type of metric would be collected and stored for later analysis. As an architecture is associated with an experiment by its ArchitectureId, one may then execute queries to derive results for metrics for either a specific architecture or for all architectures of a specific type. As an example, say we generated n architectures of which r were complex architectures and we wanted information on adaptation times for all architectures of this type. If we conducted 10 experiments on each architecture, we would have 10 r results for complex architectures in the Experiment table. We can then execute a query to gather the adaptation times for all complex architectures by joining the Experiment and Architecture tables. Further analysis may be conducted by finding the average, coefficient of variation, range and other statistical measures on the results.

7 TESS Implementation

Figure 4 depicts the implementation of TESS in a computer cluster that consists of a master node, which acts as a gateway, and is connected via a network to the other nodes. The master node hosts the main components of the testbed: a MySQL database for TESS databases, the architecture generation module, the application generation module, and the data collection module. Additionally, the master node stores the merged log used to collate all the events from the event logs from all experiments. All other nodes host the RAF, components of the distributed application generated by the application generation module, as well as local copies of the core and RAF-specific events logs.

8 The DARE Middleware

DARE is based on a decentralized version of the MAPE-K loop model. Every node in the distributed system runs an identical instance of the DARE middleware, which is responsible for:

- Keeping track of the current configuration map of the software system, including the mapping of components to nodes and maintaining the current configuration map of the software system.
- Automatically discovering the current architecture of the software system and rediscovering the architecture after dynamic adaptation. DARE relies on gossiping and message tracing techniques for discovering and disseminating the current software architecture (consisting of components and connectors) in a decentralized fashion [4].
- Monitoring and detecting node failures.
- Analyzing the cause of node failures.
- Planning for dynamically adapting the architecture and recovery of failed nodes.
- Executing a reconfiguration template consisting of reconfiguration commands that handle instantiating components on healthy nodes and establishing the connections between application components.
- Adapting and recovering components after runtime node and/or component failures.
- Communicating with recovery and adaptation connectors (RACs) that handle the recovery of failed transactions and steer application components to a quiescent state in order to carry out dynamic adaptation [6] [7].

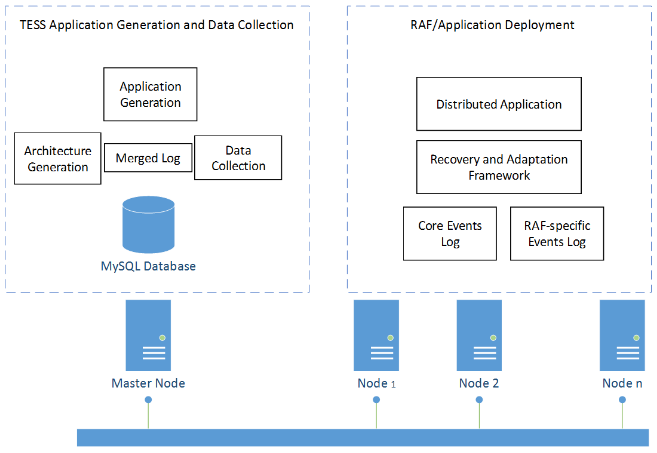

Figure 4: TESS Deployment on a Cluster

9 Experimental Process

We conducted detailed experiments using DARE in order to evaluate TESS. These experiments allowed us the added benefit of not just assessing the testbed, but also gaining valuable insight into the performance of DARE. We implemented TESS in Java, and generated 100 random architectures and clustered them according to complexity into three categories (complex, moderate, and simple). The experiments were then conducted on 10, 15, and 20 nodes of a computer cluster, where for complex architectures each node hosted approximately three components, for moderate architectures each node hosted approximately two components and for simple architectures each node hosted a single component.

