Simulation Modelling Practice and Theory 32 (2013) 42–63

Contents lists available at SciVerse ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

Simulation of service-oriented systems for Mobile Ad hoc Networks

Andry B. Cruz [*], Tuleen Boutaleb, Huaglory Tianfield

Department of Computer, Communications and Interactive Systems, School of Engineering and Built Environment, Glasgow Caledonian University, Glasgow, UK

ARTICLE INFO

Article history:
Received 21 August 2012
Received in revised form 6 November 2012
Accepted 25 November 2012

Keywords:
Simulation
Service-Oriented Architecture (SOA)
Mobile Ad hoc Networks (MANET)
OSGi
ns-2

ABSTRACT

The simulation of distributed systems that implement a Service-Oriented Architecture (SOA) imposes a set of requirements on their simulation models. These models must replicate the interactions that characterise the SOA operational paradigm and the role of the entities that support an SOA. When those interactions occur over Mobile Ad hoc Networks (MANETs), simulation models must accurately reproduce the effects that mobility and other networking factors have on the system. This paper presents the architecture of a simulation environment for MANET-interconnected Service-Oriented Systems. The simulation environment combines the realistic SOA support provided by OSGi with the network simulation capabilities of ns-2. The application-layer elements of the simulation models built for this environment are also SOA implementations, thus promoting structural model validity. This approach provides the opportunity of utilising elements or prototypes from the modelled systems as part of the simulation model, promoting its continuity. In addition, it takes advantage of the benefits that SOA has brought to field of Modelling and Simulation.

1. Introduction

Distributed systems interconnected by Mobile Ad hoc Networks or MANETs include mobile computing systems, mobile cloud applications, ubiquitous and pervasive systems, etc. In these systems, referred to as Mobile Ad hoc Networked Systems (MANS) throughout this paper, interactions between their components are triggered by the need to exchange resources and capabilities. This exchange, realised as sequences of messages transmitted over the network, helps attain collective, node-specific and component-specific goals. As resource exchange depends on the effectiveness of its supporting messaging process, the possibility of fulfilling the goals of MANS is heavily influenced by the intrinsic characteristics of the underlying network. These characteristics determine the degree of visibility between nodes, between the applications hosted by those nodes and consequently, the structure of the distributed system that results from the exchange of resources between those applications.

Amongst the most important characteristics of MANET [1], decentralisation and changing topology can be identified as challenges to the exchange of resources or capabilities. In order to cope with these challenges, the abstraction of shared resources and capabilities as *services* and the adoption of a Service-Oriented Architecture (SOA) [2] for MANS have been explored as feasible solutions [3–8]. Additionally, SOA can be considered a building block in the development of mobile cloud applications that may interoperate over a MANET [9–11].

Similar to MANS conceived under other architectural paradigms, Service-Oriented MANS have been routinely studied and evaluated through simulation. As illustrated in Fig. 1, the model of a MANS utilised in a simulation must be able to combine

* Corresponding author.
E-mail addresses: andry.cruzdiaz@gcu.ac.uk (A.B. Cruz), t.boutaleb@gcu.ac.uk (T. Boutaleb), h.tianfield@gcu.ac.uk (H. Tianfield).

1569-190X/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.simpat.2012.11.010

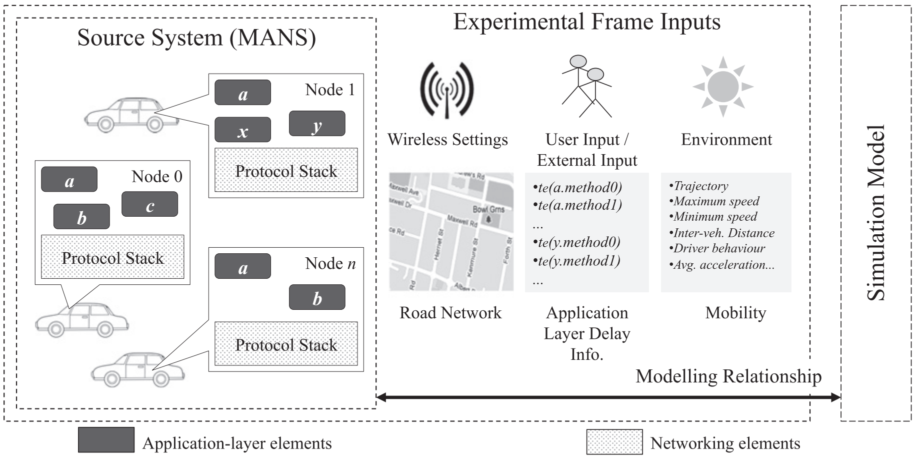

Fig. 1. Source system and experimental frame inputs of a MANS illustrating the difference between application, networking elements and the experimental input.

application-layer and networking features of a source system. It must also be able to replicate the effects of the inputs in an experimental frame on the modelled features. In particular, on the dynamic structure of the resulting system.

The features of a source system considered as part of the simulation model of a MANS are normally studied under different experimental conditions. Therefore, it is desirable that those elements exhibit loose coupling with the elements that generate the effects of the experimental input. It is also appropriate that the dynamic structure of MANS emerge at simulation time from the interactions between modelled features and experimental conditions. Thus exempting modellers from having to include the expected structural changes within the model. In these respects, the SOA design paradigm has been regarded as an appropriate approach to the design of dynamically reconfigurable models [12]. Moreover, the loose coupling promoted by SOA furnishes model elements with a high degree of re-usability.

In the specific case of Service-Oriented MANS, simulation models that are also SOA implementations offer advantages versus alternative approaches. These models can help eliminate validity and continuity problems between their elements and the elements of the modelled systems. The modularity promoted by SOA permits the composition of models from fine grained elements. These elements may have a one-to-one correspondence with the elements of a source system, thus contributing to structural validity. That is, how well the model mimics state transitions in the source system in a component-by-component fashion. Continuity is encouraged when technological constraints allow elements from a source system to be plugged into its simulation model or vice versa. As modifications to a shared element are immediately reflected in the model and source system, model validity is continuously retained.

A number of frameworks for the simulation of service-oriented systems (referred to as *SOSims*, for short) proposed to date could be utilised for the execution of models that represent Service-Oriented MANS. However, in order to simulate MANS, *one or more* of the following issues would have to be tackled depending on the selected simulator:

1. The lack of out-of-the-box, realistic networking simulation support to the level required by MANET.
2. The construction of models that are tightly coupled to the simulation environment or to the implementation of a simulation modelling formalism.
3. The burden of incorporating position awareness support within application-layer elements of the model. Location awareness is the most basic form of context awareness that can be expected to exist in most MANS.
4. The impossibility of utilising actual applications (or their prototypes) as part of the simulation model. This constraint precludes the selected simulation environment from acting as a debugging aid for Service-Oriented MANS.
5. The infeasibility of modelling service-oriented systems that have a dynamic structure.

As an alternative to using existing SOSims, network simulators such as JiST/SWANS [13], ns-2 [14] or OMNeT++ [15] could be employed. These tools provide accurate means for the simulation of MANETs. Nevertheless, the simulation scenarios would be required to incorporate a number of application-layer extensions so as to reproduce the features of a Service-Oriented MANS. These extensions would be tightly coupled to the simulation tool and could be prone to promote poor model continuity and validity.

The novel SOSim presented in this paper strives to solve the described issues that arise from simulating Service-Oriented MANS using existing simulators. At the same time, the presented SOSim takes advantage of the enhanced modularity provided by SOA, both as modelling approach and as the architecture of the simulation environment. Models that realise an SOA feature reusable elements and permit scalable simulations. In the same manner, the service-oriented architecture of the simulation environment facilitates its extensibility and customisation. SOA also promotes loose coupling between simulation environment and simulation models.

The proposed SOSim relies on the OSGi[1] [16] specification as a key enabler of its flexibility. It takes advantage of the MANET simulation and scripting capabilities of ns-2 and provides coherent timing control and delay modelling across all elements in the model. When the simulated MANS are also based on OSGi, a technology that has been widely embraced in the development of embedded and in-vehicle telematics systems; the SOSim permits consumer and provider applications or prototypes from a source system to be plugged into the simulation model. This capability facilitates the realistic evaluation, debugging and refinement of those components as long as they adhere to a minimum set of constraints required by the simulation environment.

The rest of this paper is organised as follows: Section 2 presents a review of the previous works related to the topic of service orientation in the field of simulation, Section 3 provides a brief background on SOA and OSGi. Section 4 presents the two-layered architecture of the SOSim, whereas Sections 5 and 6 describe the composition and instantiation of a simulation model, respectively. Section 7 offers details on the approximate time morphism that characterises a simulation, as well as the components that control the execution of a model. Section 8 presents an application case study that shows how network and application layer delay play a combined role on the gauged performance of a source system. A discussion, conclusion and future works have been included in Sections 9 and 10, accordingly.

2. Literature review

The role of SOA in simulation design and implementation has been studied from two different angles:

- As an architectural approach in the conception of simulation tools, models and frameworks. Normally with the purpose of enabling model element re-usability and distributed, scalable simulations.
- As the architecture of the source systems to be modelled and simulated, i.e., whether a framework provides default support for the simulation of models that represent source systems with a service-oriented application layer. Such support is not related to the actual architecture of the model or simulation tools.

It is therefore important to make a clear distinction between: (a) frameworks that are SOA implementations or use services to support the simulation of models, and (b) SOSims built to simulate models of SOA source systems.

In the context of distributed simulations, implementations of the High Level Architecture (HLA) [17] standard are the norm. The integration of HLA realisations with service-oriented systems, mostly based on web services, has been identified as a means of increasing model re-usability and loose coupling of HLA federates [18]. The resulting combination also enables integration between simulation models, production/source systems and systems of systems [19–22]. Despite these solutions allow the simulation of HLA federates implemented as services, they do not offer default support for the simulation of service-oriented systems. That is, their relationship with SOA is constrained to Modelling and Simulation (M&S) aspects. Non-HLA frameworks like those put forward in [23–25] have also taken advantage of SOA from an M&S perspective.

SOSims, or the frameworks that provide default support for the simulation of service-oriented systems, incorporate off-the-shelf functionalities required in an SOA implementation. These include service brokers or registries, service registration, advertisement and discovery mechanisms, etc. SOSims, however, do not necessarily take advantage of the benefits that SOA has brought to M&S. Despite the simulation models they support represent service-oriented systems, these models or the SOSims themselves may not be service-oriented.

The SOA-DEVS Modelling Framework (SOAD), first presented in [26], is one of the most relevant endeavours in SOA simulation. SOAD is an implementation of the Discrete Event System Specification (DEVS) formalism based on DEVSJAVA, a Java implementation of DEVS. It also relies on DEVS-Distributed Object Computing (DEVS-DOC) [27] for the separation of software and hardware concerns at the model level. The hardware model included in SOAD provides simple network simulation, but lacks default support for realistic MANET simulation. Furthermore, brokers in SOAD do not operate as services themselves which could constitute an obstacle to simulating service-based systems that function in a peer-to-peer fashion. As a consequence of the formalism-based nature of SOAD, the possibility of including elements from source systems as part of a model are limited.

