DeSARM: A Decentralized Mechanism for Discovering Software Architecture Models at Runtime in Distributed Systems

Jason Porter
Department of Computer Science
George Mason University
Fairfax, Virginia 22030
Email: jporte10@gmu.edu

Daniel A. Menascé
Department of Computer Science
George Mason University
Fairfax, Virginia 22030
Email: menasce@gmu.edu

Hassan Gomaa
Department of Computer Science
George Mason University
Fairfax, Virginia 22030
Email: hgomaa@gmu.edu

***Abstract*—Runtime models play a critical role in modern self-adaptive systems. Hence, runtime architectural models are needed when making adaptation decisions in architecture-based self-adaptive systems. However, when these systems are distributed and highly dynamic, there is an added need to discover the systems software architecture model at runtime. Current methods of runtime architecture discovery take a centralized approach, in which the process is carried out from a single location. These methods are inadequate for large distributed systems because they do not scale up well and have a single point of failure. Also, systems of such size consist of nodes that are typically highly dynamic in nature. Existing approaches to architecture discovery are not capable of addressing these concerns. This paper describes DeSARM (Decentralized Software Architecture discoveRy Mechanism), a completely decentralized and automated approach for runtime discovery of software architecture models of distributed systems based on gossiping and message tracing. DeSARM is able to identify at runtime important architectural characteristics such as components and connectors, in addition to synchronous and asynchronous communication patterns. Furthermore, through its use of gossiping, it exhibits the properties of scalability, global consistency among participating nodes, and resiliency to failures. The paper discusses DeSARM's architecture and detailed design, and demonstrates its properties through experimentation.**

I. Introduction

Software architecture—the high level structure of a software system including a collection of components, connectors and constraints—plays an increasingly critical role in the design and development of any large complex software system. These artifacts (i.e., components, connectors, and constraints) are needed to reason about the system and act as a bridge between requirements and implementation as well as provide a blueprint for system construction and composition. The architecture helps in the understanding of complex systems, supports reuse at both the component and architectural level, indicates the major components to be developed together with their relationships and constraints, exposes changeability of the system, and allows for verification and validation of the target system at a high level [32].

Software architectures have been very influential in self-adaptive systems, i.e., systems that are capable of self-configuration, self-optimization, self-healing and self-protection, and are also called self-* or autonomic systems [17]. In architecture-based self-adaptive systems, components dynamically change in order to continuously adhere to architectural properties and system goals. As a result, these systems require runtime architectural models when making adaptation decisions [11]. However, when these systems are distributed and highly dynamic there is an added need to discover the system's architectural model at runtime.

Current methods of runtime architecture discovery take a centralized approach to constructing architectures dynamically. These methods are inadequate for large distributed systems because they do not scale up well and have a single point of failure. Also, systems of such size are often dynamic in nature due to changes in the runtime environment, where nodes join and leave the system unpredictably, and are subject to failures. Existing approaches do not address the aforementioned concerns.

We describe a method for runtime software architecture discovery in a completely decentralized manner. This is accomplished by keeping message logs at each node and disseminating among the nodes identified component interaction patterns (i.e., synchronous, asynchronous, single or multiple destination) derived from the logs through *selective gossip* exchanges. With selective gossiping, information dissemination takes place only between nodes that are part of the same application. This results in the containment of network traffic, which in effect reduces communication overhead. All references to DeSARM's gossiping method heretofore refer to selective gossiping. Through its use of gossiping, our mechanism addresses the limitations of the current approaches and exhibits the following properties: (1) resiliency to failures, where the gossiping protocol always converges irrespective of node failures, (2) global consistency among participating nodes, where all nodes converge to the same architectural view, and (3) scalability, where the gossiping protocol can accomodate systems of increasing size [29]. Accordingly, the scope of our work is the derivation of architectural models at runtime to be used in decentralized decision making for

architecture-based adaptation in large distributed systems. This effort is part of a larger project, RASS: **R**esilient **A**utonomic **S**oftware **S**ystems [27], aimed at developing a highly-resilient and highly-dependable system, which is capable of making adaptation decisions in a distributed fashion without centralized control. To achieve this goal, each node requires a complete model of the system at runtime.

Thus, the main contribution of this paper is DeSARM: **De**centralized **S**oftware **A**rchitecture discove**R**y **M**echanism, a completely decentralized and automated method and system for runtime discovery of software architecture models of distributed systems using gossiping [6][18] and message tracing. The information dissemination and relatively fast convergence capability of the gossiping protocol aid each node in deriving a complete model of the architecture. We also demonstrate experimentally DeSARM properties of resiliency, global consistency, and scalability. It is important to note that these properties relate to the architecture discovery process and not to application failure recovery or adaptation, which are outside the scope of this paper.

The rest of this paper is organized as follows. Section II discusses related work. Section III provides some basic assumptions. Section IV describes the architecture of DeSARM. Section V presents the results of our experiments with DeSARM. Section VI provides a discussion on important issues related to the architecture discovery mechanism. Finally, Section VII presents some concluding remarks and discusses possible future work.

II. Related Work

In this paper we use architecture discovery instead of recovery as is often found in the literature. Architecture recovery refers to cases where the architecture was known but may have been lost due to erosion or improper documentation [3], [15], [33]. In contrast, architecture discovery refers to situations in which the architecture was not previously known. This is due to the fact that the architecture may not have previously existed or may have changed at runtime due to the dynamic nature of the system as is the case with dynamic software adaptation.

Software architecture discovery approaches can be classified into dynamic, static, and hybrid, i.e., combining both dynamic and static analysis. Some examples of static analysis approaches include [24][4][19] and examples of hybrid approaches are [28][30]. However, as the objective of this paper is discovering architectural models at runtime, we shall focus only on dynamic approaches as follows.