Two types of experiments were conducted: self-healing and self-adaptation. The self-healing experiments included both component failures and node failures. For component failures, each component randomly fails during execution according to its failure probability, specified in the *Components* table. With regards to node failures, a random node was selected then taken down accordingly [3]. Component recovery is done by the RAC by instantiating the failed component at another node and replaying the messages (stored in the RACs message queues) that were in transit to/from that component. The self-adaptation experiments involved removing a randomly selected component and replacing it with a load balancing architectural pattern. This entailed adding a load balancing component along with 2 or more replicas of the original component. For further details on DARE's approach to failure recovery and adaptation readers are directed to [3]. As mentioned in Section 2, all relevant events from these experiments were written to the Core Events Log for later analysis.

10 Experimental Results

The experiments reported in this section are related to the core metrics (component recovery time, node recovery time, and component adaptation time). TESS gathered 30 observations of each metric for each architecture complexity type (complex, moderate, and simple) for three node counts (10, 15, and 20). This data was then used by TESS to calculate the mean and 95% confidence intervals (CI) for these metrics. Also, for each metric a two-factor statistical ANOVA procedure was conducted (see e.g., [8] for a description of ANOVA). Here, the factors are architecture complexity with three levels: simple, moderate, and complex and node count with three levels: 10, 15, and 20 nodes. The hypotheses for the ANOVA experiments are:

H_0: (a) the architecture complexity has no impact on the given metric (equivalently, the metric average is the same for all complexity levels), (b) node count has no impact on the given metric (equivalently, the metric average is the same for all node counts), and (c) there is no interaction between the architecture complexity and node count.

H_1: (a) the architecture complexity has an impact on the given metric (equivalently, the metric average is not the same for all complexity levels), (b) node count has an impact on the given metric (equivalently, the metric average is not the same for all node counts), and (c) there is interaction between between the architecture complexity and node count.

Tables 3 and 4 show statistics (average, 95% CI, and range) for the number of components and number of connections between components for each architecture complexity type. The values in these tables help explain the observed behavior when we analyze the metrics described in what follows.

Table 3: Mean, 95% CIs and Range for No. of Components

complexity	mean	1/2 CI	range
complex	26.2	$\pm$ 0.88	21-30
moderate	20.9	$\pm$ 0.69	17-25
simple	13.8	$\pm$ 0.91	10-20

Table 4: Mean, 95% CIs and Range for No. of Connections

complexity	mean	1/2 CI	range
complex	109.9	$\pm$ 4.1	94-151
moderate	83.5	$\pm$ 2.6	69-95
simple	50.3	$\pm$ 3.5	32-65

10.1 Core Metrics

The metrics reported here are: component recovery time, node recovery time, and component adaptation time.

10.1.1 Component Recovery Time

The aim of this experiment was to assess the impact of both architecture complexity and different node counts on component recovery time within DARE. The mean and 95% confidence intervals for component recovery time for each architecture complexity are shown in Tables 5, 6 and 7 for 10, 15 and 20 nodes, respectively. Table 8 shows the results of the two-factor ANOVA for architecture complexity and node count for component recovery time, and Fig. 5 shows component recovery time by architecture complexity and node count. For architecture complexity, $F > F_{\text{crit}}$ results in the rejection of the null hypothesis that architecture complexity does not impact component recovery time. The reason for this is that as architecture complexity increases a component will communicate with a larger number of neighboring components, resulting in a larger number of re-connections required by DARE after recovery. For node count, $F > F_{\text{crit}}$ results in the rejection of the null hypothesis that node count has no impact on component recovery time. This is due to the fact that for smaller node counts more components would be hosted per node for the same architectures than for a larger node count. As a consequence, there would be more overhead per node for smaller node counts which impacts DARE's component recovery mechanism. For factor interaction, $F < F_{\text{crit}}$ results in a failure to reject the null hypothesis that there is no interaction between the two factors.