As a way of incorporating fine-grained hardware modelling capabilities in SOAD, the simulator for service-based system co-design outlined in [28] includes additional abstractions for node and networking elements. The former permits accurate processing load reproduction, whereas the latter are more suitable for networks with static topologies. The extension of SOAD presented in [29] facilitates dynamic service composition at simulation time, permitting the simulation of systems with a dynamic structure. Nevertheless, these changes must be scripted within the model; a requirement that prevents the emergence of system structures from the interaction between modelled features and experimental input.

The framework described in [30] aims to co-simulate realistic communications support in a service-enabled system by exchanging messages between consumers and providers over OMNeT++. The processes that drive the transmission of messages are statically modelled in the Process Chain Language (ProC/B) as timed sequences of service calls and inputs. Similar to SOAD models, the models supported by this framework assume that the structure of the system is known a priori.

[1] Formerly known as Open Services Gateway Initiative.

Table 1
Comparison between different simulation environments.

Primary study	Meant to simulate SOA systems	Environment is based on SOA	Default MANET sim. support (1)	Loc. awareness support in app. layer (3)	Elems. from source system in model (4)	Dynamic structur support (5)
[26]	√					
[28]	√					
[29]	√					√
[30]	√		√			
[31]	√		√			√
[32]		√			√	√
[33]			√			
[35]					√	
Own	√	√	√	√	√	√

Although OMNeT++ supports the simulation of MANETs, the application-layer elements of the framework do not include facilities like service registries or position update mechanisms.

OMNeT++ is also used to provide realistic networking and mobility control to the SOAMANET environment [31]. SOAMANET relies on Model-Integrated Computing (MIC) for the description of the service-based systems and work flows within a model. MIC requires data and work flow models to be explicitly defined. The resulting models present a degree of coupling to their simulation environment that is higher than the characteristic coupling in formalism-based models.

The multi-purpose framework presented in [32] can automatically produce a simulation model and production test framework from a variety of requirement specification formats. This process, known as the DEVS Unified Process (DUNIP), is a thorough approach conceived to tackle the continuity problems that could affect formalism-based modelling previously mentioned in this paper. A DUNIP-generated model can utilise web services from the test/prototype framework which helps conserve validity as the prototype is refined. Nonetheless, incorporating realistic MANET simulation into a DUNIP model could require (a) the integration with frameworks such as [29,33] and (b) functionalities like service discovery to be modelled, so they can occur on a peer-to-peer fashion over a network simulator.

A comprehensive survey on service-oriented simulators, simulators for service-oriented systems and their taxonomy, can be found in [34].

The challenge of integrating realistic network simulations with application layer models has already received attention from the research community. Simulation environments such as [35,33] extend DEVS implementations with a set of models and capabilities that permit inter-application message exchange over network simulators. They also synchronise DEVS timing with the simulation schedulers used for communication purposes. However, those simulation environments do not offer support for SOA operation. Table 1 presents a comparison between those simulators and the surveyed SOSims around some of the issues already identified in this paper.

The simulation environment presented in this paper features modelling capabilities comparable to those offered by [26,30,31,33] and the SOA support of OSGi. It simultaneously provides support for the simulation of Service-Oriented MANS and exploits the advantages of SOA from the perspective of M&S. Its reliance on the real-world capabilities of OSGi narrows the gap between source systems and their simulation models.

3. Background

Service orientation is a technology-agnostic software design paradigm based on a set of principles that aim to produce reusable, scalable, interoperable and modular software [36]. The principles that characterise SOA design seek to constraint component coupling to the interface or contract level and message format. The resulting loose coupling enables service-oriented systems to operate in a *find-bind-execute* mode, where consumers have the possibility of locating, *binding* and eliciting resources from providers at runtime. This paradigm can be contrasted against approaches where consumers are coupled to specific provider instances at design time.

Despite the notion of SOA is usually linked to web services and distributed systems, its design principles have been adopted within stand-alone setups so as to maximise component re-usability and reduce system complexity. In the case of OSGi, a framework originally conceived as a single-node service gateway, SOA provides the levels of modularity and dynamic structure required in embedded environments.

An OSGi framework instance has a local registry that consumers may query in order to *find* suitable providers. Registry querying is done syntactically based on a service contract. Optionally, queries can include LDAP filters on the properties that suitable providers are expected to expose. Both querying and service registration can be attained using Application Programming Interfaces (APIs) or based on metadata files, such as OSGi Declarative Services (DSs). Consumers that declare queries using metadata receive updates from the registry as suitable provider instances become available. When metadata files are used, these are included as part of the *bundles* or packaging units that contain the elements of an application.

The OSGi Release 4 version 4.2 Service Compendium [16] introduced the Remote Services (RSs) specification. RS enables service sharing between different framework instances or between framework instances and non-OSGi service-oriented sys-

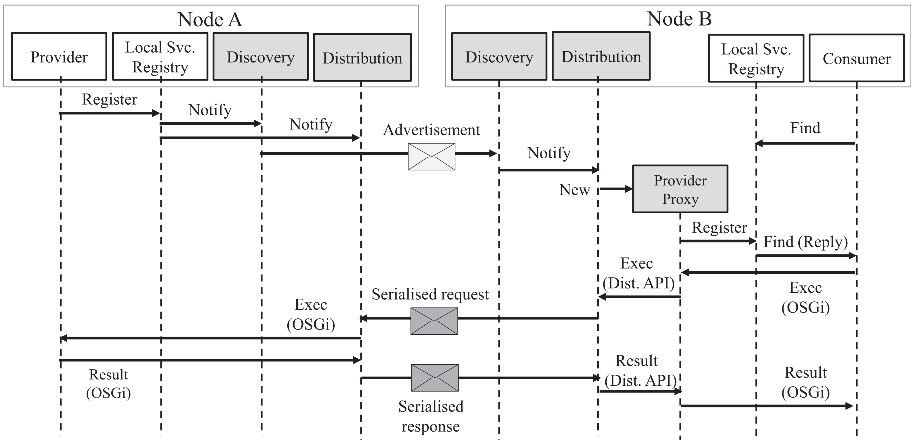

Fig. 2. Sequence diagram illustrating service discovery and invocation operations between two OSGi peers.

tems. Since then, RS implementations like the Eclipse Communication Framework (ECF) [37] have permitted distributed, service-oriented systems totally or partially based on OSGi to be formed.

The RS specification describes the contracts and basic operation of a set of components that are in charge of *service discovery*. These components are also responsible for service message formatting or *service distribution* in a way that is transparent to the consumers or providers within a framework instance. As illustrated in Fig. 2, the discovery components in a node find the local services that have been *exported*, i.e., whose properties indicate that they are available to consumers hosted within any framework instance. If those discovery components operate on a proactive manner, an advertisement is generated and transmitted. However, the discovery strategies, frequency and content of advertisement messages are not part of the OSGi specification and depend entirely on the implementation of the discovery components. Implementors can choose between customised message formats or standard formats such as those defined by the Service Location Protocol (SLP) [38], Domain Name System-Service Discovery (DNS-SD) [39], amongst others. These messages can be sent proactively, reactively, on a multi-hop manner, towards a centralised registry, etc., according to the needs of the system. Discovery implementations are also in charge of signalling other components in a framework instance when a remote provider is no longer available.

When an advertisement is received by a discovery counterpart, e.g.: a node listening on port 427 for SLP-formatted User Datagram Protocol (UDP) messages, the recipient notifies a corresponding in-node distribution container. The latter *imports* each one of the services described in the advertisement by allocating a proxy object implemented within the container. The instantiated RS proxy is then registered as a provider of the discovered service with the local registry, allowing consumer applications within the node to bind to the proxy instance. Aside from certain (*name*, *value*) properties that imported services must exhibit in compliance with the RS specification, proxies registered as service providers are indistinguishable from concrete providers within a node.

Service call requests issued by consumers to local proxies are forwarded to the distribution container, which formats and transmits service messages. A compatible distribution container in the provider side receives those messages and translates them into normal OSGi method calls, which are forwarded to the in-node, concrete provider instance indicated in the message. Following a similar process, request responses are returned to the original callers. The format of the request/response messages depends exclusively on the implementation of a distribution container. Message layouts can be proprietary, remote procedure calls, Simple Object Access Protocol (SOAP) compliant, etc., depending on interoperability requirements.

In the context of SOA model simulations, the proxy-based operational approach of RS and the loose coupling between application elements in the model facilitate the re-direction of inter-element communications into ns-2. Message re-direction is achieved by furnishing the environment with simulatable discovery and distribution components. From the point of view of the simulation environment architecture and simulation execution, RS afford the possibility of carrying out distributed simulations.

4. Architecture of the simulation environment

The SOSim environment presented in this paper integrates a network simulation layer based on ns-2 with a set of OSGi simulation services. The ns-2 network simulator provides realistic data communications, mobility and scripting support as well as simulation-wide event scheduling capabilities. These capabilities are complemented by the OSGi simulation services, in charge of instantiating and executing the elements in the model that correspond to the application layer. OSGi simulation services furnish the instantiated model with the ability to operate in simulation time, maintaining coherence between application and data communication event timing.

The following subsections offer more detail on the elements that constitute the layers of the SOSim.

4.1. Application simulator architecture

The application simulator consists of a set of components that instantiate the system model, modify its behaviour according to the parameters that define the experimental frame, and mediate between the interactions of the elements in the model. The *support bundles* that make up the application simulator provide:

- A *Discrete Event Simulation Portal* (DES Portal) service that handles all Inter-Process Communication (IPC) with the network simulator.
- A *Network Adapter* service that decouples the system from the standard networking classes. This allows the detouring of messages towards the DES Portal.
- A *Timer Factory* service used in the instantiation of timers, which are useful for functionality scheduling and operate in simulation time.
- A *Simulation Proxy Factory*, used to control timing in the application layer by mediating between component interactions. Simulation proxies also permit application-layer components to be controlled from ns-2 script files.
- *RS implementations* of the OSGi discovery and distribution mechanisms which utilise the network adapter service.
- A simple *Positioning Service*, easing the process of evaluating applications that require mobility and position awareness. This is achieved in accordance with the model's mobility pattern, normally handled by the network simulator. This basic service relieves modellers from dealing with position elicitation from simulator-specific structures.
- An *Instantiator* component that dynamically builds the simulation model at start-up, according to a provided configuration.

Simulation support bundles operate alongside a set of user-defined bundles that model the application layer of a Service-Oriented MANS. Fig. 3 illustrates these user-defined or *source system bundles* and the simulation support bundles within a model before components are instantiated.