Israr et al. [16] describe SAMEtech, a dynamic approach for automating the discovery of architecture models and layered performance models from message trace information. DiscoTect [31] uses a set of pattern recognizers and knowledge of the architectural style being implemented to map low-level system events into high-level architecturally meaningful events. Bojic and Velasevic [3] use test cases that cover relevant use-cases, and concept analysis to group system entities together that implement similar functionality. Vasconcelos et al. [33] use specified use-cases to generate execution traces from which interaction patterns are identified using pattern detection in order to define architectural elements. Yuan and Malek [35] take a dynamic approach to discovering the architectural model of a distributed application by generating a component interaction model using data mining. Huang et al. [15] use the reflective ability of the component framework to discover an up-to-date architecture from the running system. In contrast to the aforementioned, DeSARM is based on a decentralized approach to architecture discovery. This approach is effective for large distributed systems which pose scalability, single point of failure and dynamicity concerns, which are not addressed by the existing methods.

Also related to our work is architecture-based software dynamic adaptation, which addresses the dynamic software reconfiguration of the architecture model and corresponding implementation for the purpose of runtime adaptation and evolution (see e.g., [10], [21], [23], [25], [34]).

III. Assumptions

This paper makes the following assumptions:

1) *If a component fails, it is restarted on the same node if the node is still running*. This is common in modern component-based systems whereby components can be restarted when they are detected to have failed.
2) *If a node fails, all the components running on that node fail*. This assumption is obvious and does not require further comments.
3) *If a node cannot be restarted after a failure, its components can be moved to other node(s) using an existing component recovery mechanism not within the scope of this work*. This assumption is based on existing work on a runtime application recovery mechanism with which DeSARM is being integrated. See Section VII.
4) *Software components can communicate either through a connection-less transport protocol such as UDP or a connection-oriented protocol such as TCP*. Both types of communication are common in distributed systems and DeSARM supports either.
5) *The software architecture is not known because it may dynamically change due to churn and failures*. As explained in the introduction, this is the focus of this work.
6) *The software deployment on physical nodes is not known.* The set of nodes on which the software system is deployed is not assumed to be known and is discovered by DeSARM.

IV. Software Architecture Discovery Method

This section discusses DeSARM's architecture. We first describe the structure of a node running DeSARM. Then, we give an overview of how gossiping and message tracing are incorporated into the discovery process. We then delve into the details of the architecture model discovery method.

A node in a distributed system consists of three layers according to Fig. 1: application layer, DeSARM layer, and communication middleware. The application layer consists of the distributed application components that communicate

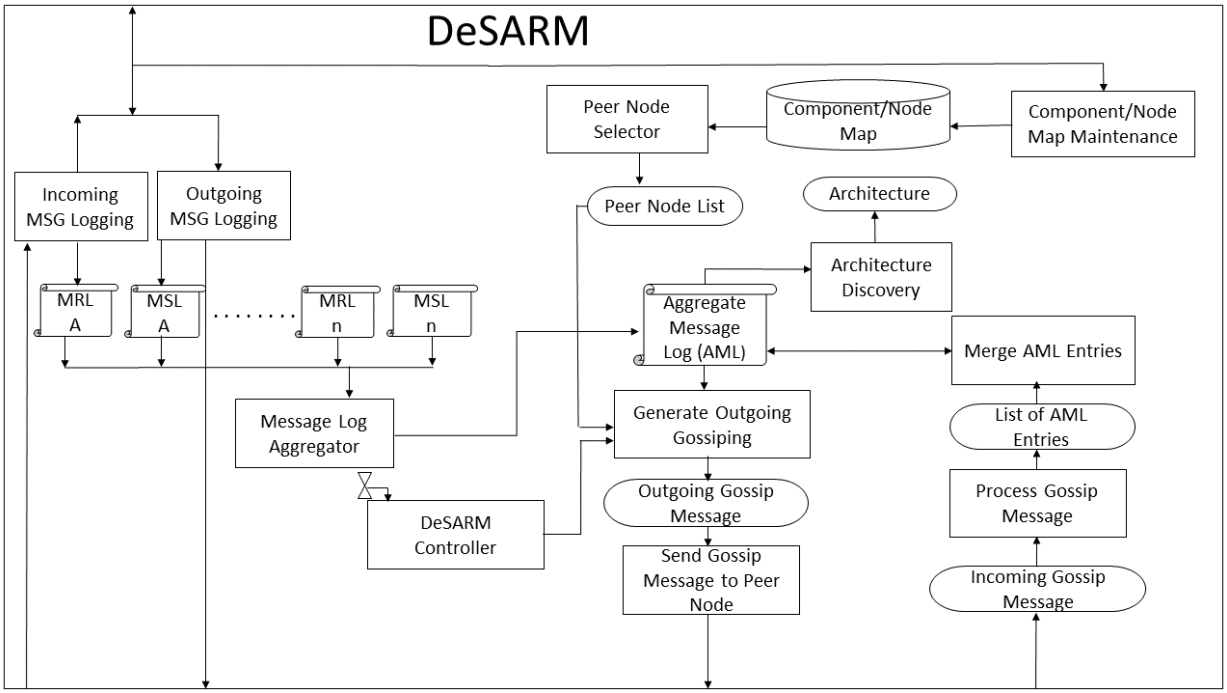

Fig. 2. Details of the DeSARM Layer.