Table 5: Mean and 95% CIs for Component Recovery Time (10 Nodes)

complexity	mean (sec)	1/2 CI (sec)
complex	31.2	± 2.83
moderate	28.9	± 4.30
simple	23.0	± 2.39

Table 6: Mean and 95% CIs for Component Recovery Time (15 Nodes)

complexity	mean (sec)	1/2 CI (sec)
complex	22.0	± 1.32
moderate	21.2	± 1.13
simple	18.5	± 1.05

Table 7: Mean and 95% CIs for Component Recovery Time (20 Nodes)

complexity	mean (sec)	1/2 CI (sec)
complex	18.0	± 0.91
moderate	16.9	± 0.95
simple	15.2	± 0.98

10.1.2 Node Recovery Time

The aim of this experiment was to assess the impact of both architecture complexity and node count on node recovery time within DARE. The mean and 95% confidence intervals for node recovery time for each architecture complexity are shown in Tables 9, 10 and 11 for 10, 15 and 20 nodes, respectively. Table 12 shows the results of the two-factor ANOVA for architecture complexity and node count for node recovery time, and Fig. 6 shows node recovery time by architecture complexity and node count. For architecture complexity, $F > F_{\text{crit}}$ results in the rejection of the null hypothesis that architecture complexity does not impact node recovery time. This is due to the fact that more complex architectures consist of a higher number of components (see Table 3) being hosted per node resulting in larger node recovery times.

Table 8: Two-Factor ANOVA for Component Recovery Time

Source of Variation	F	P-value	F crit
Architecture Complexity	16.702	1.49E-07	3.030
Node Count	82.306	1.93E-28	3.030
Interaction	2.065	0.0858	2.406

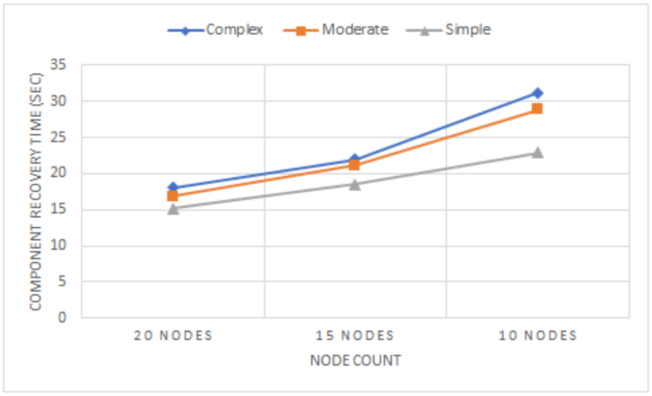

Figure 5: Component Recovery Time by Architecture Complexity and Node Count

For node count, $F > F_{\text{crit}}$ results in the rejection of the null hypothesis that node count has no impact on node recovery time. As mentioned in the previous experiment, smaller node counts host more components per node than larger node counts for the same architectures. This is due to fact that if the number of components within an architecture is fixed, but the number of nodes used to host the architecture is reduced, more components will have to be hosted per node to enable the reduced node count. This in effect results in longer recovery times for smaller node counts. For factor interaction, $F > F_{\text{crit}}$ results in the rejection of the null hypothesis that there is no interaction between the two factors. This is due to the fact that: (a) more (less) complex architectures implies more (less) components hosted per node for the same node count and (b) a larger (smaller) node count implies less (more) components hosted at a node for the same architectural complexity.

Table 9: Mean and 95% CIs for Node Recovery Time (10 Nodes)

complexity	mean (min)	1/2 CI (min)
complex	6.4	± 0.85
moderate	4.9	± 0.59
simple	2.9	± 0.45

10.1.3 Component Adaptation Time

The goal of this experiment was to assess the impact of architecture complexity and node count on component

Table 10: Mean and 95% CIs for Node Recovery Time (15 Nodes)

complexity	mean (min)	1/2 CI (min)
complex	3.5	± 0.22
moderate	2.6	± 0.32
simple	1.6	± 0.20

