TESS: Automated Performance Evaluation of Self-Healing and Self-Adaptive Distributed Software Systems

Jason Porter
Computer Science Department
jporte10@gmu.edu

Daniel A. Menascé
Computer Science Department
menasce@gmu.edu

Hassan Gomaa
Computer Science Department
hgomaa@gmu.edu

Emad Albassam
Computer Science Department
ealbassa@gmu.edu

ABSTRACT

This paper deals with the problem of evaluating and testing recovery and adaptation frameworks (RAF) for distributed 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. Using the collected data, TESS automatically performs a thorough and scientific analysis of the efficiency and/or effectiveness of a RAF. This paper presents a case study on the use of TESS to evaluate DARE, a RAF developed by our group.

CCS CONCEPTS

• **General and reference** → **Empirical studies**; **Measurement**; **Metrics**; **Performance**; • **Computer systems organization** → **Distributed architectures**; • **Software and its engineering** → **Software architectures**; *Software performance*; *Software reliability*;

KEYWORDS

automated experimentation testbed; distributed component-based software systems; experimental design; self-healing software; self-adaptive software; software architecture.

ACM Reference Format:
Jason Porter, Daniel A. Menascé, Hassan Gomaa, and Emad Albassam. 2018. TESS: Automated Performance Evaluation of Self-Healing and Self-Adaptive Distributed Software Systems. In *Proceedings of ACM/SPEC International Conference on Performance Engineering (ICPE'18).* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3184407.3184408

ICPE'18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5095-2/18/04…$15.00
https://doi.org/10.1145/3184407.3184408

1 INTRODUCTION

Evaluating the performance of distributed software systems in the presence of failures and adaptation is very challenging. This challenge is exarcerbated by the lack of global state knowledge, by the possibility of multiple concurrent failures of networks and nodes, and by message delays. This paper focusses on the performance evaluation 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 MAPE-K (Monitor, Analyze, Plan, and Execute based on Knowledge) model for autonomic computing [7]. Self-healing is the capability of a software system to automatically detect failures, recover to a consistent state, and 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. [10]). This paper deals with the complex problem of performance testing and measurement of distributed recovery and adaptation frameworks for distributed software systems.

The work reported here was developed in the context of the Resilient Autonomic Software Systems (RASS) project (www.cs.gmu.edu/~menasce/rass/) aimed at designing, developing, and evaluating a framework to support highly decentralized component-based software systems. As part of the RASS project, we developed DARE (Distributed Adaptation and RECovery middleware), an architecture-based, decentralized middleware that provides self-configuration and self-healing properties to large and highly dynamic component-based software architectures [3]. We previously described DARE using an emergency response system application as an example. In that process 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 the values of a variety of metrics in a database. The data collected could then be used to automatically perform a thorough and scientific analysis of the efficiency and/or effectiveness of a recovery and adaptation framework (RAF), such as DARE.

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. The focus of this paper is TESS, a testbed to automatically and thoroughly evaluate autonomic systems such as DARE and

similar RAFs. Thus, TESS is complimentary to DARE and DARE is used here as a case study for demonstrating TESS' capabilities.

The specific and unique contributions of this paper are: (1) design and implementation of TESS, (2) metrics to evaluate recovery and adaptation frameworks for distributed software systems, and (3) discussion of the results of using TESS for the evaluation of DARE.

This paper is organized as follows. Section 2 discusses the functionalities of RAFs and their interaction with TESS. Section 3 describes the design of TESS. Section 4 describes the DARE framework and Section 5 describes the experimental procedure used to evaluate DARE using TESS and the results of these experiments. Section 6 discusses related work and Section 7 provides concluding remarks.

2 RECOVERY AND ADAPTATION FRAMEWORK

TESS is designed to work with RAFs 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

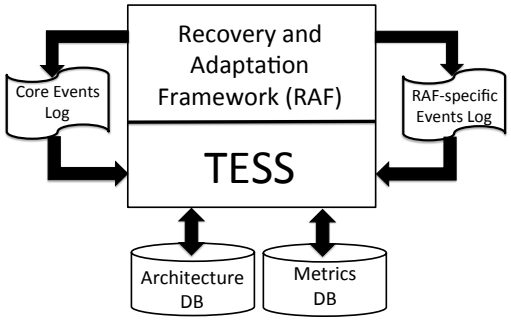

Figure 1: RAF Architecture and Interaction with TESS

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

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

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 on the same node, if the node did not fail, or on 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 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 illustrates a few entries of the Core Events Log: (a) Component C1 failed at node N2 at time t=101 and recovered at N2 at t=120. (b) Node N4 failed at t=130 and components C2 and C3 running at that node also failed. (c) Component C2 recovered at t=135 at node N5 and C3 recovered at node N6 at t=137. (d) Node N4 recovered at t=152.