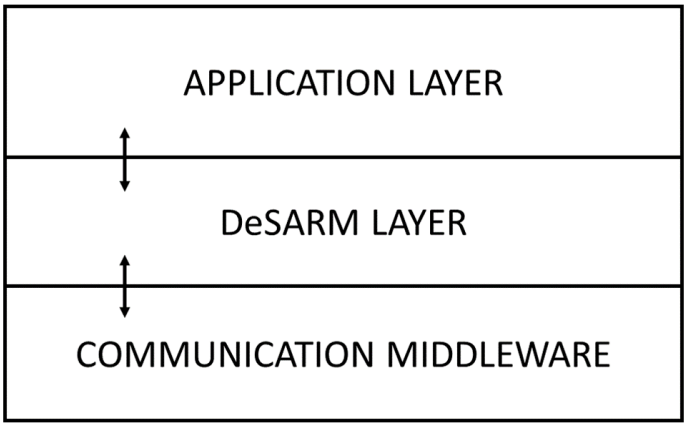

Fig. 1. Layered Structure at a Node of a Distributed System.

over the network via messaging events. Each component has two logs: message sent log (MSL) and message received log (MRL) as shown in Fig. 2. The DeSARM layer forms a wrapper around the communication middleware and provides the same interface to the application layer components as the communication middleware. DeSARM consists of a number of modules each providing different functionality (see Fig.2):

- Message logging: All incoming messages are logged before being passed to the application layer components and all outgoing messages are logged before being passed to the communication middleware. All messages are logged to stable storage.
- Message log aggregation: The MSLs and MRLs of all components are aggregated to form an aggregate message log (AML) at each node. The AML contains the interactions between all components and their respective destination components. This is further merged with the AMLs from incoming gossip messages received from other nodes (see below).
- Gossip-based dissemination: This forms the core of our architecture discovery method and enables the distribution of AMLs from each node throughout the system.
- Peer node selection: This is achieved through the maintenance of a component/node database which is derived by identifying component IDs and their related node IDs from incoming and outgoing messages. This ensures that only nodes running components that are part of the same application are selected for dissemination.
- Control: The DeSARM controller manages the execution of the gossip process by maintaining the timing between consecutive rounds.
- Architecture discovery: This is used to derive the architectural model of the system based on the message traces.

Finally, the communication middleware provides network access allowing the sending and receiving of component-level and gossiping messages between nodes.

Gossip is an epidemic protocol, which due to its simplicity, robustness and flexibility makes it ideal for reliable dissemination of data in large-scale distributed systems. In gossip-based dissemination, data spreads exponentially fast and takes O (log$N$) rounds to reach all N nodes [18]. The essence of this approach, which lies at the core of all gossip-based dissemination approaches, was first introduced in the seminal

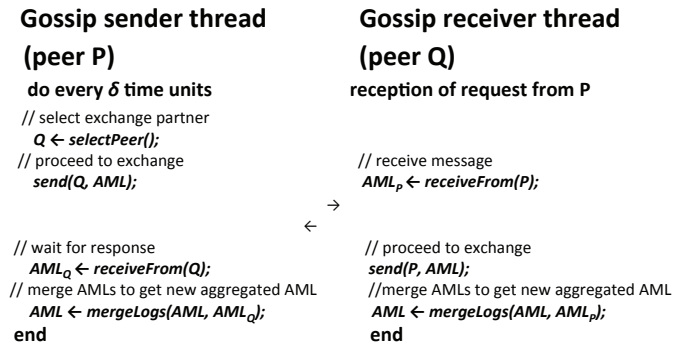

Fig. 3. Design of the Gossip-based Dissemination Framework

paper by Demers et al. [5] and involves the dissemination of data by allowing randomly chosen pairs of nodes to exchange new information. After the exchange, the two nodes forming a pair should have the same information effectively reducing the entropy of the system [29].

The main elements of the gossip-based dissemination framework are: (1) *peer selection*, where a peer (node) selects another peer uniformly at random from the set of available peers, (2) *data exchanged*, which involves the exchange of data between peers and is specific to the use of the gossip mechanism, and (3) *data processing*, which details how each peer handles the information received from other peers and is also specific to the use of the gossip mechanism [18]. DeSARM's gossip-based dissemination is depicted in Fig. 3 and consists of:

- **Peer selection:** A peer P periodically chooses another peer Q uniformly at random from the set of peers that participate in the same application, i.e., DeSARM uses selective gossiping.
- **Data exchanged:** The AML is sent from one peer to another.
- **Data processing:** The received AML is merged with the local AML at each node to produce an updated AML.

Once a node detects that there are no further updates to the gossiped AMLs, it assumes that convergence was reached and a complete software architecture can be derived.

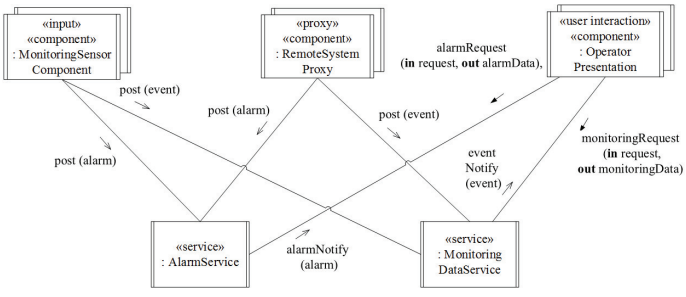

Fig. 4. Emergency Monitoring System.

Each log entry in the MSL or MRL has the following fields:

- Timestamp
- Destination Type: single destination (SD) or multiple destination (MD)
- Message Type (mt): request reply (RQ), no-reply requested (NR), a reply to previous request (RP)
- Transaction ID
- Message Unique ID (mid)
- Return ID: equals 0 if mt $\neq$ RP, otherwise equals "mid" of original request message
- Source Component: component sending the message
- Destination Component: component receiving the message
- Source Node ID: ID of sending node
- Destination Node ID: ID of receiving node

The MRL and MSL for each component are scanned and the interaction patterns for these messages are identified. The following types of messages are considered: Reply requests (RQ), No-reply requests (NR), and Replies (RP).