Table 11: Mean and 95% CIs for Node Recovery Time (20 Nodes)

complexity	mean (min)	1/2 CI (min)
complex	1.7	± 0.18
moderate	1.3	± 0.09
simple	0.8	± 0.09

adaptation time within DARE. The mean and 95% confidence intervals for component adaptation time for each architecture complexity are shown in Tables 13, 14 and 15 for 10, 15 and 20 nodes, respectively. Table 16 shows the results of the two-factor ANOVA for architecture complexity and node count for component adaptation time, and Fig. 7 shows component adaptation time by architecture complexity and node count. For architecture complexity, $F > F_{\text{crit}}$ results in the rejection of the null hypothesis that architecture complexity does not impact component adaptation time. This is a consequence of the fact that a component that has a higher number of interactions with other components will take longer to complete these interactions and then transition to the quiescent state [9], thereby allowing it to be removed and replaced. For node count, $F < F_{\text{crit}}$ results in a failure to reject the null hypothesis that node count does not impact component adaptation time. From Tables 13 and 14 it can be seen that an increase in node count from 10 to 15 nodes results in an increase in component adaptation time, and from 15 to 20 nodes there is a decrease in component adaptation time (see Tables 14 and 15). However, these differences are not statistically significant because the F value for node count (2.438) is less than F_{crit} (3.030) (see Table 16). For factor interaction, $F < F_{\text{crit}}$ results in a failure to reject the null hypothesis that there is no interaction between the two factors.

11 Related Work

Several research areas are related to our work. First, is the performance evaluation of distributed systems.

Table 12: Two-Factor ANOVA for Node Recovery Time

Source of Variation	F	P-value	F crit
Architecture Complexity	74.0	3.48E-26	3.030
Node Count	214.642	7.56E-56	3.030
Interaction	11.256	1.93E-08	2.406

Table 13: Mean and 95% CIs for Component Adaptation Time (10 Nodes)

complexity	mean (min)	1/2 CI (min)
complex	4.5	± 1.31
moderate	3.7	± 0.78
simple	2.6	± 0.61

Table 14: Mean and 95% CIs for Component Adaptation Time (15 Nodes)

complexity	mean (min)	1/2 CI (min)
complex	5.1	± 1.07
moderate	4.1	± 0.80
simple	3.4	± 0.71

Table 15: Mean and 95% CIs for Component Adaptation Time (20 Nodes)

complexity	mean (min)	1/2 CI (min)
complex	3.9	± 2.31
moderate	3.1	± 0.69
simple	2.5	± 0.71

Table 16: Two-Factor ANOVA for Component Adaptation Time

Source of Variation	F	P-value	F crit
Architecture Complexity	6.194	0.0024	3.030
Node Count	2.438	0.0893	3.030
Interaction	0.085	0.987	2.406

