Emulating Web Services-Based Systems Hosted in Ad Hoc Wireless Networks

Petr Novotny and Alexander L. Wolf

Imperial College London
London, UK

Imperial College London
Department of Computing
Technical Report DTR-2016-7 March 2016

ABSTRACT

Abstract

The emerging use of ad hoc wireless networks combined with current trends in the use of Web Services-based systems pose new challenges to accurate emulation of these systems. Current network simulators lack the ability to replicate the complex message exchange behavior of services while service simulators do not accurately capture mobile network properties. In this paper, we provide an overview of a generic Web Services framework for emulating both a service behavioral model and an ad hoc wireless network. The framework is implemented as a generic Web Services system in Java EE hosted in the CORE/EMANE emulation environment.

1 Introduction

The deployment of Web Services-based systems in ad hoc wireless networks is rapidly evolving with increasing use of this type of network environments [4, 5]. Structured as a complex of interdependent, interrelated services, they can provide end users with a rich set of information combined from multiple sources. This model of information delivery is in sharp contrast to the flow-based, point-to-point model that has been traditionally supported in ad hoc wireless networks. Analyzing the behavior of such complex applications is a challenging task. The challenge is compounded when services run on ad hoc wireless networks with highly *dynamic topology* such as mobile ad hoc networks (MANETs), hybrid wireless networks (HWNs), vehicular ad hoc networks (VANETs) and others in which the fluidity of the underlying network greatly impacts performance and availability.

Web Services-based systems consist of some number of services that interact with one another in order to complete client requests. Each service provides a set of methods for use by other services or clients. A method may use any number of other methods provided by other services to carry out its functionality. Thus, services are interconnected with each other. Clients initiate the flow of service requests by sending messages that request method executions, and then wait for some response. In the environment of networks with dynamic topology, the availability of services hosted on network nodes is unknown at design time. Thus, the services must discover and bind to other services they use at runtime. This behavior is enabled by service discovery mechanisms specifically tailored to support service discovery and binding in the changing environment of the networks with dynamic topology.

An emulator that can closely replicate the behaviors of Web Services-based systems running on the network with dynamic topology can be a valuable analysis tool. In particular, it can provide a means to predict performance when designing, deploying, or managing the system. It can help with validation and verification of system management methods designed for this type of environment. Furthermore, it can model network traffic workloads that are characteristic of dynamic topology networks. Existing simulation and emulation tools fall short of providing such capabilities. Packet-level network simulators, such as NS-3[1] and QualNet[2] provide detailed implementations of mobile, wireless networks, but lack the ability to replicate complex behavioral aspects of service-based systems. These aspects are addressed in high-level service simulators [12], which unfortunately do not provide a means to simulate a complex network layer.

In our early work, we introduced NS-3 based Service-based system simulator [9], successfully used in design and evaluation of several service-based system management methods [6, 8, 10, 11, 13]. This simulator provides an approach to simulate service-based systems and related mechanisms in the early stage of design and development. However, this framework is based on the NS-3 simulator and hence uses approximations of many aspects of the network behavior, components, and protocols, and thus is missing some of the important network characteristics and details. The actual implementations of service-based systems hosted in networks with dynamic topology need to address problems of connectivity between nodes and consequently their performance is sensitive to the details of the network environment. The connectivity between nodes depends on a combination of several network aspects, each playing an important part in the overall outcome. Hence, experimental tool closely replicating the peculiarities of the networks with a dynamic topology is required in order to provide an accurate environment for evaluation of the systems and methods.

In this paper, we introduce a new framework for realistic emulation of Web Services-based systems hosted on networks with dynamic topology. With system and behavioral models of Web Services built on top of the high fidelity network emulator, our approach allows the replication of various critical aspects, such as the cascading flows of messages in complex conversations, comprehensive client-driven workload profiles and the propagation of faults through services. Furthermore, the emulator provides generic and easily extendable models that can be used to capture modern Web Services-based platforms, such as SOA, operating in networks with dynamic topology.