These message types allows us to identify synchronous vs asynchronous interactions. In the former case, since reply messages are guaranteed to have a request, then the original request reply message and its associated reply message are treated as a single synchronous interaction (SY). If the original message was sent as a unicast (SD) then the tuple (source, destination, SY, SD) is added to the AML. Otherwise, if the original message was sent as a multicast (MD), then the tuple (source, destination, SY, MD) is added to the AML. No-reply requested messages on the other hand are treated as asynchronous interactions (AS) and added to the AML as (source, destination, AS, SD) if the message was sent as a unicast, or (source, destination, AS, MD) if the message was multicasted. The AML is treated as a set so only unique tuple entries are allowed for each component interaction, irrespective of the frequency of such interactions in the message logs because a software architecture does not consider how many times a certain type of interaction occurred between components. Further details on the algorithms implemented by DeSARM can be found in [26].

After each round of gossiping, the updated AML is used to incrementally discover the architecture, which is represented as a labeled directed graph. The vertices of this graph correspond to unique component ids and the edges correspond to unique component interactions. Edges are labeled with the interaction patterns (SY or AS) and destination types (SD or MD). Further details on this process can also be found in [26].

To depict how DeSARM works, we use an example architecture of a distributed emergency monitoring system (see Fig. 4). The architecture consists of five types of components with three instances each of the Monitoring Sensor and RemoteSystem Proxy components, and a single instance of the other components. This example assumes that each component is assigned to a single node. When referring to components we mean component instances unless otherwise mentioned. The communication patterns within the system are:

- Operator Presentation sends synchronous messages with reply to Alarm Service and Monitoring Data Service.

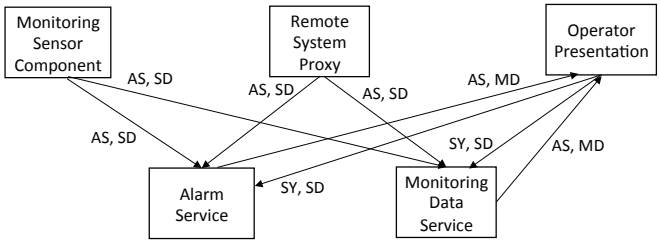

Fig. 5. Discovered Architecture Model as Labeled Directed Graph

- Alarm Service and Monitoring Data Service send asynchronous multicast messages to Operator Presentation.
- Monitoring Sensor and Remote System Proxy send asynchronous unicast messages to Alarm Service and Monitoring Data Service.

The discovered architecture model corresponding to Fig. 4 is the graph shown in Fig.5. This shows the five component types of the system and their respective communication patterns as edge labels, which would have been derived by the architecture discovery process.

V. DeSARM Implementation and Experiments

DeSARM was implemented in Java, and was developed by extending an open source implementation of gossip [14] that manages a list of nodes on a network for cluster membership. We extended the open source gosssip implementation by adding the core functionality of DeSARM, as described in Section IV above, as well as incorporated push-pull based gossip (discussed further below). We emulated a distributed system by implementing each node of the distributed system on a different virtual machine (VM) and spread the VMs over physical machines connected over a network. The VMs communicate over TCP/IP so they can be located anywhere on the network. The DeSARM implementation is heavily multithreaded with different functions of DeSARM implemented as different threads. Some examples of threads include sending and receiving of gossip messages, message log aggregation, architecture discovery, component/node database maintenance, and sending and receiving of component messages. All the communication between nodes uses Java sockets. DeSARM modules normally communicate using UDP, however if gossip exchanges become too large, resulting in message fragmentation in multiple packets, TCP is used to guarantee message integrity.

Our experiments demonstrate the operation of DeSARM and assess its convergence and the number of messages exchanged by the DeSARM middleware. Two sets of experiments were completed. In the first experiment, two types of tests were conducted. In the first test, there were no component/node failures. In the second test, we added random failures with subsequent recovery for each of the components. This second test reveals the impact of failures on the convergence of DeSARM to the final architecture. In the second experiment, the scalability of the mechanism is examined.

A. Experiment I

We use the application described in Fig. 4, whose architecture is known, and show that DeSARM converges exactly to that known architecture. Table I shows the mapping between nodes, components, and physical machines for this architecture. As discussed above, the known and discovered architectures are represented as graphs; we compare the similarity between the two (the known and current version of the discovered architecture) over time. For that, we use the graph comparison algorithms proposed in [20][36][9] and a graph similarity metric that ranges from 0 to 1, where 0 indicates no similarity and 1 indicates that the two graphs are identical. The use of graph comparison using the similarity measure is only for convergence checking during the experiments and is not part of DeSARM's implementation. We plot the evolution of the similarity metric over time to display the convergence speed of the discovery mechanism.

TABLE I
MAPPING OF NODES TO COMPONENTS AND PHYSICAL MACHINES.

Node	Software Component	Machine
Node 1	Monitoring Sensor Component (MSC)	Machine 1
Node 2	Monitoring Sensor Component 2 (MSC2)	Machine 1
Node 3	Monitoring Sensor Component 3 (MSC3)	Machine 1
Node 4	Remote System Proxy (RSP)	Machine 1
Node 5	Operator Presentation (OP)	Machine 2
Node 6	Alarm Service (ArmS)	Machine 2
Node 7	Remote System Proxy 2 (RSP2)	Machine 1
Node 8	Monitoring Data Service (MDS)	Machine 2
Node 9	Remote System Proxy 3 (RSP3)	Machine 2

Figure 6 shows how the architecture converges over time to the known architecture at each of the nodes shown in Table I under a no-failure case. Different nodes converge at different rates but at time 80 sec all nodes have converged to the correct software architecture. Nodes 6 and 9 are the first to converge and node 7 is the last. Note that in our implementation of gossiping, each time a node i sends a gossip message to node j, node j replies with a gossip message. This way, two nodes will exchange AMLs more often, leading to faster convergence. Because of the random nature of peer selection in the gossip protocol, some nodes may gossip more often with others, leading to different convergence rates among nodes. Additionally, the convergence rate is affected by the communication pattern among components.