Different instances of certain source system and support components are expected to exist across several nodes in a simulation model. For instance, all nodes in the model must have their own instance of the network adapter service. In the same

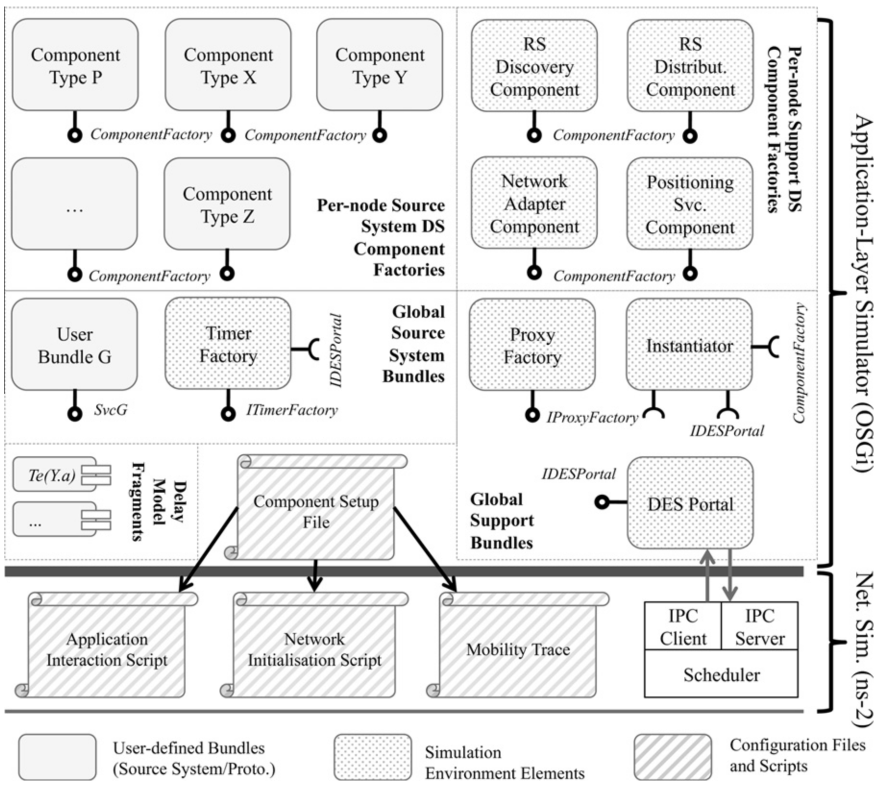

Fig. 3. Ball and socket diagram of the simulation environment at *setup* time depicting the DS *ComponentFactory* services corresponding to per-node components. Framework Runtime Bundles are not shown.

way, some or all nodes could provide the same user-defined service. In order to replicate these scenarios without having to create several instances of the OSGi framework, the simulation services provide *framework multiplexing*.

Depending on the stage of the multiplexing process, the internal structure of the environment's application layer presents three clearly differentiated phases: (a) a *simulation setup* phase, shown in Fig. 3, which includes the simulation support and source system bundles as they are loaded by the framework before multiplexing, (b) an *instantiated system* phase, where the components defined by the loaded bundles have been instantiated or multiplexed according to a description of the model to be simulated, and (c) a *simulation* phase, where the instances that make up the model are executed. These instances interact with each other according to the modelled experimental frame, the network characteristics described by the model and the behaviour modelled within each component.

4.2. Network simulator architecture

The network simulator has the role of enabling realistic communications. It processes the mobility traces for a specific scenario and acts as a scheduler for application-layer events. Therefore, this layer of the co-simulator interacts with the application layer simulation in two different ways: making possible data transmission and controlling the timing of application-layer events in combination with a model's simulation proxies. The operation of these proxies is described later in this paper.

For the purposes of IPC handling, ns-2 version 2.34 has been enhanced with two classes (IPC server and client) which manage intercommunication over UDP with the OSGi DES Portal component. Single instances of those two classes are created when the network simulator is launched.

The IPC client instance is in charge of dispatching scheduled application layer events and data to the DES Portal from the main ns-2 execution thread. In contrast, the IPC server runs in a separate thread and listens for ns-2 commands or messages issued by the application simulator.

The ns-2 simulator has been originally conceived as a single-threaded application. As a consequence, ns-2 is not intrinsically thread-safe. However, the operation of the IPC server thread guarantees mutual exclusion between the evaluation of received commands and the simulator's event dispatcher. The incoming commands received by the IPC server thread are issued by the application layer before the co-simulation starts. They may also be generated during a simulation at times when the ns-2 event dispatcher thread is blocking, waiting for the application simulator to handle an event or message.

Alongside the events that result from model execution, the network simulation triggers the following three additional events:

- *Simulation start-up notification:* Serves to inform the DES Portal of the availability of the network simulator. This notification is triggered once a warm-up period indicated in the model has elapsed.
- *Clock tick events:* occur every millisecond (simulation time). This event is dispatched to the DES Portal with the objective of maintaining the simulation time at the OSGi level, mainly for debugging and logging purposes.
- *Position update events:* Collect the positions and speeds of the nodes in ns-2 and pass them to the DES Portal. These are forwarded to the Positioning Service Provider, which notifies those applications in the model that have subscribed to receive location updates.

The utilisation of the popular ns-2 for the simulation of network elements in the model furnishes the environment with default MANET simulation support. Amongst existing environments, only SOAMANET [31], DEVS/ns-2 [33] and the proposal of Bause et al. [30] share this characteristic. Nevertheless, SOAMANET does not take advantage of SOA from the M&S perspective and its models are tightly coupled to the framework, as discussed in Section 2. Similarly, [30] can simulate a MANET, but does not offer default support for dynamic service discovery. DEVS/ns-2 could be used to simulate MANS, yet this framework would require additional modelling efforts as it lacks the means of simulating service-oriented systems.

5. Model composition

Models created for the SOSim feature decoupled elements of three different kinds: (a) those that model the application layer of a source system, executed by the application simulator, (b) networking elements, responsible for modelling the underlying MANET, and c) elements that model an experimental frame, which may be executed by the application simulator or the network simulator according to their implementation.

5.1. Application-layer elements

The application-layer aspects of a simulation model are defined by a set of model bundles. These include source system bundles and some of the bundles provided by the SOSim. The internal structure of the application layer within each simulated is made up from instances of the components declared by model bundles. This structure is defined in a *component setup file*. Amongst other parameters that will be described later in this paper, this file dictates which nodes should host which component instances at simulation time.

Application-layer elements of the model may also include OSGi *fragments*, packaging units whose components extend those defined in a specific bundle. These are used by the SOSim to compute the application execution delay associated to a service call.

5.1.1. Model bundles

Model bundles define software components whose instances are expected to be hosted by the simulated nodes. Model bundles are comprised of source system bundles. They also include the support bundles that provide positioning services, RS implementations and the network adapter services.

A set of model bundles in a simulation model may define application-layer components of which an instance has to be created for more than a node in the model. For instance, each node should have its own instance of the component that provides the network adapter service. The components created for those *per-node bundles* integrate applications of one or more of the following types:

- Stateful applications, where identical components hosted by several simulated nodes can maintain state data that might differ from node to node.
- Applications that directly handle the inputs generated by an experimental frame. These inputs normally correspond to application instances hosted by specific nodes.
- Applications with *delay annotations*, where the completion time of at least one of their methods is considered to impact upon the outcome of a simulation.

In order to facilitate the creation of multiple instances of the components defined by per-node bundles, modellers implementing these components are constrained to utilising OSGi Declarative Services. They must also define those components as *DS component factories*. DS components encompass standardised, readable metadata. These metadata include application properties as name-value pairs, as well as information about the services provided and consumed by the instances of a component. These data are used at model instantiation time towards the support of node multiplexing. Declaring DS components as factories prevents their instantaneous allocation when their containing bundle is loaded or *activated*. Instead, DS factories offer a *ComponentFactory* service that can be consumed by an instantiator, which may create multiple instances of the same component.

Although per-node bundles provide service metadata and the means of creating multiple instances of the components they define, they do not contain information about the structure of the simulation models in which they can be integrated. In Fig. 3, for example, an instance of a component *P* might be required to exist in all nodes of the model, but that information is not part of the bundle that defines *P*. Instead, a *component setup file* includes information regarding the structure of the model to be instantiated from the bundles loaded by the framework.

Model bundles may also define global components which usually provide stateless utility services within a node. The methods of these services are assumed to execute in negligible time and they do not receive direct input from the elements that model the experimental frame. As the applications defined by those *global bundles* do not store state data, are not the direct target of network messages and don't compose any other services, a single instance within the model is enough to simulate a utility service that is locally available to all modelled nodes. In contrast with per-node bundles, global bundles are not constrained to using DS or any other form or OSGi services implementation, but the services they provide must be registered at bundle activation time. The classification of support and source system bundles according to the number of instances created from their components is shown in Fig. 3.

The fact that the applications in a model are defined by OSGi bundles permits bundles from an OSGi-based MANS to be plugged into a model. This can be contrasted with formalism-based approaches like SOAD and its derivations [26,28,29], where application elements have to be re-written as DEVS models. Although options such as [32,35] have the same plug-in capabilities, those frameworks do not support the simulation of a MANET.

5.1.2. The component setup file

The component setup file is an XML document whose structure must comply with a specific Document Type Definition or DTD. An excerpt of this type definition is shown in DTD 1. The parameter values contained in the component setup file are divided in three main sections according to the aspect they cover. These sections are:

- A *general* set of values used by the environment at setup time. These include the path to the network simulator executable and the locator of the *network initialisation script* in charge of allocating the nodes within ns-2.
- *Scenario* parameters, including values such as number of nodes in the simulation, warm-up period duration or start time, simulation end time, the paths to the *mobility trace* and *application interaction* scripts which represent inputs from the experimental frame.
- The *node* configuration parameters, indicating which component instances are hosted by which nodes and their properties.

Under the node configuration, the *factory* elements declare which instances of the DS component factory service will be used to instantiate the components hosted by each node. As a design choice, all of the single-token, single-occurrence, unstructured data properties of factory elements and of their children elements are represented as XML attributes.