We have used the emulator for evaluation of several system management methods such as in the method of Delay Tolerant Harvesting of Monitoring Data for MANET-Hosted Service-Based Systems [7] or methods

[1] `http://www.nsnam.org/`

[2] `http://www.scalable-networks.com/products/qualnet/`

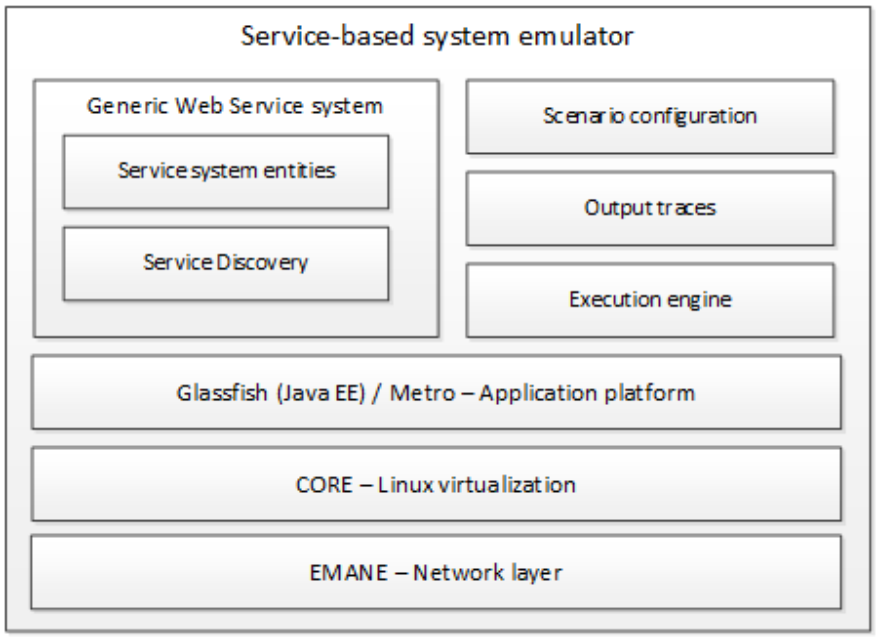

Figure 1: Architecture of the generic Web Services-based emulation framework

of Dependence Discovery and Fault Localization in Service-Based Systems Hosted in MANETs [6].

A source code of components of the emulator is available for download at the following address: https://github.com/jcmldev/generic-webservice-system

2 Architecture

The generic Web Services framework consists of network and application layer components. Figure 1 illustrates the architecture of our emulation framework.

2.1 Network layer

On the network layer is used the Common Open Research Emulator (CORE) [3] and Extendable Mobile Ad-hoc Network Emulator (EMANE)[4].

The CORE is a virtualization tool for emulation of computer networks [2, 1]. The CORE runs on Linux platform and uses native Linux mechanism of process isolation (namespace isolation) to create low overhead virtual machines. Within the CORE virtual machines is provided standard Linux operating system environment with all of the native system functionalities. The virtual machines are connected to the network provided and managed by the EMANE.

The EMANE provides real-time modeling of link and physical layer connectivity so that network protocol and application software can be experimentally subjected to the same conditions that are expected to occur in real wireless ad hoc networks. In the EMANE, only the two lowest levels of the OSI model stack are emulated (i.e. the physical layer and the data link layer) and on all layers above them are used real network protocols provided by the Linux operating system.

The combination of these two tools provides high fidelity real-time emulation environment for evaluation of distributed systems. The setup and integration of the CORE and EMANE is a complex matter discussed

[3] `http://cs.itd.nrl.navy.mil/work/core/index.php`
[4] `http://cs.itd.nrl.navy.mil/work/emane/index.php`

by Ahrenholz et al. [3].

2.2 Application layer

The emulation tools provide bare Linux operating system environment of nodes connected into the wireless ad hoc network. Within the Linux environment is hosted the generic Web Service system.