Figure 7 shows the evolution of the architecture similarity when components fail. As the figure shows, failures start to occur after the first 80 sec, before convergence was achieved. In fact, the value of the similarity metric was equal to 0.9375 (i.e., < 1) at all nodes at $t = 80$ sec. The failure probability of each component, while processing, is set at 20% (a relatively high value) and the average component down time is set at 180 seconds. Thus, at approximately $t = 260$ sec, the failed components will start to recover from the failure and resume sending messages. DeSARM automatically resumes its message collection and gossiping of newly updated AMLs when components start to recover. When that happens, convergence

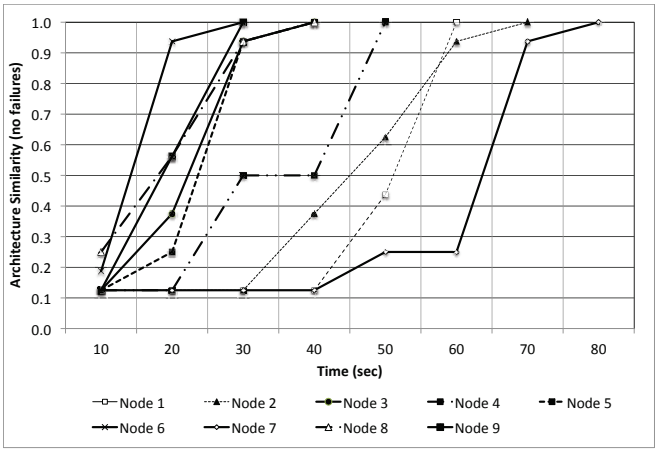

Fig. 6. Architecture similarity at the 9 nodes as a function of time with no failures.

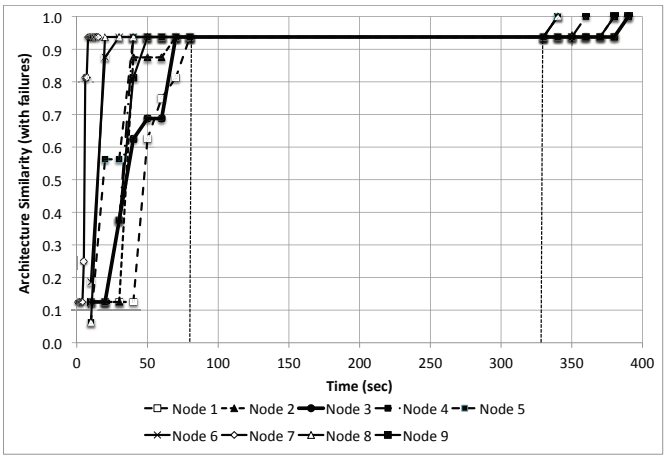

Fig. 7. Architecture similarity at the 9 nodes as a function of time with component failures.

is achieved as illustrated in Table II, which shows the times at which nodes 1-9 converge after failure recovery. As shown in Table II, node convergence times are spread between $t = 340$ sec and $t = 400$ sec.

Figures 6 and 7 are representative of similar results we obtained in other experiments.

TABLE II
CONVERGENCE TIME OF NODES 1-9 UNDER COMPONENT FAILURES.

1	2	3	4	5	6	7	8	9
390	380	390	360	340	340	400	340	380

Table III shows the number of gossip messages sent and received per node until convergence is achieved in the case when failures do not occur and in the case when failures occur. As the table shows, the average number of sent and received gossip messages when no failures occur is almost 1/3 of the corresponding number when there are failures.

For illustration and debugging purposes, each node collected

TABLE III
NUMBER OF GOSSIP MESSAGES SENT PER NODE.

	Sent (no failures)	Received (no failures)	Sent (failures)	Received (failures)
1	24	14	61	36
2	22	15	59	52
3	20	22	63	50
4	22	14	60	55
5	21	19	39	54
6	22	21	60	68
7	25	11	65	69
8	22	32	50	59
9	21	21	63	59
Avg.	22	19	58	56

an event log (not part of DeSARM) during the experiments. Entries in these logs are timestamped in nanoseconds and correspond to events such as sending application-level messages, sending/receiving DeSARM gossip messages, and computing architecture similarity metrics. At the end of the experiment, the event logs of all nodes were sort-merged offline to produce a single log. Figure 8 shows a few excerpts of this log. The first two entries of this log show application-level messages sent by component MSC at Node 1 to components ArmS (at Node 6) and MDS (at Node 8). The next two entries correspond to similar messages sent by MSC2 at Node 2, to components ArmS and MDS. Later in time, DeSARM at Node 1 sends a gossip message to Node 8 with Node 1's current view of the AML, namely [(MSC,ArmS,AS,SD), (MSC,MDS,AS,SD)]. This view only reflects the messages that component MSC at Node 1 sent to nodes 6 and 8. Later in time, DeSARM at Node 8 receives the following gossip message from Node 4: [(RSP,ArmS,AS,SD), (RSP, MDS,AS, SD), (MDS, OP, AS, MD), (MSC3,MDS,AS,SD), (MSC2,MDS,AS,SD), (ArmS,OP,AS,MD), (MSC3, ArmS, AS, SD), (OP, ArmS, SY, SD), (MSC2, ArmS, AS, SD)]. As a result, Node 8's similarity metric becomes 0.6875. Later in time, Node 8 receives a gossip message from Node 9 with an AML that reflects Node 9's current view of the architecture. This AML is aggregated with Node 8's AML resulting in an AML that reflects the entire software architecture. When the similarity metric for Node 8 is next computed, it shows a value of 1, indicating convergence at Node 8.