DTD 1 Excerpt of the DTD for the component setup file.

```
<!ELEMENT configuration (general, scenario, node)>
<!ELEMENT general (simulator_exec, simulator_init)>
<!ELEMENT scenario (number_nodes, start_time, end_time, mobility_trace, input _script)>
<!ELEMENT node (factories, global?)> <!ELEMENT factories (factory+)>
<!ELEMENT factory (delay?, properties?, nodenums?)>
  <!ATTLIST factory name NMTOKEN #REQUIRED>
  <!ATTLIST factory order NMTOKEN #REQUIRED>
  <!ATTLIST factory type (infra | scriptable)>
  <!ATTLIST factory policy (all | specific | random)>
  <!ATTLIST factory prob #IMPLIED>
  <!ELEMENT delay (method+)>
    <!ELEMENT method EMPTY>
    <!ATTLIST method name NMTOKEN #REQUIRED>
    <!ATTLIST method meanms NMTOKEN #IMPLIED>
    <!ATTLIST method noisems NMTOKEN #IMPLIED>
    <!ATTLIST method delaymodel NMTOKEN #IMPLIED>
<!ELEMENT properties (prop+)>
  <!ELEMENT prop EMPTY>
    <!ATTLIST prop name NMTOKEN #REQUIRED>
    <!ATTLIST prop value NMTOKEN #REQUIRED>
  <!ELEMENT nodenums (nodeid+)>
    <!ELEMENT nodeid (#PCDATA)>
<!ELEMENT global (service+)>
  <!ELEMENT service (#PCDATA) >
```

Factory configurations must include four attributes, specifically, name, order, type and policy. The *name* is a string that must match a DS component name declared by the model bundles. The *order* mimics the effect of the OSGi start level within each node, indicating the sequence in which component instances must be created. The *type* defines if the component to be generated from the factory must have a scripting object counterpart within ns-2. Besides, it might indicate if its methods have an associated delay. In both cases, the value of *type* must be set to *scriptable*, indicating that the application-layer component instance can be controlled from an ns-2 script. The *policy* attribute defines which nodes will be fitted with an instance of the component produced by the factory. The supported instantiation policies are:

- For *all* nodes: a component instance is allocated for each node in the model.
- *Specific* nodes: instances are allocated exclusively for the nodes whose numbers are included in a given list under the `factory/nodenums` tag.
- *Random* nodes: components are instantiated for a uniformly random fraction of the participating nodes. This fraction is determined by a probability parameter (`prob`) included in the factory definition entry.

Furthermore, factory configurations may also include a list of property name-value pairs (under `properties`). This list of properties is assigned to each component allocated from the factory at instantiation time.

Parameters that determine method execution delay are also associated with the factory used to instantiate a given component. If the type of the factory has been declared as *scriptable*, it is possible to add a delay child element to the factory configuration entry. This element encloses a list of `method` entries. Each method entry contains name attribute value alongside one or more of the following attributes:

- A `delaymodel` attribute, containing the fully qualified name of a class that implements a delay model function or T_e. This function is utilised to determine the execution time t_e of its associated method.
- A `meanms` (t_μ) representing the average delay for completing the execution of the method, in milliseconds.
- A `noisems` (t_w) which denotes the maximum deviation time in the method's delay.

If an implementation of a delay model function is not provided for a method, the SOSim utilises the uniform distribution to compute the method's local delay according to the formula:

$$t_e = max[U(t_\mu - t_w, t_\mu + t_w), 0] \tag{1}$$

Methods that are considered to execute with negligible delay do not need to have a corresponding `method` element in the component setup file.

5.1.3. Delay model fragments

Fragments are OSGi bundle-like packages whose main purpose is to provide classes or resources to specific bundles within the framework, effectively "extending" those target bundles.

Delay model fragments belong exclusively to the simulation model. That is, unlike source system bundles, delay model fragments are not interchangeable with any element of the modelled system. These fragments extend the proxy factory support bundle. They expose classes whose fully qualified names match the value of a *delaymodel* attribute included in the component setup file. Each class implements a public *getDelay* function, used to compute the simulation time that a method must take for it completion. The characteristics of this function and the impact of the computed delay on the simulation process are explained later in this paper.

5.2. Networking-layer elements

A simulation model must include a network initialisation script written for ns-2 in the Tool Command Language (Tcl). Its path is specified as part of the component setup file.

The network initialisation script is in charge of instantiating the wireless communication channel(s) to be utilised, defining the type of PHY and data link layers of the nodes, the characteristics of the antennas to be simulated, the routing protocol, the queuing mechanism of the wireless interfaces and a signal propagation model. The network initialisation script must also allocate the ns-2 wireless node instances using the `scenario/number_nodes` parameter provided at model instantiation time.

The instructions in a network initialisation script are similar to those utilised for the setup of MANET simulations in ns-2. But unlike ns-2 simulation scenarios, modellers are not required to declare transport-layer agents, application elements or application behaviour within the Tcl script. Agents within ns-2 are allocated in response to commands received via IPC. These commands are generated by the application simulator according to the needs of each application, e.g.: when an RS distribution component allocates a UDP socket through the network adapter service, the latter request a corresponding UDP agent to be created within ns-2. Application behaviour is modelled by the application-layer elements declared in the model bundles.

The network initialisation script must also configure the generation of simulation traces that are going to be produced by ns-2.

5.3. Experimental frame elements

Node mobility is one of the most important input parameters that describe the experimental frame in which MANS are evaluated. Mobility support in the SOSim is provided by ns-2, in which the movement of nodes is usually defined by a movement trace file. Therefore, the configuration of a simulation model must include the path of such trace file in the *scenario* section of the component setup file. Valid traces can be produced with Carnegie Mellon's *setdest*, *vanetMobiSim* [40], etc.

In order to reduce the random bias introduced by mobility traces in the simulation results, MANS are routinely evaluated using several traces. In consequence, the simulation of MANS models where only the mobility trace changes resembles the simulation of a pure ns-2 model under different mobility conditions; bundle models remain the same and only a single parameter in the configuration setup file has to be changed.

The elements that model an experimental frame must also permit the simulation of external input on the modelled system. External input can be generated by a component declared within a source system bundle, where timed input event generation must be carefully implemented to happen in simulation time. As an alternative, external input can be modelled using Tcl scripts. The path to an input generation script is included in the `application_interaction` parameter of the component setup file. As it has been previously discussed, script-based input modelling requires the components that will handle the input events to be produced from *scriptable* factories. Allocation of components from scriptable factories include the creation of a proxy counterpart object within ns-2. Proxy objects in the networking simulation layer are stored in a standard Tcl map named `app_` that is indexed by the combination *ComponentName:n*, where *ComponentName* is the name of the main class declared as a component in the DS configuration and *n* is the node number that hosts the component instance.

In order to provide an example of script-based input modelling, it can be assumed that *Component Instance X* in node 0 (Fig. 4)) handles user input by means of an *acceptUserInput* method that takes a single integer number as its parameter. It is then possible to provide such input at a given simulation time by means of the application interaction script, which would include a line like the following:

```
$ns_ at 1.73 "$app_(CmpX:0) tellapp acceptUserInput 5".
```

where `$ns_` is the *Simulator* instance in ns-2, and `1.73` is the time at which the `tellapp` command will be dispatched to the network simulator proxy `$app_(CmpX:0)`. A `tellapp` command with m arguments will pass the argument list $arg_1 \cdots arg_{m-1}$ to the method specified by arg_0 in the target component.

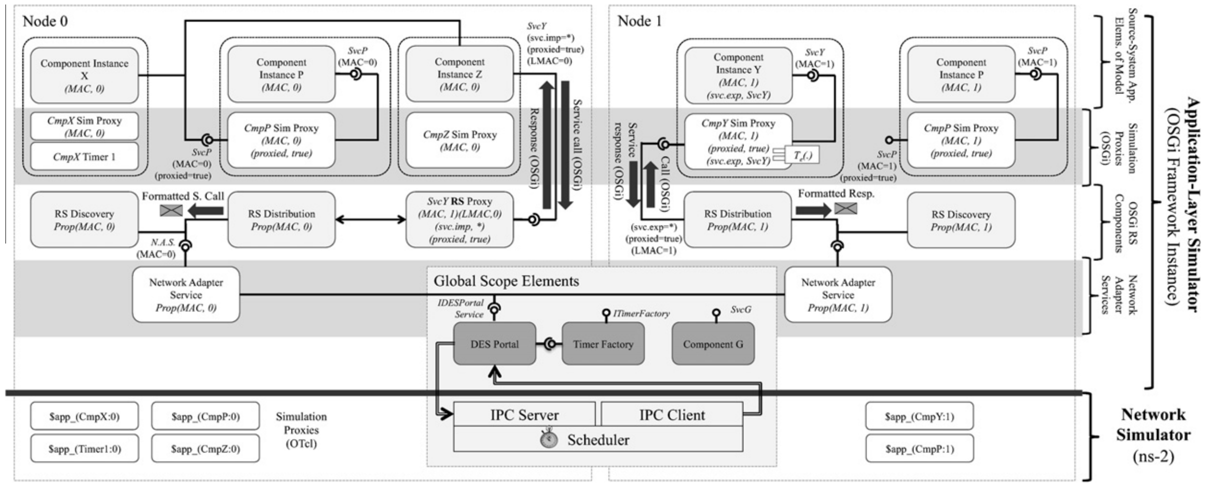
Node 0
Component Instance X (MAC, 0)
CmpX Sim Proxy (MAC, 0)
CmpX Timer 1
Component Instance P (MAC, 0)
SvcP (MAC=0)
CmpP Sim Proxy (MAC, 0) (proxied, true)
Component Instance Z (MAC, 0)
CmpZ Sim Proxy (MAC, 0)
SvcY (svc.imp=*) (proxied=true) (LMAC=0)
Response (OSGi)
Service call (OSGi)
SvcP (MAC=0) (proxied=true)
RS Discovery Prop(MAC, 0)
Formatted S. Call
RS Distribution Prop(MAC, 0)
SvcY RS Proxy (MAC, 1)(LMAC,0) (svc.imp, *) (proxied, true)
N.A.S. (MAC=0)
Network Adapter Service Prop(MAC, 0)
Global Scope Elements
IDESPortal Service
ITimerFactory
SvcG
DES Portal
Timer Factory
Component G
IPC Server
IPC Client
Scheduler
$app_(CmpX:0)
$app_(CmpP:0)
$app_(Timer1:0)
$app_(CmpZ:0)
Simulation Proxies (OTcl)
Node 1
Component Instance Y (MAC, 1) (svc.exp, SvcY)
SvcY (MAC=1)
CmpY Sim Proxy (MAC, 1) (proxied, true) (svc.exp, SvcY)
$T_c(.)$
Service response (OSGi)
Call (OSGi)
(svc.exp=*) (proxied=true) (LMAC=1)
RS Distribution Prop(MAC, 1)
Formatted Resp.
RS Discovery Prop(MAC, 1)
Component Instance P (MAC, 1)
SvcP (MAC=1)
CmpP Sim Proxy (MAC, 1) (proxied, true)
SvcP (MAC=1) (proxied=true)
Network Adapter Service Prop(MAC, 1)
$app_(CmpY:1)
$app_(CmpP:1)
Source-System App. Elems. of Model
Simulation Proxies (OSGi)
OSGi RS Components
Network Adapter Services
Application-Layer Simulator (OSGi Framework Instance)
Network Simulator (ns-2)

Fig. 4. Detailed diagram showing two nodes, their model elements and the corresponding proxies for their scriptable components after model instantiation. The ns-2 proxy object counterparts of the application proxies are also depicted.