The generic Web Service system is built on Java platform. As an application platform, we use reference implementation of Java EE, Glassfish [5]. The Glassfish is an open-source software and provides a full implementation of the Java application server environment. The Glassfish server is a container within which additional server components are installed. For the hosting of the web services, we use Glassfish Metro [6]. The Metro is a reference implementation of a standard Java web services stack. The generic Web Service system provides abstractions for entities (services and clients) and their interconnection models.

Finally, the emulator provides methods for engineers to configure the emulation scenarios and their parameters, to run the emulation, and to generate output traces. In what follows, we describe the models and their current implementation.

3 Application layer components

3.1 Generic Web Services system

The generic Web Services system consists of several abstraction models of: entities, interconnections, workloads, messages and deployment.

3.1.1 Entity model

Entity model provides the building blocks of the generic Web Service system.

- *Client* represents application used by an end user. Each client behaves as an autonomous entity that contacts a set of Web Services at times (random or deterministic) configured by the engineer.
- *Service* represents autonomous self-contained functional unit. Each service has a set of methods that are available to be used by clients and other services. Each method contains an abstract definition of its computation consisting of delays to simulate processing time and a set of steps that send requests to other services.

3.1.2 Interconnection model

The interconnection model defines the methods in other services with which each entity (Client or Service) in the system interacts (i.e., sends service requests and receives responses). Two types of interconnections are defined: client-to-service, and service-to-service. The system provides a deterministic generator of the system interconnections, which allows the engineer to have control over the specific interactions in the system.

The interconnection model also defines the *service discovery* mechanism used during the system runtime. The service discovery mechanism provides functionality to the clients and services to discover service instances to send messages to during the system runtime. It is an essential component in service-based system where interconnections are defined between entities and contracts (i.e. types of services) but not services themselves. Thus, during the system runtime, the entities have to discover which actual service instance they should interact with based on the type of the service instances and their availability. This task is particularly important in the wireless ad hoc network where the dynamic topology is continuously altering connectivity between nodes and consequently the availability of services.

[5] `https://glassfish.java.net/`
[6] `https://metro.java.net/`

In general, before sending a request, the system entity will query a service registry for a service instance to send the request to. The service registry will select the most appropriate service instance based on its availability or some other metric. The emulator provides a default discovery mechanism; based on metric from routing tables (i.e. hop distance).

3.1.3 Message model

In a typical service-based system, there are three types of messages exchanged between entities: requests, responses, and exceptions. Request messages are used to invoke methods in other services while response messages are sent by services back to the requesting entity upon the completion of the requested method. Exception messages are used to propagate fault symptoms caused by network or service faults.

The generic Web Service system is implemented as using SOAP protocol. However, the system can be also configured to use REST or some other RPI protocol.

The flow of messages exchanged between services during the processing of a client request is called a *conversation*. In the generic system, all messages contain information about the conversation to which they belong. The conversation information is designed as the behavior of WS-* standard, namely the WS-Addressing.[7]. Moreover, every message exchanged between entities of the system contains the following fields:

- *ConversationID* an integer identifier of the conversation the message is part of
- *From* a URL of the service which sent request between the pair of services

3.1.4 Workload model

In Web Service systems the workload is initiated by clients sending requests to services. The workload model defines the rates of such requests. The client repeatedly, and at pre-configured random times selects one method to request out of the set of available service methods and then waits for a response.

In Figure 2 is shown client workload algorithm. In our current implementation, the method is selected uniformly at random from the set of available methods. For each client, the set of the available service methods is defined by the interconnection model.

Upon reception of a request, the requested service method is invoked and depending on the configuration of interconnections, further messages will be sent to other services. In Figure 3 is shown the processing of a request by a service.

3.1.5 Deployment model

The deployment model specifies the mapping between physical network nodes and the instances of entities of the system (i.e. clients and services).

3.2 Scenario configuration

The configuration of the emulation scenario has two steps. In the first step, the definition of the network is designed in CORE scenario editor. The design ofnetwork scenarios is described in CORE documentation[8]. In the second step, a corresponding configuration of the generic Web Service system to be hosted in the network is defined. The configuration of the service system is stored in an XML file. The file contains definition and values of parameters of the entity, interconnection, workload and deployment models of thegeneric Web Service system. An example of a system scenario is provided in Listing 1.