A RAF is assumed to exhibit the following functionalities: (a) *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. (b) *Recovery from node*

Table 2: Example 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

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. (c) *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 interconnecting them with existing components [8].

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. TESS starts the experiments upon receiving these parameters.

3 DESIGN OF TESS

Figure 2 depicts the design of TESS, 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 user-defined software architectures to the architecture database through a user interface. Each architecture consists of a number of components and connectors that interconnect 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 in Section 3.3, 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 runtime 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 of a variety of metrics related to failures and their recovery, as well as adaptation, and stores these values in a 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). 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.

3.1 Architecture Generation

The architecture generation stage of TESS involves the dynamic generation of random architectures represented as labeled directed graphs [13]. Nodes are associated with component types; edges correspond to connectors and indicate the types of communication patterns between components. We consider three types of communication patterns: (1) Component A sends a synchronous (SY) message to another component and blocks while waiting for a reply. (2) A sends an asynchronous (AS) message to a single destination (SD) and can continue processing because no reply is expected. (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. Thus, the three possible labels for an edge are: (SY, SD), (AS, SD), and (AS, MD). The architecture generation algorithm enforces the folowing constraints: (1) the architecture graph must be connected, (2) multicast messages can only be asynchronous, and (3) a component type can only send a multicast message in response to a message.

The generated architectures are stored in the Architecture DB described next.

3.2 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).

The *Architecture* table has the following columns:

- *ArchitectureId*: unique id for the architecture (primary key).
- *NumComponents*: number of components.
- *NumEdges*: number of edges (connections).
- *NumSyncMessages*: total number of synchronous connections.
- *NumAsyncMessages*: total number of asynchronous connections
- *NumUnicastMessages*: total number of unicast message interfaces.
- *NumMulticastMessages*: total number of multicast message interfaces.
- *ArchComplexity*: architecture complexity, inspired by the cyclomatic complexity for computer programs [9]:

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

  We consider component-type as opposed to component-instance architectures and cluster them into simple, moderate, and complex using k-means clustering [6].
- *ClusterId*: id of the cluster for this architecture.

The columns of the *Component* table are:

- *ComponentId*: unique id of a component (primary key).
- *Type*: type of component (e.g., sender, receiver, sender-receiver, receiver-sender).
- *ArchitectureId*: unique id of the architecture (foreign key).
- *FailureProbability*: probability that a component fails after receiving a message.