B. Experiment II

To test the scalability of the approach we tested DeSARM on Argo [2], a high performance computing cluster operated by the Office of Research Computing at George Mason University. For this purpose, we put together a synthetic application with 30 components, each one of them residing on a different node of the research cluster. Some components have a synchronous communication interface only, sending and receiving only synchronous messages, some have an asynchronous communication interface only, sending and receiving only asynchronous messages, while others comprise both synchronous and asynchronous interfaces, sending synchronous messages and receiving asynchronous messages or

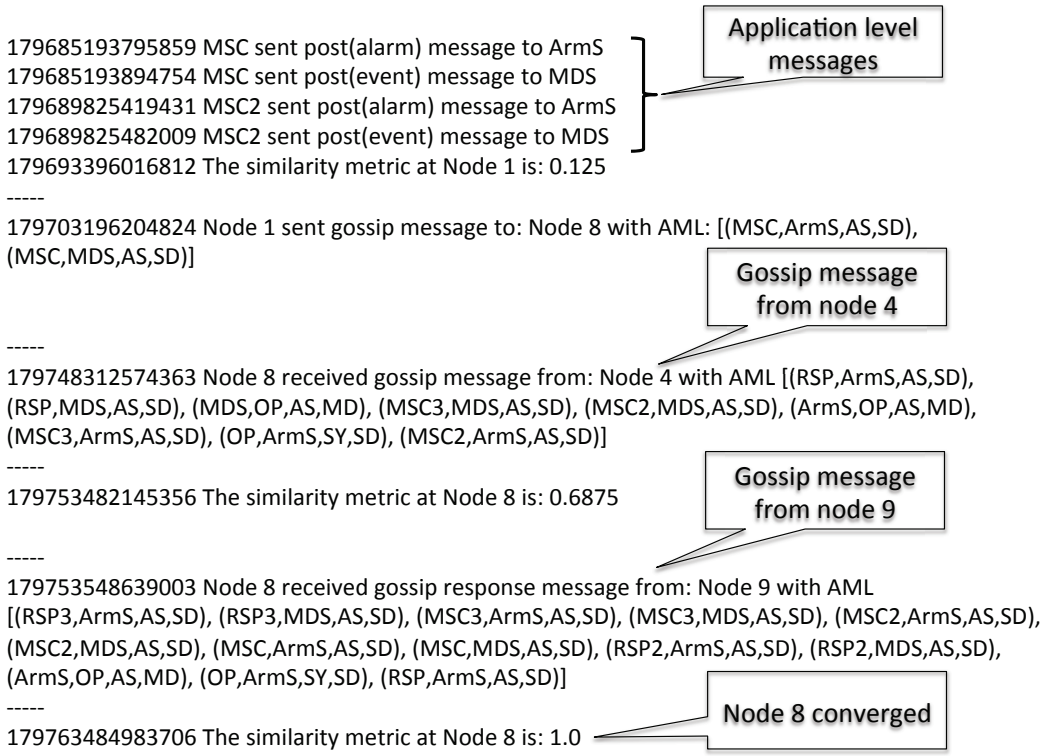

Fig. 8. Excerpts of event trace. Components: Monitoring Sensor (MSC, MSC2, and MSC3), Remote System Proxy (RSP, RSP2, and RSP3), Operator Presentation (OP), Alarm Service (ArmS), and Monitoring DataService (MDS).

vice versa. All communication between components take place at a random time interval to emulate local processing between message exchanges.

Each component communicates with only two other components so that the initial AML at each node is a very small fraction of the complete AML. Therefore, the value of the similarity metric is very small to begin with. It starts at zero in some cases, when components on a node send a synchronous message to another component and have to wait for the reply. This way the DeSARM instance at each node would require more information to derive the complete architectural model than in the previous experiment.

The convergence time at the 30 nodes varied significantly due to the randomness in message exchange. The node that took the longest time to converge converged in 260 sec (i.e., 4.3 minutes) as shown in Table IV. This table shows the progression of the similarity metric over time. The table also shows that the rate of convergence, roughly defined as the increase in convergence over time is slower at the beginning and faster at the end. For example, after 58% of the time, the similarity metric has only achieved the value of 0.27. At 85% of the time, the similarity metrics achieved 0.73. While the slowest node to converge took over four minutes, the fastest took 30 seconds.

VI. Discussion

We discuss here some additional issues of interest, some of which are being addressed in ongoing work.

The first issue we address is whether DeSARM's selective gossiping mechanism always converges, i.e., if there is the possibility of different nodes converging to different discovered architectural models. To address this issue and to ensure convergence, we employed anti-entropy based gossip in DeSARM. In anti-entropy gossip, the gossiping protocol runs indefinitely albeit at specified regular intervals [29]; this means that all nodes continuously make gossip exchanges. Because gossiping never stops, updates will reach all nodes and the system will always eventually converge. However, such a mechanism can result in unnecessary message exchanges especially if no architectural changes have taken place within the system. This problem is exacerbated in large distributed systems. To address this issue we adjusted the protocol to gradually decrease the gossip rate δ when no new updates to the AML have been detected over a period of time. Then, when an update is received, the original gossip rate is restored to speedup convergence.