The configuration file contains the following parts: (Note, only the key aspects of the configuration are described in the followingtext. Detailed description is provided within the system components.)

[7] `http://www.w3.org/Submission/ws-addressing/`

[8] http://downloads.pf.itd.nrl.navy.mil/docs/core/core-html/usage.html

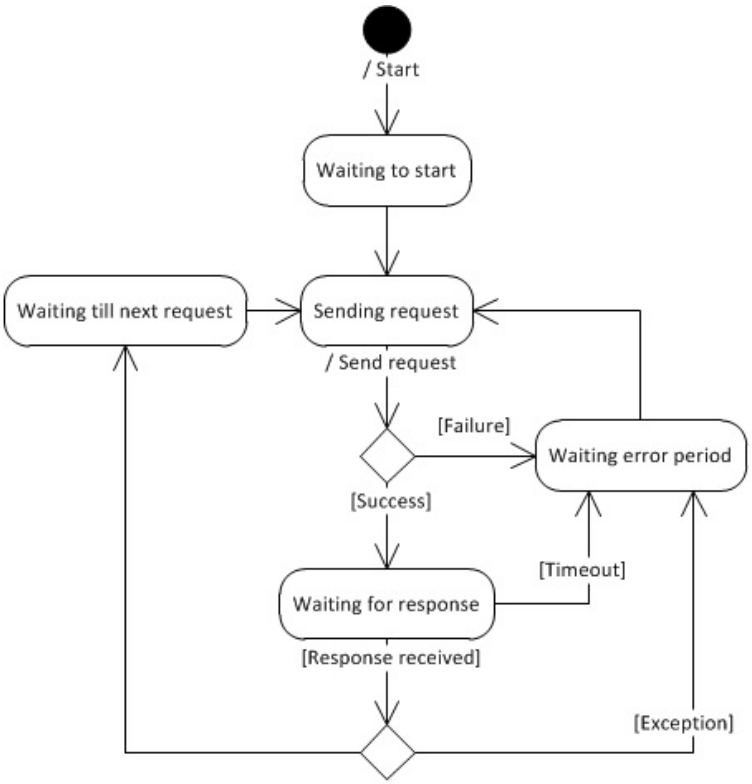

Figure 2: State-machine of the client workload algorithm

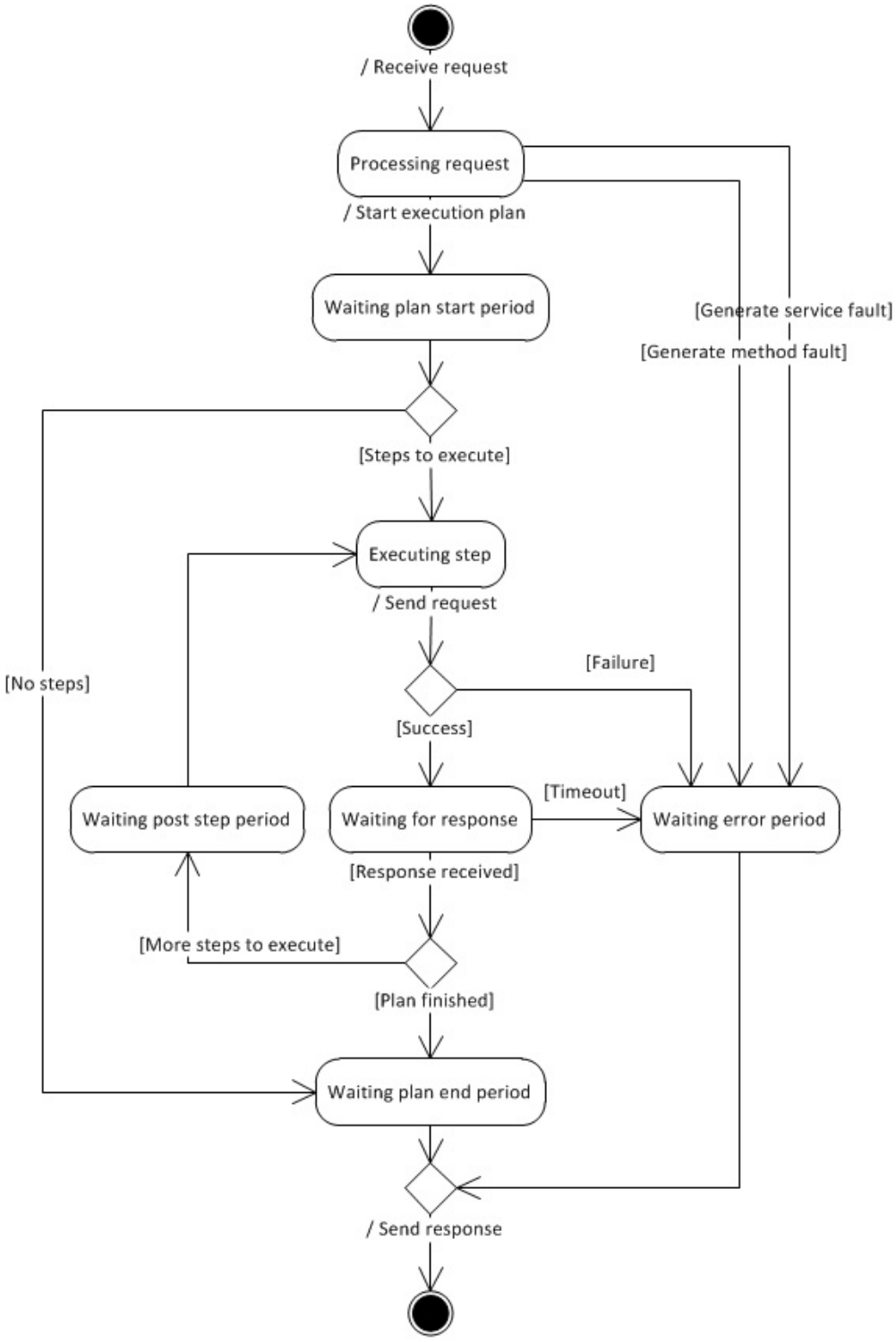

Figure 3: State-machine of the service workload algorithm

3.2.1 Experiment

In the element *experiment* are defined parameters of the experiment, such as the response message timeout or a naming pattern of services hosted in the Java EE server.

3.2.2 Clients

In the element *clients* are defined client applications. Each element *client* defines a single instance of a client deployed and instantiated ona node. The client element contains the *ip* attribute referring to an IPaddress defined in CORE network scenario. The client element further contains a definition of the execution plan. The plan consists of a series of steps, each referring to a particular type of service and method of that service. The seriesof steps is executed by the client application at the scenario runtime. The steps are executed in the order as listed in the configuration. Once all steps have been executed, the client will execute the whole plan again starting from the first step, repeating this process till the client is stopped. After a step is completed either by receiving a response message or fault, the client will wait for a period of time (i.e. delay) to execute a next step in the plan. The delay is defined in client element in the attributes *stepDelayDistributionStart* and *stepDelayDistributionEnd*. These attributes define the minimum and maximum values of the uniform random distribution from which the delay is taken.

The scenario allows to define a single *default client* configuration by setting the attribute *default* to the value *true* on one of the client elements. If the scenario contains such a configuration, the execution engine will instantiate a single client application on each network node with the default client configuration parameters. This feature allows simplifying theconfiguration of clients in scenarios where all clients are expected to behave uniformly.

3.2.3 Services

In the element *nodes* are defined service instances and their deployment onto the network nodes. Each node which hosts one or more services is defined byan element *node* with attribute *ip* referring to the IP address ofnode defined in the CORE scenario definition. Each element node contains a definition of services hosted on it. The element *service* defines an instance of a service hosted on the node with the name defined in attribute *name*. The element service further contains elements *method*, each defining method exposed by the service for invocation. Each method contains a definition of an execution plan. The definition of the execution plan is same asthat of the client. The method further contains a definition of delay used before the first step of the plan is executed and after the last step of the plan is executed. Each step also contains definition of delay executed after the step has finished.