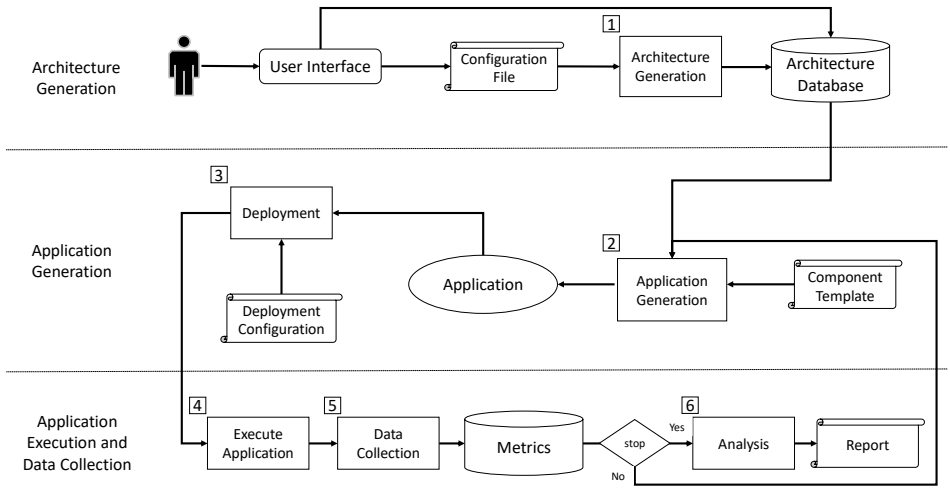

Figure 2: TESS Design

- *AvgMessageProcessingTime*: average time between a component receiving a message and sending a reply.
- *ProbSendSyncMessage*: probability that a message sent by a component is synchronous.
- *ProbSendAsyncMessage*: probability that a message sent by a component is asynchronous.
- *SendSync*: Flag indicating if a component sends synchronous messages.
- *SendAsync*: Flag indicating if a component sends asynchronous messages.
- *RecSync*: Flag indicating if a component receives synchronous messages.
- *ProbSendUnicastMessage*: probability that a message sent by a component is unicast.
- *ProbSendMulticastMessage*: probability that a message sent by a component is multicast.

3.3 Application Generation

The application generation phase of TESS uses a *universal component template* to drive the runtime behavior of the components of the architecture. The universal template component is instantiated at runtime in many different and random ways based on the static and probabilistic attributes of the component for a given architecture stored in the Architecture DB (see Section 3.2). For example, based on the static *Type* attribute, a component can be classified as a *sender*, *receiver*, *sender-receiver*, or *receiver-sender*. As an example of a probabilistic attribute, the *FailureProbability* attribute of a component determines if it fails after receiving a message.

We now describe the universal component template from the point of view of a component S that receives a message m from component C.

If a component is a *sender*, it sends an asynchronous message with probability *ProbSendAsyncMessage* and/or a synchronous message with probability *ProbSendSyncMessage*.

If component S is a *receiver*, it receives message m and fails with probability *FailureProbability* for that component as determined in the Architecture DB. If the component does not fail, it waits for a uniformly distributed time with average *AvgMessageProcessing Time*, specified for component S in the Architecture DB, to simulate the time taken by the component to process the message and act on it. If message m is synchronous, S replies to component C.

If component S is a *sender-receiver*, it sends an asynchronous message with probability *ProbSendAsyncMessage*, sends a synchronous message with probability *ProbSendSyncMessage*, receives message m and fails with probability *FailureProbability*. If S does not fail, it processes message m, and replies to it if m is a synchronous message.

Finally, if S is a *receiver-sender*, it receives message m and fails with probability *FailureProbability*. If S does not fail, it processes the received message, sends an asynchronous message with probability *ProbSendAsyncMessage*, sends a synchronous message with probability *ProbSendSyncMessage*, processes the received message m, and replies to it if m is a synchronous message.

The mechanism for potentially sending an asynchronous message works as follows. A uniformly distributed random number p between 0 and 1 is generated. If this number is less than or equal to the probability that the component sends an asynchronous message, then the component will either send a unicast message or a multicast message.

When sending an asynchronous message, component S sends a unicast message with probability *ProbSendUnicastMessage* or a multicast message with probability *ProbSendMulticastMessage*. In the unicast case, a message is sent to a randomly chosen component consistent with the generated architecture. Multicast messages are sent to the multicast group prescribed by the architecure.

Therefore, different components behave differently from each other because they have different static attributes generated during the architecture generation phase and because of the random values of the probability attributes generated at run-time.

3.4 Application Execution and Data Collection

The various components of the generated application are deployed to 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 the master log into the databases used by TESS. From this merged log, metrics are gathered and stored in the Metrics DB for later analysis.

As mentioned before, metrics are classified into core metrics and RAF-specific metrics. The core metrics gathered during experimentation include: (a) Component Recovery Time: time elapsed since a component failure was detected until it recovered. (b) Node Recovery Time: time elapsed since a node failure was detected until the components running at that node are recovered to a new node. (c) Adaptation Time: time elapsed from start to finish of an adaptation procedure.

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*: average component recovery time.
- *NodeRecoveryTime*: average node recovery time.
- *AdaptationTime*: adaptation time.
- *NumCompFailures*: number of component failures.
- *NumNodeFailures*: number of node failures.
- *NumAdaptations*: number of adaptation events.