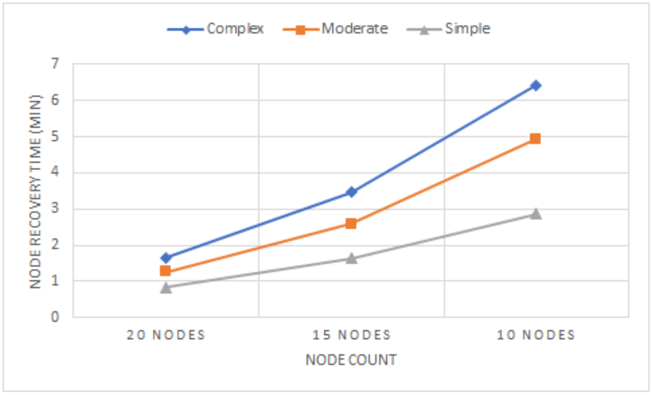

Figure 6: Node Recovery Time by Architecture Complexity and Node Count

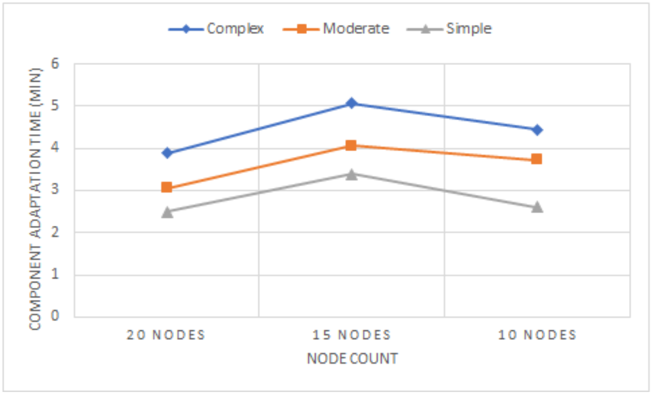

Figure 7: Component Adaptation Time by Architecture Complexity and Node Count

In [10] Mohamed et al. describe the performance evaluation of distributed event-based systems. Michael et al. [11] describe CloudPerf, a framework for the performance evaluation of distributed multi-tenant cloud environments. Wouw et al. [12] discuss the performance evaluation of distributed SQL query engines. In [13] Apte describes the performance evaluation of distributed software systems using queueing theory, and Sachs et al. [14] describe the performance evaluation of distributed message-oriented middleware.

Also related to our work is the performance evaluation of self-adaptive systems and self-healing systems. With regards to the former, [15], [16], [17], and [18] all describe approaches to the performance evaluation of self-adaptive systems. In the case of the latter, [19] and [20] both describe the performance evaluation of self-healing systems.

Another area related to our work is testbeds. In [21], Younan et al. describe a tesbed environment for evaluating Internet of Things (IoT) devices. Sakellari et al. [22] provide a survey of testbeds suitable for various aspects of research in cloud computing. Also of interest are testbeds developed for evaluating distributed systems (see e.g. [23], [24], [25], [26]).

In contrast to the previous, TESS while developed for distributed software systems, focuses on both self-healing and self-adaptation frameworks. To the best our knowledge there does not exist another testbed that provides an automated approach to the performance evaluation of both self-adaptive and self-healing distributed software systems.

12 Concluding Remarks

Several recovery and adaptation frameworks have been proposed for self-healing and self-adaptation of distributed software systems. In most cases, these frameworks are evaluated with one or two distributed system application examples and in many cases little or no quantitative evaluation is conducted [27]. For that reason, we decided to design and implement TESS, described in detail above, to assist in the quantitative evaluation of recovery and adaptation frameworks. TESS was designed and implemented as a tool that can be used to evaluate a variety of self-adaptive and self-healing frameworks such as DARE and others.

TESS follows the well-known principles of experimental design [8] by generating random architectures that are clustered into complex, medium, and simple architectures, and running experiments where node and component failures and component adaptations occur randomly. The metrics gathered by TESS are stored in a database and stored procedures are used to generate a variety of useful metrics such as averages, confidence intervals, and statistical procedures such as ANOVA.

Our use of TESS to evaluate DARE illustrates how TESS can be used for detailed experimental evaluation of recovery and adaptation frameworks. TESS could be extended to automatically track and report on detailed elements of the recovery and/or adaptation times as long as that information is available in the logs generated by RAFs. This would allow users to obtain a better understanding of the major sources of delay in each case. Additionally, it is possible to extend TESS to consider additional core metrics such as the ones proposed in [27] for adaptation.

Acknowledgements

This work was partially supported by the AFOSR grant FA9550-16-1-0030 and the Office of Research Computing at George Mason University.

References

[1] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[2] D. Menascé, H. Gomaa, J. Sousa *et al.*, "Sassy: A framework for self-architecting service-oriented systems," *Ieee Software*, vol. 28, no. 6, pp. 78–85, 2011.

[3] E. Albassam, J. Porter, H. Gomaa, and D. Menascé, "Dare: A distributed adaptation and failure recovery framework for software systems," in *the 14th IEEE International Conference on Autonomic Computing (ICAC)*, 2017.

[4] J. Porter, H. Gomaa, and D. Menascé, "Desarm: A decentralized mechanism for discovering software architecture models at runtime in distributed systems," in *11th Intl. Workshop on Models@run.time*, 2016.