Upon dispatching a scheduled input event at simulation time, the `tellapp` method handler in the network simulation proxy forwards the call to the IPC client object in ns-2, which passes the received control data to the DES Portal service. From there, and based on the name and node of the ns-2 sender proxy, the DES Portal delivers the command to the corresponding simulation proxy, in charge of invoking the target *acceptUserInput* method with the appropriate parameters at the appropriate simulation time.

Enabling input modelling via Tcl scripting furnishes the SOSim with a high degree of flexibility. This facilitates the coding of complex input generation algorithms at the scenario level, outside of the source system or its model elements.

6. Environment start-up and model instantiation

Once the source system bundles have been developed, scripts have been prepared and the component setup file has been created, it is necessary to start an OSGi framework instance with both support and source system bundles. After all bundles are loaded by the framework, configuration and script files are read and the elements of the model they describe are created in both layers of the simulator. The following subsections offer more detail on model instantiation and the packaging of the simulator.

6.1. SOSim start-up

Similar to other OSGi application, the SOSim can be deployed as a folder containing a set of bundles, alongside an OSGi *config.ini* file. The bundles required include source system bundles, delay model fragments and bundles that define external input generators, support bundles, and *framework runtime bundles* or third party bundles required for the operation of any of the previously mentioned bundles. By default, the framework runtime bundles must provide basic OSGi support, DS and Service Component Runtime (SCR) support, registry interaction contract definitions, as well as the definition of the contracts that must be implemented by the RS components, all of which are provided by existing OSGi implementations such as Equinox and Apache Felix.

The *config.ini* configuration file of an OSGi application lists the names of the bundles or fragments that have to be loaded in the framework. It also lists their *start-up* level, or the order in which they should be activated. As the SOSim services don't rely on any specific ordering, all bundles in this file can be set to *autostart* without indicating any start-up order.

After deploying the bundles and start-up information using the directory and config.ini approach or any other means of OSGi system deployment, the simulator can be launched using the start-up option `-Dconfig = P`, where P is the path to the component setup file.

6.2. Instantiation process

Bundles configured as *autostart* are activated as soon as they are loaded. Bundles that declare DS component factories immediately register their corresponding *ComponentFactory* service instances with the framework's registry. Similarly, components declared in global bundles register actual instances of the services they are meant to provide.

As soon as the instantiator support bundle is activated by the framework, the former proceeds to read the component setup file indicated by the `config` launch parameter. From that file, the number of nodes in the model, the list of global services declared, the list of per-node factories and the paths to the mobility trace, network initialisation script and network simulator are read. Those data are used to execute the steps outlined in Algorithm 1.

The instantiation process is crucial in attaining framework multiplexing. In a real system, visibility between per-node component instances depends on the network visibility of their host nodes. In a simulation model, all per-node component instances can be hosted by the same OSGi framework instance. Therefore, a series of steps are taken in order to adjust component visibility according to their disposition within a node and the inter-node visibility determined by the simulated MANET. These steps are:

1. Extension of all default properties of a component instance with the property name-value pair (*MAC*, *n*), where *n* is the node number for which the component has been created (line 36 of Algorithm 1).
2. Before creating components from a *ComponentFactory*, the service references of the components created by the factory are examined. When a service reference does not explicitly call for a remote provider or global service, the selection criteria are augmented with the filter (*MAC* = *n*). This indicates that the provider has to be an instance also hosted by node *n*, alongside the consumer application (line 17).
3. If a service reference points to a remote provider, the selection criteria are augmented with the filter (*LMAC* = *n*). The RS proxy for a remote service provided by a node *m* and discovered by node *n* will exhibit the properties (*MAC*,*m*)(*LMAC*,*n*). This extension ensures that the consumer of a service *S* in node *n* will be bound to the RS proxy that has resulted from node *n* discovering an instance of *S* hosted by node *m* (line 14).

Algorithm 1. Model instantiation from config. and component factories

```
Require: NumberNodes N, GlobalServices G, CompFactories F, FactoryConfigs C, MobilityTrace T, NetworkSimulatorPath
    NS, NetworkInitialisationScript I
1: Launch instance of the network simulator at NS.
2: Wait until all services in G have been registered.
3: Wait until all expected factories in C have been bound and added to F.
4: Execute script I in network simulator passing parameter N.
5: Load mobility trace T in network simulator.
6: Initialise service reference target storage S.
7: for all $f \in F$
8:    $dummyInstance \leftarrow f.newInstance()$
9:    $R \leftarrow SRC.getReferences(dummyInstance)$
10:    Initialise reference storage P for factory f.
11:    for all $r \in R$ where r does not target a service in G do
12:       if "service.imported=*" $\in r.target$ then
13:       {Reference calls for a REMOTE provider, i.e.: an RS proxy.}
14:       $r.target.augmentCriteria(LMAC = \mathcal{N})$
15:       else
16:       {Reference calls for an in-node (local) provider.}
17:       $r.target.augmentCriteria(MAC = \mathcal{N})$
18:       end if
19:       $P.add(r)$
20:    end for
21: $S.put(f, P)$
22: $dummyInstance.dispose()$
23: end for
24: for all $c \in C$ ordered by $C.order$ do
25:    $f \leftarrow F.get(c.factoryName)$
26:    for $n = 0\ N - 1$ do
27:       if instance from f should be created for node n then
28:       $P \leftarrow S.get(f)$
29:       {Get properties for comp. f:n indicated in factory config f}
30:       $X[f:n] \leftarrow f.properties()$
31:       for all $r \in P$ do
32:       {Replace value $\mathcal{N}$ added in 14 or 17 with node number n}
33:       $r.target \leftarrow$ replace$(r.target, \mathcal{N}, n)$
34:       $X[f:n].put(r.name+$".target"$, r.target)$
35:       end for
```

(*continued on next page*)

```
36:       $X[f:n].put(MAC, n)$
37:       $i \leftarrow f.newInstance(X[f:n])$
38:       Create corresponding proxies for $i$ according to config. $c$ in app and networking layers.
39:       **end if**
40:    **end for**
41: **end for**
42: Start simulation by sending *run* command to scheduler in ns-2.
```

Reference examination, as required in steps 2 and 3, is carried out using the OSGi Service Component Runtime (SCR).

A sample instantiated system is shown in Fig. 4. The diagram illustrates how per-node components have their properties and service references extended with *MAC* and *LMAC* criteria. The figure also illustrates how simulation proxies mediate between the interaction of actual source system component instances in the model. Inter-instance mediation is fundamental in the enforcement of application layer delay during simulation execution.

7. Simulation Execution

The execution of a simulation model is triggered by the instantiator when all expected global services have been registered and all per-node component instances have been allocated. This event is handled by the DES Portal, which reads the simulation warm-up value w from the component setup file and issues an "at" command to the network simulator, scheduling the generation of the *simulation start-up notification* event for time w. Subsequently, the DES Portal issues a *run* command to the ns-2 scheduler, starting the simulation process in the networking layer of the SOSim.

A warm-up period $w > 0$ s is desirable when the mobility trace in the model places the nodes in positions that are not a reflection of a realistic scenario. In such case, only mobility-related events are executed during the warm-up period. This allows nodes to reach stable mobility patterns. Once w seconds have elapsed, the simulation start-up notification is dispatched to the DES Portal signalling that the application-layer simulation must begin.

Upon receiving a simulation start-up notification, the DES Portal enables the services registered by the simulation proxies in the system, permitting that consumers bind to local providers. From that point onwards, the execution of the simulation are a sequence of one or more of the following application-layer events, which are triggered by the ns-2 scheduler: (a) *timed input handling*, such as the input included in an application interaction script, (b) *timer-controlled tasks*, where the timers have been obtained from the timer factory and operate in simulation time, (c) *remote binding operations*, executed by a consumer when it gets a reference to a remote service, which in turn occurs as the result of data communications, e.g., the reception of a service advertisement; and (d) *data communications*, when a listener associated to a network socket is invoked upon receiving data. The execution of event handlers may result in the scheduling of further events. This cycle might continue according to the behaviour of application-layer elements and until the simulation end time is reached.

The following sections offer additional detail on the approximate time morphism that results at execution time from application-layer delay modelling. In addition, it describes the operation and implementation of the key components that control the co-simulation process.

7.1. Application-layer delay modelling and its resulting approximate time morphism

As it has been previously discussed, the SOSim facilitates the partial construction of models using application-layer elements or prototypes from a source system. It also makes possible the simulation of delay linked to the execution of application-layer service calls.

To ensure that source system/prototype components remain decoupled from the simulation environment, method delay is provided by modular artefacts that belong exclusively to a simulation model.

Application delay is enforced during simulation execution by mediating the interactions between components using simulation proxies, as illustrated in Fig. 4. This delay enforcement approach results in an approximate time morphism during the simulation, which is influenced by the provided delay parameters and delay model functions.

In order to describe the manner in which delay parameters and model functions shape the resulting approximate time morphism, the execution of the service consumer Z shown in Fig. 4 can be used as an example. For this purpose, it is possible to assume that Z features a single method m_Z that, upon execution, sequentially consumes a remote service Y (via a remote proxy rs_y) and a local service P, in that order. P and Y feature a simple request-response choreography on two methods m_Y and m_P, respectively. The execution of m_Z in the described SOSim can be abstracted as a series of discrete steps or the states of a timed automata depicted in Fig. 5. In this automata, transitions corresponding to service request timeout control have not been considered for the sake of clarity.

The values of $\mathbf{t}$ shown in the automata, namely $\mathbf{t}_{ez}, \mathbf{t}_{ey}$ and $\mathbf{t}_{ep}$, represent the local delay of executing the methods m_Z, m_Y and m_P, respectively. Those values are computed from the delay configuration provided for each method as soon as it is invoked. Alternatively, local delays are equal to zero if no delay configuration has been provided for the method. As illustrated in Fig. 5, these local delays are enforced by the SOSim *before* the method is executed. For instance, the execution of m_P occurs

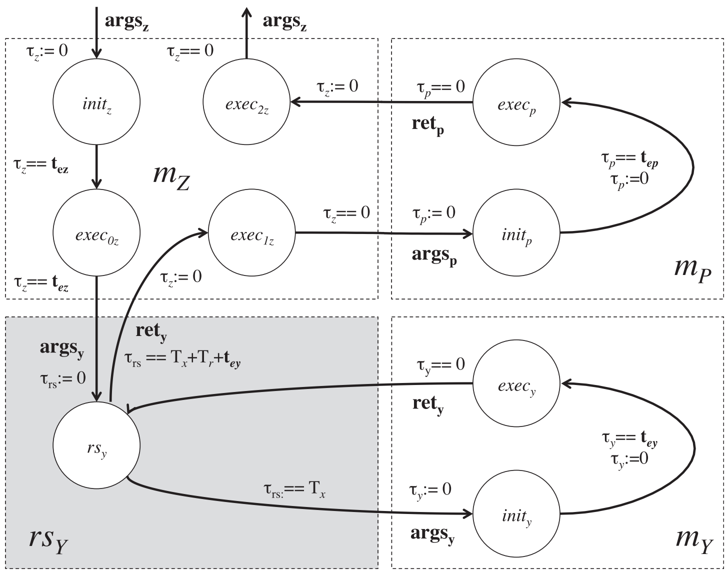

Fig. 5. Timed transitions associated with the simulation of a sample method m_Z.

in 0 s (simulation time) $\mathbf{t}_{ep}$ seconds after the service call is delivered to P. Similarly, the execution of the remote method m_Y will occur $T_x + \mathbf{t}_{ey}$ seconds after the service call has been delivered from m_Z to the remote proxy rs_Y, where T_x is the simulation time of transmitting the parameters and service call m_Y from Z to Y.