The *Architecture* table has a single entry for each generated architecture and the *Components* table has a single entry for each component of a particular architecture. The *Experiments* table has multiple entries for metrics associated with a given architecture. In other words, for each architecture, multiple experiments are conducted and numerous values for each type of metric are collected and stored for later analysis. Because an architecture is associated with an experiment by its ArchitectureId, one may run queries to obtain metrics (e.g., average, coefficient of variation, range and other statistical measures) for either a specific architecture or for all architectures of a specific type.

4 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 [13].
- 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 run-time 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 [1] [2].

5 EXPERIMENTAL PROCESS

Figure 3 depicts the deployment of TESS in a computer cluster that consists of a master node, which acts as a gateway and is connected to the other nodes. The master node hosts the main components of the testbed: a MySQL database for TESS databases, the architecture and application generation modules and the data collection module. Additionally, the master node stores the merged log used to collate all the events from the event logs for all experiments. All other nodes host the RAF and components of the distributed application generated by the application generation module, and local copies of the core and RAF-specific events logs.

We conducted detailed experiments on the use of TESS to automatically evaluate DARE. TESS, which was implemeted in Java, generated 100 random architectures and clustered them according to complexity as 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. Connectors were hosted on separate nodes.

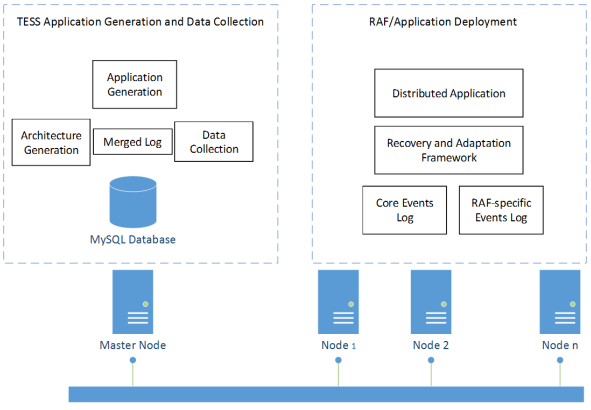

Figure 3: TESS Deployment on a Cluster

We conducted self-healing (component and node failures) and self-adaptation experiments. 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 and then taken down accordingly [3]. Component recovery is done by the RAC instantiating the failed component at another node and replaying the messages (stored in the RAC's 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. See [3] for more details on DARE's approach to failure recovery and adaptation.

5.1 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 [6] was conducted. 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 (i.e., the metric average is the same for all complexity levels), (b) node count has no impact on the given metric (i.e., the metric average is the same for all node counts), and (c) there is no interaction between architecture complexity and node count.

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

Tables 3 and 4 show statistics (average, 95% Confidence Interval (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	± 0.88	21-30
moderate	20.9	± 0.69	17-25
simple	13.8	± 0.91	10-20

5.2 Core Metrics

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

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

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

5.2.1 Component Recovery Time. This experiment assessed the impact of both architecture complexity and different node counts on component recovery time. Tables 5, 6 and 7 show the mean and 95% confidence intervals for component recovery time for each architecture complexity 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. For architecture complexity, $F > F_{\text{crit}}$ results in the rejection of the null hypothesis and acceptance of the alternative hypothesis that the average component recovery time is impacted by architecture complexity. This happens because as architecture complexity increases, a component will communicate with a larger number of neighboring components, resulting in a larger number of reconnections required after recovery. For node count, $F > F_{\text{crit}}$ results in the rejection of the null hypothesis and acceptance of the alternative hypothesis that the number of nodes impacts the average 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 recovery overhead per node for smaller node counts. 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. In other words, we fail to prove the alternative hypothesis that there is interaction between architecture complexity and node count.

Table 5: 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: 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: 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

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

5.2.2 Node Recovery Time. This experiment assessed the impact of architecture complexity and node count on node recovery time. Tables 9, 10 and 11 show the mean and 95% confidence intervals for node recovery time for each architecture complexity 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. For architecture complexity, $F > F_{\text{crit}}$ results in the rejection of the null hypothesis and acceptance of the alternative hypothesis that architecture complexity impacts the average 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. For node count, $F > F_{\text{crit}}$ results in the rejection of the null hypothesis and acceptance of the alternative hypothesis that the number of nodes impacts the average 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 and acceptance of the alternative hypothesis that there is interaction between architecture complexity and node count. 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: 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

Table 10: 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: 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

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

5.2.3 Component Adaptation Time. This experiment assessed the impact of architecture complexity and node count on component adaptation time. Tables 13, 14 and 15 show the mean and 95% confidence intervals for component adaptation time for each architecture complexity 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. For architecture complexity, $F > F_{\text{crit}}$ results in the rejection of the null hypothesis and acceptance of the alternative hypothesis that architecture complexity impacts average 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 [8], 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. In other words, we failed to prove the alternative hypothesis that node count impacts the average 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. In other words, we failed to prove the alternative hypothesis that there is interaction between architecture complexity and node count.