Hence, the method execution works as follows: when the service method is invoked, the processing first executes the delay defined before the start of thefirst step, then in sequence all steps are executed, after each step the relevant delay is executed, at the end of the plan, the delay defined as the final delay of the plan is executed.

In larger scenarios involving tens or hundreds of network nodes and service instances, the configuration of services may involve a large number of configuration elements. For these cases, the software suite provided contains anapplication which allows generating this part of the configuration algorithmically.

Listing 1: An example XML configuration file. (Due to the extensive length of the file this example contains only subset of the scenario definition.)

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- all delays are in milliseconds -->
<dd>
    <experiment
        useSingleMachineDebugCode="false"
        serviceSystemMessageExchangeTimeout="60000"
    serviceSystemMessageExchangeTimeoutAdditionalWraperTime="2000"
        serviceSystemGenericServiceAddress="http://{ip}:8080/{service_name}/GWS"
        />
```

```
<clients>
  <!--
    ip - ip of node on which the configured client is deployed
    conversationIdSeed - seed of ids for this particular client
    stepDelayDistributionStart - lower value of distribution of period between requests
    stepDelayDistributionEnd - uppper value of distribution
  -->

<!-- default client configuration -->
    <client
        ip="0.0.0.0"
        default="true"
        conversationIdSeed="10000"
        stepDelayDistributionStart="2000"
        stepDelayDistributionEnd="4000">
      <step targetMethod="0" targetService="WS1"/>
      <step targetMethod="1" targetService="WS1"/>
      <step targetMethod="0" targetService="WS2"/>
      <step targetMethod="1" targetService="WS2"/>
      <step targetMethod="0" targetService="WS3"/>
      <step targetMethod="1" targetService="WS3"/>
      <step targetMethod="0" targetService="WS4"/>
      <step targetMethod="1" targetService="WS4"/>
      <step targetMethod="0" targetService="WS5"/>
      <step targetMethod="1" targetService="WS5"/>
    </client>

    <client
        ip="10.0.0.1"
        default="true"
        conversationIdSeed="10000"
        stepDelayDistributionStart="2000"
        stepDelayDistributionEnd="4000">
      <step targetMethod="0" targetService="WS1"/>
      <step targetMethod="1" targetService="WS1"/>
      <step targetMethod="0" targetService="WS2"/>
      <step targetMethod="1" targetService="WS2"/>
    </client>

</clients>

<!-- methods are indexed/referenced according to their order -->
<nodes>
    <node ip="10.0.0.1">
        <service name="WS1">
            <method
                startDelayDistributionStart="10"
                startDelayDistributionEnd="20"
                endDelayDistributionStart="10"
                endDelayDistributionEnd="20">
                <step
                    endDelayDistributionStart="10"
                    endDelayDistributionEnd="20"
                    targetMethod="0"
                    targetService="WS102"/>
            </method>
            <method
                startDelayDistributionStart="10"
                startDelayDistributionEnd="20"
                endDelayDistributionStart="10"
                endDelayDistributionEnd="20">
                <step
                    endDelayDistributionStart="10"
                    endDelayDistributionEnd="20"
```