As the return values $\mathbf{ret}_y$ from m_Y have to be transmitted back to m_Z, the total delay T_{YZ} associated to the execution of m_Y, as perceived by m_Z is given by $T_{YZ} = T_x + T_r + \mathbf{t}_{ey}$, where T_r is the time it takes for the return values to be transmitted from Y to Z. Both T_r and T_x are dynamically determined at simulation time within ns-2.

In the described scenario, the approximate time morphism that results from the delay enforcement approach of the SOSim when executing m_Z, has been illustrated in Fig. 6. The figure also shows the impact upon input and output delivery times versus a realistic sequence of events. For instance, the transmission of $args_y$, that would have occurred at t_{nReal} is re-scheduled to occur at t_{nSim}, putting in evidence how the length of the modelled delays might determine the accuracy of the obtained results. The effects of the time morphism on these results are minimised when stateless service providers are simulated, as the order in which operations occur in those components does not alter the output of the service calls they handle.

7.2. The simulation proxy

Simulation proxies are created at instantiation time for all per-node components in the model regardless of their status as consumers and/or providers. Simulation proxies are in charge of delivering events (and their parameters) to the corresponding application-layer elements of a simulation model, as those events are dispatched by the ns-2 scheduler. When the proxied application is a service provider, the simulation proxy has the responsibility of intercepting all incoming service calls, computing the local delay corresponding to the execution of the call handler method, and enforcing that delay using the ns-2 scheduler.

7.2.1. Service binding mediation

Mediating service calls between consumers and providers in a model requires those calls and their corresponding responses to be intercepted by the simulation proxies. This must be accomplished in a transparent manner, without requiring any design-time coupling between consumer applications and the proxy objects. For that purpose, a series of steps are carried out at model instantiation time, before consumers can bind to the providers they require.

At instantiation time and during DS metadata examination, the contracts for the services offered by the components created from a *ComponentFactory* are read and stored by the instantiator. Later on, for each per-node component instance created from that DS factory, the instantiator requests the allocation of a corresponding simulation proxy from the proxy factory service. This request includes a list of the properties and services (if any) offered by the proxied component, together with a reference to the component itself. The simulation proxy allocated by the factory is furnished with the same properties and as the provider of the services offered by its *target* component, also keeping the reference to that target. In addition, the proxy exhibits the property name-value pair *(proxied, true)*.

Apart from the MAC and LMAC filter extensions described in Section 6.2 and that permit node multiplexing, service reference filters that do not point to global service providers are also extended with the criterion *(proxied = true)*. This extensions aims to ensure that consumers will be bound to services that meet the criteria of their original LDAP filters and that

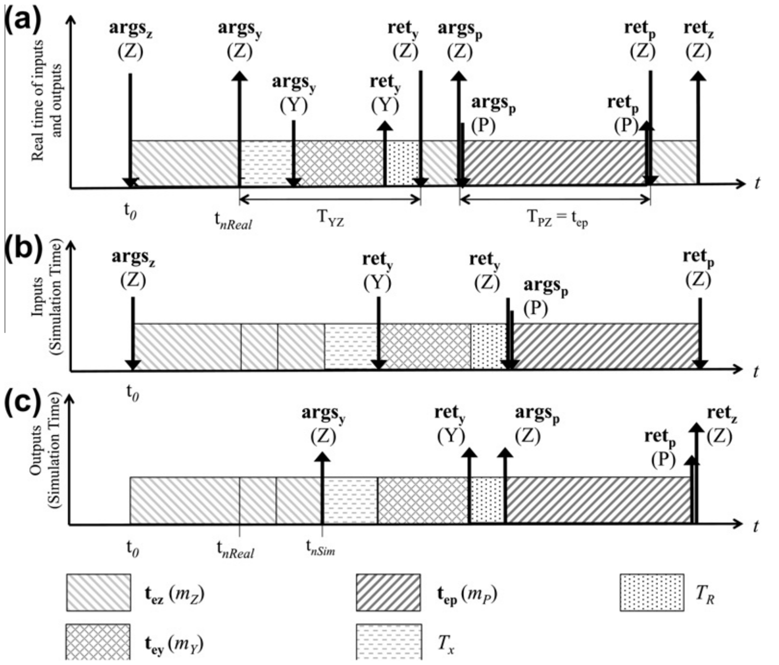

Fig. 6. Real time of inputs and outputs (a) versus input (b) and output (c) timing according to the approximate morphism implemented in the simulation environment.

expose the *(proxied, true)* property. The resulting binding is shown in Fig. 4. This supplementary criterion is what prevents consumers from binding to concrete instances of providers within a node, binding instead to the proxies that offer the services they require.

Service consumer applications may be programmed to issue service calls as soon as a provider is located. When consumer and provider are hosted by the same simulated node, those calls might be issued before the model is fully instantiated and prior to receiving a simulation start-up notification. In order to prevent any interaction from occurring before the networking elements of the model are ready to communicate data, proxies of service provider applications do not register the services they provide until the DES Portal notifies the application layer of the simulator that the network simulation layer has "warmed up" and that the simulation must start.

7.2.2. Mediation execution

A typical scenario exemplifying the way in which event delivery from the scheduler and application-layer delay enforcement are achieved is shown in Fig. 7. In the depicted scenario, a scheduled input is received by a consumer application that, as a result of the stimulus, proceeds to issue a synchronous service call to a provider within the same simulated node. The handler method for the service call in the provider has been configured with an application layer delay, which requires the re-scheduling of its execution by the application proxy.

As a first step, the dispatched *tellapp* invocation is delivered from ns-2 to the DES Portal. With the purpose of ensuring thread safety, the DES scheduler dispatcher thread in the network simulator remains blocked until the transmitted command or event has been handled by the application layer, giving the latter the change of issuing new schedule entries before unblocking.

When a command or event is received by the DES Portal from the ns-2 scheduler, a new delivery helper thread in charge of locating the command/event target simulation proxy is created. Meanwhile, the DES Portal Incoming Commands listener thread blocks until the child thread that has been created enters the WAIT state or until it terminates.

A command that has been received by a simulation proxy is parsed as it arrives. Once the target method has been identified, the proxy verifies with the configuration metadata if the method execution should be re-scheduled (because a delay for its completion has been assigned in the configuration) or if it should invoke the method immediately.

In the sample scenario shown in Fig. 7, the input as received by the consumer must be immediately handled, therefore, the proxy uses reflection to run the target method in the actual consumer application. At the same time, the scheduler dispatcher thread in the network simulation remains blocked, preventing the simulation time from advancing.

In this particular example, the method that handles the *tellapp* in the consumer application has to issue a call to a previously bound local provider as part of its programmed execution flow. Because consumers are bound to services provided by proxies rather than those provided by the actual applications, the service call is intercepted by the provider's proxy.

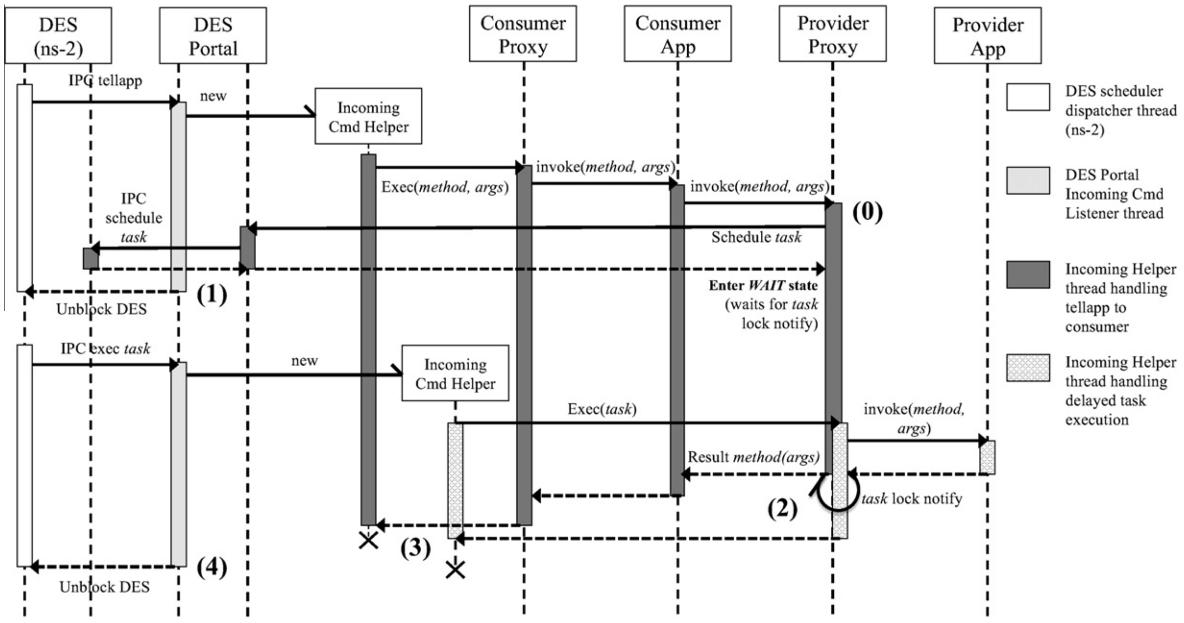

Fig. 7. Sequence diagram for a consumer invocation of a delay annotated method on a local provider, as mediated by the application proxies.

Similar to the reception of a *tellapp* command, the provider proxy will determine from the model configuration if the target method has been annotated with delay, which is the case in the sample scenario.

As discussed in Section 7.1, delay annotated methods have their execution re-scheduled with $\Delta t = t_e$, where t_e is method-specific according to the model configuration. As illustrated by marker 0 in Fig. 7, the proxy does not invoke the method in the concrete provider, but rather computes the total delay from the delay model, creates a *task* entry with a unique name for the current invocation and submits an *exec task* to the ns-2 scheduler, with itself (the proxy) as a target and with the task name as a parameter. This command is scheduled for delivery back to the proxy on the computed Δt. At this point (illustrated by marker 1), the provider proxy sets the current thread to WAIT state using a lock associated with the unique task name. The thread remains blocked until a different thread notifies of the completion of the task via the task's lock.