[5] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, Jul. 1976. [Online]. Available: http://dx.doi.org/10.1109/TSE.1976.233837

[6] E. Albassam, H. Gomaa, and D. Menascé, "Model-based recovery connectors for self-adaptation and self-healing," in *Proc. 11th Intl. Joint Conf. Software Technologies*, 2016.

[7] ——, "Model-based recovery and adaptation connectors: Design and experimentation," *Software Technologies*, 2017.

[8] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, NY, 1991.

[9] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Tr. Software Engineering*, vol. 16, no. 11, pp. 1293–1306, 1990.

[10] S. Mohamed, M. Forshaw, N. Thomas, and A. Dinn, "Performance and dependability evaluation of distributed event-based systems: A dynamic code-injection approach," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 2017, pp. 349–352.

[11] N. Michael, N. Ramannavar, Y. Shen, S. Patil, and J.-L. Sung, "Cloudperf: A performance test framework for distributed and dynamic multi-tenant environments," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 2017, pp. 189–200.

[12] S. v. Wouw, J. Viña, A. Iosup, and D. Epema, "An empirical performance evaluation of distributed sql query engines," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM, 2015, pp. 123–131.

[13] V. Apte, "Performance analysis of distributed software systems: Approaches based on queueing theory," in *Software Architecture, 2007. WICSA'07. The Working IEEE/IFIP Conference on*. IEEE, 2007, pp. 39–39.

[14] K. Sachs, S. Kounev, J. Bacon, and A. Buchmann, "Performance evaluation of message-oriented middleware using the specjms2007 benchmark," *Performance Evaluation*, vol. 66, no. 8, pp. 410–434, 2009.

[15] M. Becker, M. Luckey, and S. Becker, "Model-driven performance engineering of self-adaptive systems: a survey," in *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures*. ACM, 2012, pp. 117–122.

[16] ——, "Performance analysis of self-adaptive systems for requirements validation at design-time," in *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*. ACM, 2013, pp. 43–52.

[17] J. Ehlers, A. van Hoorn, J. Waller, and W. Hasselbring, "Self-adaptive software system monitoring for performance anomaly localization," in *Proceedings of the 8th ACM international conference on Autonomic computing*. ACM, 2011, pp. 197–200.

[18] D. Perez-Palacin and J. Merseguer, "Performance evaluation of self-reconfigurable service-oriented software with stochastic petri nets," *Electronic Notes in Theoretical Computer Science*, vol. 261, pp. 181–201, 2010.

[19] E. G. Pereira, R. Pereira, and A. Taleb-Bendiab, "Performance evaluation for self-healing distributed services and fault detection mechanisms," *Journal of Computer and System Sciences*, vol. 72, no. 7, pp. 1172–1182, 2006.

[20] E. Grishikashvili, R. Pereira, and A. Taleb-Bendiab, "Performance evaluation for self-healing distributed services," in *Parallel and Distributed Systems, 2005. Proceedings. 11th International Conference on*, vol. 2. IEEE, 2005, pp. 135–139.

[21] M. Younan, S. Khattab, and R. Bahgat, "An integrated testbed environment for the web of things," *ICNS 2015*, p. 83, 2015.

[22] G. Sakellari and G. Loukas, "A survey of mathematical models, simulation approaches and testbeds used for research in cloud computing," *Simulation Modelling Practice and Theory*, vol. 39, pp. 92–103, 2013.

[23] T. Buchert, C. Ruiz, L. Nussbaum, and O. Richard, "A survey of general-purpose experiment management tools for distributed systems," *Future Generation Computer Systems*, vol. 45, pp. 1–12, 2015.

[24] C. Leng, M. Lehn, R. Rehner, and A. Buchmann, "Designing a testbed for large-scale distributed systems," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 400–401, 2011.

[25] L. Leonini, É. Rivière, and P. Felber, "Splay: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze)." in *NSDI*, vol. 9, 2009, pp. 185–198.

[26] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 255–270, 2002.

[27] L. Birdsey, C. Szabo, and K. Falkner, "Identifying self-organization and adaptability in complex adaptive systems," in *11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, 2017.