Table 13: 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

6 RELATED WORK

There are two main areas related to our work. The first is the performance evaluation of distributed systems. In [11] Mohamed et

Table 14: 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: 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

al. describe the performance evaluation of distributed event-based systems. Sachs et al. [15] 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. Becker et al. [4] describe an approach to the performance evaluation of self-adaptive systems while Pereira et al. [12] describe the performance evaluation of self-healing systems.

In contrast to the previous works, TESS focuses on both self-healing and self-adaptation frameworks for distributed software systems. 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.

7 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 [5]. For that reason, we decided to design and implement TESS, described above and in more detail in [14], 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. Thus, TESS is complimentary to DARE, which was used as a case study to demonstrate and evaluate TESS.

TESS follows well-known principles of experimental design [6] 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 metrics such as averages, confidence intervals, and statistical procedures such as ANOVA [6]. TESS can also be used to evaluate a RAF on a user-define architecture.

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 [5] 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] Emad Albassam, Hassan Gomaa, and Daniel Menascé. 2016. Model-based Recovery Connectors for Self-adaptation and Self-healing. In *Proc. 11th Intl. Joint Conf. Software Technologies.*

[2] Emad Albassam, Hassan Gomaa, and Daniel Menascé. 2017. Model-Based Recovery and Adaptation Connectors: Design and Experimentation. *Software Technologies* (2017).

[3] Emad Albassam, Jason Porter, Hassan Gomaa, and Daniel Menascé. 2017. DARE: A Distributed Adaptation and Failure Recovery Framework for Software Systems. In *the 14th IEEE International Conference on Autonomic Computing (ICAC).*

[4] Matthias Becker, Markus Luckey, and Steffen Becker. 2012. 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, 117–122.

[5] Lachlana Birdsey, Claudia Szabo, and Katrina Falkner. 2017. Identifying Self-Organization and Adaptability in Complex Adaptive Systems. In *11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems.*

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

[7] Jeffrey O Kephart and David M Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50.

[8] Jeff Kramer and Jeff Magee. 1990. The evolving philosophers problem: Dynamic change management. *IEEE Tr. Software Engineering* 16, 11 (1990), 1293–1306.

[9] T. J. McCabe. 1976. A Complexity Measure. *IEEE Trans. Softw. Eng.* 2, 4 (July 1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

[10] Daniel Menascé, Hassan Gomaa, Joao Sousa, and Sam Malek. 2011. SASSY: A framework for self-architecting service-oriented systems. *IEEE Software* 28, 6 (2011), 78–85.

[11] Saleh Mohamed, Matthew Forshaw, Nigel Thomas, and Andrew Dinn. 2017. 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, 349–352.

[12] E Grishikashvili Pereira, Rubem Pereira, and A Taleb-Bendiab. 2006. Performance evaluation for self-healing distributed services and fault detection mechanisms. *J. Comput. System Sci.* 72, 7 (2006), 1172–1182.

[13] Jason Porter, Daniel Menascé, and Hassan Gomaa. 2016. DeSARM: A Decentralized Mechanism for Discovering Software Architecture Models at Runtime in Distributed Systems. In *11th Intl. Workshop on Models@run.time.*

[14] Jason Porter, Daniel Menascé, Hassan Gomaa, and Emad Albassam. 2017. Design and Experimentation of an Automated Performance Evaluation Testbed for Self-Healing and Self-Adaptive Distributed Software Systems. In *Technical Report GMU-CS-TR-2017-2.* Department of Computer Science, George Mason University.

[15] Kai Sachs, Samuel Kounev, Jean Bacon, and Alejandro Buchmann. 2009. Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. *Performance Evaluation* 66, 8 (2009), 410–434.