The DES Portal Incoming Listener Thread that had remained blocked waiting for its child thread to terminate or enter the WAIT state, unblocks. It proceeds to send a signal to the network simulator indicating that the previously delivered event, the initial *tellapp*, has been handled. From there, the scheduler dispatcher thread resumes its normal execution, eventually dispatching the *exec task* back to the proxy when Δt) has elapsed.

Reception of an *exec task* by a proxy implies that the method associated with the task name must be immediately invoked on the proxy's concrete target. In the sample scenario, the target method in the provider is not composed of any other services (and as a consequence does not issue any other service calls). As soon as the invocation ends, the proxy assigns the outcome of the execution to the task entry and notifies the corresponding lock about task completion (marker 2 in Fig. 7).

The thread handling the *exec task* command blocks until the original thread that had submitted the task completes its execution or enters the WAIT state again.

At the moment the thread that has submitted the task is notified of its completion, the activation of the provider proxy on that thread collects the corresponding result value and returns it to the consumer. In the example shown in Fig. 7, the consumer does not require any other service, completing its execution as soon as the value from the provider is obtained, thus terminating the thread in which the *tellapp* was originally being handled.

When the submitter of a task completes or enters the WAIT state, the thread handling the subsequent *exec task* terminates. This is indicated in the figure by marker 3. Once again, the DES Portal Incoming Listener Thread that delivered the *exec task* event is free to unblock the scheduler dispatcher when its child thread terminates (marker 4). At the end of this sequence, the consumer that had issued the service call at t_0, receives the response at simulation time $t_0 + t_e$, where t_e is the local delay of the call handler in the provider side. In a similar scenario, but where the consumed service is remotely provided, interactions are intercepted by a local RS proxy. This produces additional delays linked to request and response exchange over the network simulator.

7.3. Delay model implementation

The *getDelay* function, partially described in 5.1.3, is the only method that is required to exist in the classes defined within a delay model fragment. The contract for this method is defined by an interface outlined in the SOSim support bundles and it must be implemented by those classes.

When a simulation proxy is created for a component instance, its constructor examines the configuration corresponding to the associated component factory. If one or more methods within the configuration include `delaymodel` attributes, the classes indicated by those attributes are instantiated and retained by the proxy.

During the execution of a simulation, proxies determine if the methods targeted by the intercepted service calls have an associated delay model. If that is the case, the *getDelay* function is called in the corresponding delay model. This function can utilise the parameters of the service call and the state of the proxy (such as the current load) in the computation of t_e.

7.4. Timer-controlled tasks

In addition to timed input events and service calls issued at binding time, the interaction between components at simulation time can be triggered by timer-controlled tasks within an application element. Timer-controlled tasks refer to time-triggered service invocations between the component instances of a model.

The SOSim seeks to ensure that components or prototypes of a source system can be used in a simulation model. Therefore, guaranteeing that timers in those components operate in simulation time calls for an architectural constraint to be imposed on their design. Bundles that define simulatable components whose code creates instances of the *java.util.Timer* class must obtain those instances from the *Timer Factory* service. This service is provided by a support bundle included in the SOSim.

The default implementation of the Timer Factory returns instances of an extended version of the Timer class. When a consumer of that service requests an instance of a timer, a simulation proxy and its network simulation counterparts are allocated for the returned object (as seen in Fig. 4, Component Instance *X*). Considering timers as individual applications enables the scheduling of events that are specific to the timer operation, without involving the network simulator proxies for the applications that contain the timers.

The *TimerTasks* that have been scheduled to execute with a delay Δt are treated as delayed method calls by the application proxies that represent their target timers. These proxies create a corresponding *task* and submit an equivalent *exec task* to the scheduler. This event is later dispatched and sent back to the submitting timer through its proxy, that then runs the scheduled task.

7.5. Service discovery and distribution

The simulation environment provides a default implementation of the discovery and distribution RS APIs for SOA models, built around the Remote Service Admin (RSA) specification.

The default *discovery* mechanism works on a proactive fashion, advertising the exported services on a regular interval with the help of a timer and discovering single-hop neighbouring providers. Advertisement messages are of a proprietary format and contain a serialised collection of property sets. Each property set or *service endpoint description* characterises one of the exported services in a node and includes contract name, version, the properties of the service provider and those properties that are required by the OSGi RS specification. Despite the format provided with the default RS discovery implementation does not implement any of the standard service discover protocol formats, modellers can create or modify the RS discovery component so as to change the layout and content of the advertisements.

The byte stream that results from serialising the advertisement collection is broadcast using UDP with the help of the Network Adapter Service. This service forwards the data to ns-2 through the DES Portal which passes it to the intended ns-2 UDP agent for transmission.

Recipients of an advertisement are determined by ns-2 according to the network conditions. Upon hearing an advertisement, recipients de-serialise its contents and syntactically compare the service names it contains versus the internal registry in the node. If a new service instance is detected the node's RSA is notified, resulting in the local registration of an RS proxy for the discovered instance. At the same time, a *lease* with a configured duration is created in the discovery registry. If the service had been previously discovered and its RS proxy is still active in the registry, its lease is immediately renewed.

Service "undiscovery" occurs when an existing lease has not been renewed during a period of time equal or greater than the lease's assigned duration. With the help of a timer task, the discovery agent isolates the *endpoint* identifiers with expired leases and notifies the RSA of their removal.

The *distribution* container included in the environment is in charge of maintaining the RS proxies for the services that have been discovered or removed. RS proxies implemented by the default distribution container are similar to the simulation proxies described in sub-Section 7.2. Their main difference is that service calls and their responses are serialised, transmitted over the network simulator, and de-serialised prior to delivering them to the involved parties.

RS proxies keep a record of the requests that have been sent over the network to their delegating providers. It is by means of this record that received responses are matched to the appropriate requestors. Additionally, RS proxies control request timeouts using timer tasks. The execution of these tasks fail requests that remain unfulfilled after a configured t_{to} period has elapsed.

Remote requests received by the distribution container are immediately delivered to the simulation proxies of the services that have been exported, hence securing appropriate delay handling.

The high level of decoupling between applications and the DS distribution component permit that modellers create their own implementations of a distribution container. New container implementations do not require any changes to the application bundles in the model. Customised DS distribution components may serialise requests and responses as Remote Procedure Calls (RPC), SOAP, Javascript Object Notation-RPC (JSON-RPC), etc., also permitting different discovery and distribution strategies within RS elements to be implemented and simulated.

The RS implementations provided with the SOSim allow consumers to discover and access service providers at simulation time. This feature enables the structure of the model to emerge at simulation time, thus providing dynamic structure support. In contrast, simulation frameworks like [26,28,30] require consumer-provider relationships between simulated instances to be embedded in the model. Other frameworks facilitate the simulation of dynamic models, but cannot simulate MANETs [29,32] by default or don't permit components from the simulated system to be used as part of the model [31].

7.6. The network adapter service

The network adapter service decouples the application layer from the objects that provide access to the network. Applications with direct access to the network, like the RS implementation components, must obtain the sockets they need from this service. The simulatable implementation of the network adapter service allocates the corresponding ns-2 agents as soon as sockets are created, by sending commands to ns-2 via the DES Portal.

Fig. 8 illustrates how a UDP that originates from a node with IP *sip* is transmitted to its intended destination *dip:dport* via ns-2. Sequence *a*) in the figure corresponds to the steps associated to transmission request. In the sequence, an application (not shown) submits a message for *dip*:*dport*. The call is forwarded to the network adapter service, which produces an ns-2 command for the UDP agent counterpart of the application-layer socket.

Instead of executing the received ns-2 command, the DES Portal asks ns-2 to schedule it with 0 delay. If the *send* request passed to the UDP socket was triggered by a method handling a *tellapp*, *exec* or the reception of data from ns-2, scheduling the command permits that the submitting method continues its execution till it completes or blocks, which results in the DES being unblocked. When event dispatching in the scheduler resumes, the data transmission event is delivered to the counterpart UDP agent target *sip*:*sport*. From there onwards, the process follows the normal ns-2 execution flow. Fig. 8b illustrates the steps of the data reception sequence by socket *dip*:*dport*.

8. Mobile update service: an application case study

The application case study presented in this section illustrates one of the advantages of the SOSim: the possibility of evaluating an application using different delay models, where those delay models are decoupled from the elements that model application behaviour. This capability prevents delay modelling from becoming an obstacle to model continuity.

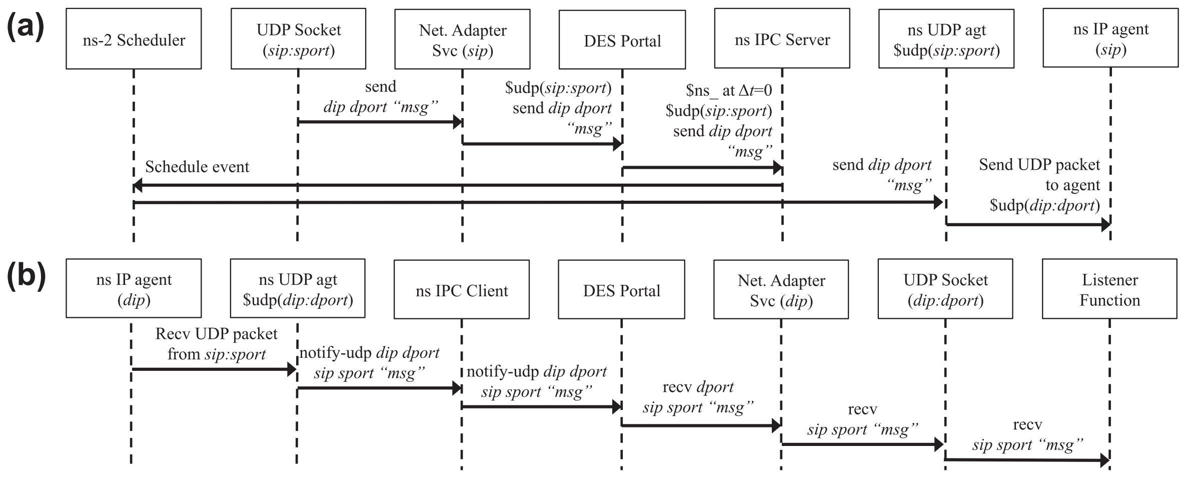
(a)
ns-2 Scheduler
UDP Socket (sip:sport)
Net. Adapter Svc (sip)
DES Portal
ns IPC Server
ns UDP agt $udp(sip:sport)
ns IP agent (sip)
send dip dport "msg"
$udp(sip:sport) send dip dport "msg"
$ns_ at Δt=0 $udp(sip:sport) send dip dport "msg"
Schedule event
send dip dport "msg"
Send UDP packet to agent $udp(dip:dport)
(b)
ns IP agent (dip)
ns UDP agt $udp(dip:dport)
ns IPC Client
DES Portal
Net. Adapter Svc (dip)
UDP Socket (dip:dport)
Listener Function
Recv UDP packet from sip:sport
notify-udp dip dport sip sport "msg"
notify-udp dip dport sip sport "msg"
recv dport sip sport "msg"
recv sip sport "msg"
recv sip sport "msg"

Fig. 8. Sequence diagram illustrating the re-direction of an UDP from a socket with port *sport* in a source node with IP address *sip*, to a destination socket *dip:dport*.