The second issue is the relationship between gossip convergence speed and the gossip rate δ. A higher value of δ implies faster convergence at the cost of higher message overhead. Other factors such as system size, component communication patterns, and gossip mode (see below) also affect convergence speed. For example, convergence speed is generally affected by the communication pattern among components, including how often they communicate and the number of components they communicate with. Two modes of gossiping can be used:

TABLE IV
SLOWEST CONVEGENCE RATE IN THE 30-NODE EXPERIMENT.

Time (sec)	0-110	120	130	140	150	160-170	180	190-200	210-220	230-250	260
Similarity Metric	0.00	0.10	0.13	0.20	0.27	0.30	0.47	0.57	0.73	0.97	1.00

push gossip (where a peer sends a message but does not receive one in return) vs. push-pull gossip (where a peer sends a message and receives one in return). Our implementation of DeSARM uses push-pull gossip, which accelerates convergence.

The third issue is the implication of dynamic architectural changes during gossiping and how it affects convergence. We are currenty addressing this issue through the integration of DeSARM with an application recovery layer that runs above DeSARM and provides DeSARM with updates to events that cause architectural changes. Examples of such events include component failures as well as component removal due to adaptation. When such events occur, this layer informs DeSARM of the affected component(s). Due to the decentralized nature of DeSARM, this update can be received at any node. Upon receipt, the node will remove from its AML all entries related to the removed component(s) that contain the component(s) as a sender or receiver and use gossip to propogate these deletions to other nodes within the system. Note that architectural changes that result in new components being added are handled automatically by DeSARM as described here.

The fourth issue is the overhead of the gossip mechanism in relation to both network traffic as well as latency in the sending and receiving of component messages. DeSARM has very little network overhead for two reasons: the size of the AMLs and its use of selective gossiping. As the AMLs contain only unique tuple entries of component interactions, the size of the AMLs are kept small. This in effect reduces the overall bandwidth needed for gossip exchanges. Also, as mentioned earlier, selective gossiping allows for the containment of network traffic to only those nodes that are part of the same application which further reduces the communication overhead of the mechanism. With respect to the latency of component messaging, DeSARM can keep it at a minimum because, as previously mentioned, all major functionality is separated into different threads. This allows for all other processing to take place in the background thus reducing any major impact the gossip mechanism will have handling component messages.

VII. CONCLUDING REMARKS

This paper described DeSARM, a completely decentralized and automated approach for software architecture discovery of distributed systems based on gossiping and message tracing. Through message tracing, DeSARM is able to identify important architectural characteristics such as components and connectors in addition to synchronous and asynchronous communication patterns. Furthermore, through the use of gossiping, DeSARM exhibits the properties of scalability, global consistency among participating nodes, and resiliency to failures. These properties were demonstrated through various experiments, with and without component failures. These experiments assessed the rate of convergence of the DeSARM nodes towards the software architecture being discovered. These experiments showed that DeSARM is resilient and is able to discover the architecture even in the presence of failures, albeit at a lower pace than the one when no failures occur. DeSARM was implemented in Java using a multi-threaded architecture.

In future work we plan to examine DeSARM's capability of discovering architectures that change over time as in the case of dynamic software adaptation [13], [12]. This work will examine different recovery and adaptation models as well as dynamic components, i.e., components that change their communication patterns at runtime based on their state or phase of execution, and the impact on DeSARM. Also of interest, is addressing the mechanism's capacity for discovering the full behavioral architecture of a system such as identifying component interaction protocols.

In the current experiments, we used a fixed architecture to illustrate how DeSARM converges with and without failures and how it operates equally well with two or thirty physical machines. In the future, we will run experiments in which a variety of software architectures are generated following certain distributions for parameters such as number of components and communication pattern types.

As mentioned previously, DeSARM is being integrated with an existing application recovery mechanism (ARM) that allows for the runtime recovery of component(s) due to node failure [1]. Through the peer sampling capability of gossip [18], DeSARM is able to identify suspected node failures and alert the ARM. Once the node failure has been verified, the ARM proceeds to recover the failed component(s) to a new node and re-instantiates them based on the current architecture provided by DeSARM. As a result of the decentralized nature of DeSARM, component recovery can be initiated from any node as each maintains a complete model of the architecture. This integration will allow for the development of applications that are resilient to failures.

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] Albassam, Emad, et al. “Model-based Recovery Connectors for Self-Adaptation and Self-Healing.” Proc. 11th Intl. Joint Conf. Software Technologies (ICSOFT 2016), July 24-26, 2016, Lisbon, Portugal.

[2] http://orc.gmu.edu/research-computing/argo-cluster/

[3] Bojic, Dragan, and Dusan Velasevic. “A use-case driven method of architecture recovery for program understanding and reuse reengineering.” IEEE Conf. Software Maintenance and Reengineerig, 2000, pp. 23–32.

[4] Corazza, Anna, Sergio Di Martino, and Giuseppe Scanniello. “A probabilistic based approach towards software system clustering.” IEEE 14th Conf. Software Maintenance and Reengineering, 2010, pp. 88-96.

[5] Demers, Alan, et al. “Epidemic algorithms for replicated database maintenance.” Proc.Sixth Annual ACM Symp. Principles of distributed computing. ACM, 1987.

[6] Dulman, Stefan, and Eric Pauwels. “Self-Stabilized Fast Gossiping Algorithms.” ACM Tr. Autonomous and Adaptive Systems (TAAS), 10.4 (2015): 29.

[7] Esfahani, Naeem, Eric Yuan, Kyle R, Canavera and Sam Malek. “Inferring Software Component Interaction Dependencies for Adaptation Support.””ACM Tr. Autonomous and Adaptive Systems (TAAS), 10.4 (2016): 26.