```
                    targetMethod="1"
                    targetService="WS103"/>
            </method>
        </service>
        <service name="WS101">
            <method
                startDelayDistributionStart="10"
                startDelayDistributionEnd="20"
                endDelayDistributionStart="10"
                endDelayDistributionEnd="20">
                <step
                    endDelayDistributionStart="10"
                    endDelayDistributionEnd="20"
                    targetMethod="0"
                    targetService="WS106"/>
                <step
                    endDelayDistributionStart="10"
                    endDelayDistributionEnd="20"
                    targetMethod="0"
                    targetService="WS202"/>
            </method>
            <method
                startDelayDistributionStart="10"
                startDelayDistributionEnd="20"
                endDelayDistributionStart="10"
                endDelayDistributionEnd="20">
                <step
                    endDelayDistributionStart="10"
                    endDelayDistributionEnd="20"
                    targetMethod="1"
                    targetService="WS107"/>
                <step
                    endDelayDistributionStart="10"
                    endDelayDistributionEnd="20"
                    targetMethod="1"
                    targetService="WS203"/>
            </method>
        </service>
    </node>
  </nodes>
</dd>
```

3.3 Output traces

During the emulation run, the system components record events, such as service message exchanges, invocation of methods and fault symptoms, into trace files for posterior analysis. A detailed description of the structure and content of the trace files is provided within the components.

4 Conclusion

In this paper, we have introduced our emulation framework of Web Service-based systems hosted in wireless ad hoc networks. The emulator is designed as a combination of CORE and EMANE network emulation engine combined with generic Web Services system implemented in Java EE. The emulator thus closely replicates the complex network behavior as well as the Web Service-based system entities and models. We have used the emulator for evaluation of several system management methods such as in the method of Delay Tolerant Harvesting of Monitoring Data for MANET-Hosted Service-Based Systems [7] or methods of Dependence Discovery and Fault Localization in Service-Based Systems Hosted in MANETs [6].

To our knowledge, there is currently no other comparable tool providing the functionality of network and service layers emulation. Therefore, we believe that the emulator can be a valuable tool for many other

researchers as well.

Acknowledgments

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

[1] J. Ahrenholz. Comparison of core network emulation platforms. In *MILITARY COMMUNICATIONS CONFERENCE, 2010 - MILCOM 2010*, pages 166–171, 2010.

[2] J. Ahrenholz, C. Danilov, T. Henderson, and J. Kim. Core: A real-time network emulator. In *Military Communications Conference, 2008. MILCOM 2008. IEEE*, pages 1–7, 2008.

[3] J. Ahrenholz, T. Goff, and B. Adamson. Integration of the core and emane network emulators. In *MILITARY COMMUNICATIONS CONFERENCE, 2011 - MILCOM 2011*, pages 1870–1875, 2011.

[4] H. Artail and S. Saab. A distributed system for consuming web services and caching their responses in manets. *IEEE Transactions on Services Computing*, 2(1):17–33, Jan 2009.

[5] P. Choudhury, A. Sarkar, and N. C. Debnath. Deployment of service oriented architecture in manet: A research roadmap. In *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, pages 666–670, July 2011.

[6] P. Novotny. *Fault localization in service-based systems hosted in mobile ad hoc networks*. PhD thesis, Imperial College London, 2014.

[7] P. Novotny, B. J. Ko, and A. L. Wolf. Delay tolerant harvesting of monitoring data for manet-hosted service-based systems. In *Services Computing (SCC), 2015 IEEE International Conference on*, pages 9–16, June 2015.

[8] P. Novotny, B. J. Ko, and A. L. Wolf. On-demand discovery of software service dependencies in MANETs. *IEEE Transactions on Network and Service Management*, 12(2):278–292, June 2015.

[9] P. Novotny and A. L. Wolf. Simulating services-based systems hosted in networks with dynamic topology. Technical Report DTR-2016-2, Department of Computing, Imperial College London, Jan. 2016.

[10] P. Novotny, A. L. Wolf, and B. J. Ko. Fault localization in MANET-hosted service-based systems. In *31st IEEE International Symposium on Reliable Distributed Systems*, pages 243–248, Oct. 2012.

[11] P. Novotny, A. L. Wolf, and B. J. Ko. Discovering service dependencies in mobile ad hoc networks. In *IFIP/IEEE International Symposium on Integrated Network Management*, pages 527–533, May 2013.

[12] W. She, I.-L. Yen, and B. Thuraisingham. WS-Sim: A web service simulation toolset with realistic data support. *Computer Software and Applications Conference Workshops*, 0:109–114, 2010.

[13] S. Tati, P. Novotny, B. J. Ko, A. L. Wolf, A. Swami, and T. La Porta. Diagnosing degradation of services in hybrid wireless tactical networks. In *SPIE Defense, Security, and Sensing*, May 2013.