[8] Ewing, John, and Daniel A. Menascé, “A Meta-controller method for improving run-time self-architecting in SOA systems,” Proc. 5th ACM/SPEC Intl. Conf. Performance Engineering (ICPE 2014), Dublin, Ireland, March 23-26, 2014.

[9] Foggia, Pasquale, Carlo Sansone, and Mario Vento. “A performance comparison of five algorithms for graph isomorphism.” Proc. 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition. 2001.

[10] Garlan, David, et al. “Rainbow: Architecture-based self-adaptation with reusable infrastructure.” Computer 37.10 (2004): pp. 46–54.

[11] Garlan, David, and Bradley Schmerl. “Using architectural models at runtime: Research challenges.” European Workshop on Software Architecture. Springer Berlin Heidelberg, 2004.

[12] Gomaa, Hassan, and Koji Hashimoto, 2012. “Dynamic self-adaptation for distributed service-oriented transactions,” Proc. 7th Intl. Symp. Softw. Eng. for Adaptive and Self-Managing Systems, SEAMS 12. IEEE Press, Piscataway, NJ, USA, pp. 11-20.

[13] Gomaa, Hassan, Koki Hashimoto, Minseong Kim, Sam Malek, and Daniel A. Menascé, 2010. “Software adaptation patterns for service-oriented architectures,” Proc. 2010 ACM Symp. Applied Computing, New York, NY, USA, pp. 462469. doi:10.1145/1774088.1774185.

[14] https://code.google.com/archive/p/java-gossip/

[15] Huang, Gang, Hong Mei, and Fu-Qing Yang. “Runtime recovery and manipulation of software architecture of component-based systems.” Automated Software Engineering, 13.2 (2006): pp. 257–281.

[16] Israr, Tauseef, Murray Woodside, and Greg Franks. “Interaction tree algorithms to extract effective architecture and layered performance models from traces.” J. Systems and Software 80.4 (2007): pp. 474–492.

[17] Kephart, Jeffrey O., and David M. Chess. “The vision of autonomic computing.” IEEE Computer 36.1 (2003): pp. 41–50.

[18] Kermarrec, Anne-Marie, and Maarten Van Steen. ”Gossiping in distributed systems.” ACM SIGOPS Operating Systems Review 41.5 (2007): 2-7.

[19] Koschke, Rainer. “Architecture reconstruction.” Software Engineering. Springer Berlin Heidelberg, 2009. pp. 140–173.

[20] Koutra, Danai, et al. Algorithms for graph similarity and subgraph matching. Technical Report Carnegie-Mellon-University, 2011.

[21] Kramer, Jeff, and Jeff Magee. “Self-managed systems: an architectural challenge.” Future of Software Engineering, 2007. FOSE’07. IEEE, 2007.

[22] Menascé, Daniel A., John Ewing, Hassan Gomaa, Sam Malek, and Joao P. Sousa, “A framework for utility-based service-oriented design in SASSY,” First Joint WOSP/SIPEW Intl. Conf. Performance Engineering, San Jose, CA, USA, January 28-30, 2010.

[23] Menascé, Daniel A., Hassan Gomaa, Sam Malek, and Joao P. Sousa. ”SASSY: A framework for self-architecting service-oriented systems.” Software, IEEE 28.6 (2011): pp. 78–85.

[24] Naseem, Rashid, Onaiza Maqbool, and Siraj Muhammad. “Improved similarity measures for software clustering.” , 2011 15th IEEE European Conf. Software Maintenance and Reengineering (CSMR), 2011.

[25] Oreizy, Peyman, et al. “An architecture-based approach to self-adaptive software.” IEEE Intelligent systems 14.3 (1999): pp. 54–62.

[26] Porter, Jason, Daniel A. Menascé, and Hassan Gomaa, “Decentralized Software Architecture Discovery in Distributed Systems.” Technical Report GMU-CS-TR-2016-2, Dept. of Computer Science, George Mason University, 2016. (http://cs.gmu.edu/ tr-admin/papers/GMU-CS-TR-2016-2.pdf).

[27] RASS Project, http://cs.gmu.edu/˜menasce/rass/

[28] Riva, Claudio, and Jordi Vidal Rodriguez. “Combining static and dynamic views for architecture reconstruction.” Proc. 6th IEEE European Conf. Software Maintenance and Reengineering, 2002.

[29] Riviere, Etienne, and Spyros Voulgaris. “Gossip-based networking for internet-scale distributed systems.” E-Technologies: Transformation in a Connected World. Springer Berlin Heidelberg, 2011. pp. 253–284.

[30] Sartipi, Kamran, and Nima Dezhkam. “An Amalgamated Dynamic and Static Architecture Reconstruction Framework to Control Component Interactions 259.” 14th IEEE Working Conf. Reverse Engineering, 2007.

[31] Schmerl, Bradley, et al. “Discovering architectures from running systems.” IEEE Tr. Software Engineering, 32.7 (2006): pp. 454–466.

[32] Taylor, Richard N., Nenad Medvidovic, and Eric M. Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[33] Vasconcelos, Aline, and Cludia Werner. “Software architecture recovery based on dynamic analysis.” Brazilian Symp. on Softw. Engineering. 2004.

[34] Weyns, Danny, and Tanvir Ahmad. “Claims and evidence for architecture-based self-adaptation: A systematic literature review.” Software Architecture. Springer Berlin Heidelberg, 2013. pp. 249–265.

[35] Yuan, Eric, and Sam Malek. “Mining Software Component Interactions to Detect Security Threats at the Architectural Level.” Proc. 13th Working IEEE/IFIP Conf. Software Architecture. 2016.

[36] Zager, Laura A., and George C. Verghese. “Graph similarity scoring and matching.” Applied mathematics letters 21.1 (2008): pp. 86–94.