Table 2
Simulation parameters.

Parameter	Value
Number of nodes	75
Simulation time	60 s
Mobility model	Random Waypoint, 1 s maximum stop.
Maximum Speeds	5, 10, 15, 20, 25, 30 and 35 m/s
Area	1 km^2
Request timeout t_{to}	100 ms
Average provider t_e	60 ms
MAC	802.11p
Data rate	6 Mbps
Propagation model	TwoRayGround
Transmission range	200 m

8.1. Problem

The deployers of several Service-Oriented MANS are required to update the settings on the nodes within existing deployments. Each deployment consists of 75 nodes. All of these nodes run a client application that consumes an update service provider locally, or located within a single network hop. Nodes also share the same protocol stack. Discovery occurs as a result of proactive advertisements transmitted by the providers.

Consumer applications within each node issue an update request (service request) to a service provider as soon as an instance of the latter is discovered. Upon receiving a request, the provider node fetches the settings for the received consumer identifier and sends them back to the requestor. Consumers that receive a response do not issue new update requests during the next 24 h. If a response is not successfully received in 100 ms, consumers deem the request as failed and issue new service calls to an available provider.

In this fictitious scenario, deployers are aware that the process of fetching a request in the provider side has a duration of 60 ± 10 ms. They are also aware that different deployments have different average speeds according to their surrounding geography. The task of the deployers is to gauge the impact of deploying the update service in nodes that are able to reply to more than one request at a time, vesus nodes that queue service requests, replying on a first-come, first-served basis.

8.2. Method

In order to implement application-layer transducer functionalitym, the deployers modify the consumer application so as to record the succesful completion of a service call. This helps determine the success rate of issued requests and how fast the nodes in each deployment have their settings updated. In order to analyse the associated network load, ns-2 traces are used. The simulation parameters employed are presented in Table 2. Seven mobility traces were generated with *setdest* for each maximum speed setting, with the objective of minimising the random bias introduced by single scenarios, accounting for a total of 49 traces.

All traces were simulated against two different delay configurations for the execution of a method *getUpdatedSettings* in the provider nodes. The first configuration indicated a `meanms` = 60 (t_μ) and a `noisems` = 10 (t_w) in its component setup file

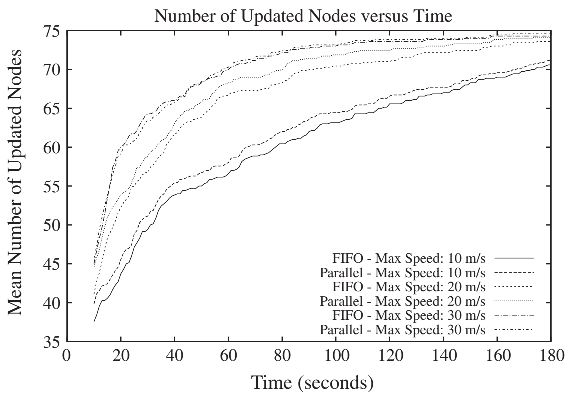

Fig. 9. Number of updated nodes during the first 3 min following the deployment of the updater service nodes.

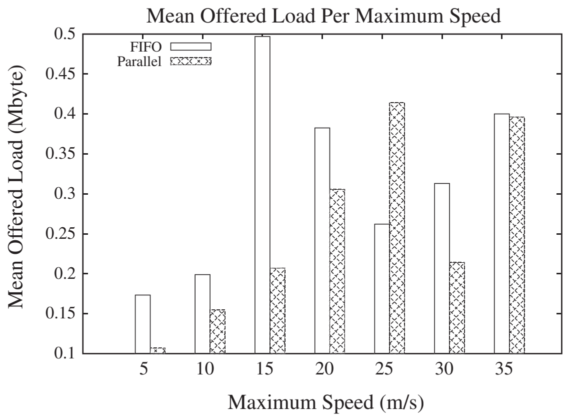

Fig. 10. Offered load associated to the update process during the first 3 min following the deployment of the updater service nodes.

for a t_e computed at simulation time according to (1). This configuration disregards the provider load, hence modelling a provider that can fulfil requests with the same delay regardless of concurrency. The second configuration included a `delaymodel` that approximates a FIFO response delay based on the number of concurrent requests being handled by the provider. The implementation of the delay model function approximates the response delay as: $t_e = n \times t_\mu + max[U(t_\mu - t_w, t_\mu + t_w), 0]$, where n is the number of service calls being processed by the provider when a new request arrives. The resulting simulations were executed in Eclipse Equinox 3.6 on top of a Java 6 virtual machine.

8.3. Results

The obtained results for the mean number of updated nodes versus time are depicted in Fig. 9 for scenarios with a maximum speed of 10, 20 and 30 m/s and over a period of 180 s. Simulations reveal that the impact of request queuing is minimal in terms of the speed at which the system is updated. However, the offered load resulting from request timeout and subsequent re-submission is, in average, higher than the load offered by the parallel deployment in low speed mobility settings, as depicted in Fig. 10. Both results show that as the maximum speed increases, the differences in terms of update speed and offered load between parallel and FIFO request execution models disappear. From the simulations, and based on the actual speed and available bandwidth in the modelled system, a decision can be made on the kind of updater node to be deployed.

From a model development perspective, this example illustrates how the task of simulating the same MANS over a number of mobility settings and networking parameters once the application-layer of the model has been developed, resembles the task of executing multiple simulations with ns-2 models. It also shows the simplicity with which different behaviours of MANS can be simulated by changing the way in which application-delay is modelled. In addition, once results have been obtained, the same updater service component can be deployed to the source system with minimal modifications.

9. Discussion

The SOSim presented in this paper exploits a model-driven approach based on SOA and permits OSGi components to be utilised within a model. These components, which represent application-layer elements, can be interchanged with counterparts from a source system. This approach promotes the separation of concerns within a model where application-layer elements are decoupled from networking elements, thus maximising their re-usability. Additionally, the possibility of sharing components between source system and simulation model transforms the SOSim into a debugging and refinement tool, as well as a valid means of feasibility testing in the context of OSGi.

The utilisation of ns-2, one of the most widely used simulators in MANET research, provides support for mobility and a set of commonly used networking protocols. ns-2 also facilitates the description of elaborate experimental frames by means of its scripting capabilities.

Although SOSim environments such as those outlined in [30,31] have strived to make available realistic network simulations using OMNeT++, their proprietary means of application layer modelling lack support for highly dynamic systems like MANS. In that regard, the reliance on an industrial-strength service platform like OSGi constitutes a clear advantage over existing solutions.

The adoption of SOA, not just as an architectural approach to the construction of simulation models, but as the design paradigm of the simulation environment, decouples the simulation services that integrate the SOSim from each other. SOA paves the way for the extension and customisation of the framework and permits the incorporation of new RS API implementations. This enables the evaluation of service discovery, distribution, selection and composition strategies in combination with the applications they support.

The versatility of the service-oriented simulation framework minimises the impact of the constraints imposed by the SO-Sim on simulatable components and on the physical deployment of simulation services and models. In spite of that, users must be aware of the relevance of those constraints when the components to be simulated utilise third party artefacts that implement their own timing control, require direct access to the network or define stateful applications that have not been declared as DS components.

Despite the default implementation of the proposed environment depends on ns-2, a discrete event simulation that does not support parallelism by default; the structural openness provided by SOA facilitates the replacement of ns-2 with other tools that support parallel simulation execution.

10. Conclusion and future work

Simulations of Service-Oriented MANS require not only the realistic replication of communications over a MANET. They also calls for the accurate reproduction of those interactions that characterise a service-oriented system. A simulation environment for Service-Oriented MANS must permit the effects of mobility and topology change on the application layer of MANS to be evaluated. This environment is posed to play a central role on the fine-tuning of MANS design and implementation.

The SOSim framework presented in this paper narrows the gap between source system and simulation model. Its reliance on SOA and OSGi provides the opportunity of utilising elements from a source system as part of a simulation model. The performance of these components can be then evaluated under operational settings that are consistent with the scenarios posed by MANETs.

Future works on the SOSim must seek to incorporate default hardware models. These models can assist with request and process prioritisation based on node load. Additionally, the SOSim must be extended to integrate energy consumption awareness.

In terms of the architecture of the SOSim environment, the possibility of node multiplexing across several OSGi framework instances and compliance with HLA should be explored as means of enhancing the scalability of the supported simulations.

References

[1] I. Chlamtac, M. Conti, J.J.N. Liu, Mobile ad hoc networking: imperatives and challenges, Ad Hoc Networks 1 (1) (2003) 13–64.
[2] H.M. Hajj, W. El-Hajj, M.E. Dana, M. Dakroub, F. Fawaz, An extensible software framework for building vehicle to vehicle applications, in: Proceedings of the 6th International Wireless Communications and Mobile Computing Conference (IWCMC '10), ACM, 2010, pp. 26–31.
[3] J.-Y. Tigli, S. Lavirotte, G. Rey, V. Hourdin, M. Riveill, Lightweight service oriented architecture for pervasive computing, International Journal of Computer Science Issues (IJCSI) 4 (1) (2009).
[4] Y. Natchetoi, H. Wu, Y. Zheng, Service-oriented mobile applications for ad-hoc networks, in: IEEE International Conference on Services Computing 2008 (SCC '08), IEEE, 2008, pp. 405–412.
[5] A. Neyem, S.F. Ochoa, J.A. Pino, Integrating service-oriented mobile units to support collaboration in ad-hoc scenarios, Journal of Universal Computer Science 14 (1) (2008) 1–35.
[6] N. Suri, M. Marcon, R. Quitadamo, M. Rebeschini, M. Arguedas, S. Stabellini, M. Tortonesi, C. Stefanelli, An adaptive and efficient peer-to-peer service-oriented architecture for manet environments with agile computing, in: Network Operations and Management Symposium Workshops 2008, IEEE, 2008, pp. 364–371.
[7] R. Tergujeff, J. Haajanen, J. Leppanen, S. Toivonen, Mobile soa: service orientation on lightweight mobile devices, in: IEEE International Conference on Web Services 2007 (ICWS 2007), IEEE, 2007, pp. 1224–1225.
[8] K. Abrougui, A. Boukerche, S. Samarah, Design and performance evaluation of QOS aware and location based service discovery protocol for vehicular networks, in: Proceedings of the 13th ACM International Conference on Modeling, Analysis, and Simulation of Wireless and Mobile Systems (MSWIM '10), ACM, 2010, pp. 73–80.
[9] D. Huang, X. Zhang, M. Kang, J. Luo, Mobicloud: building secure cloud framework for mobile computing and communication, in: Fifth IEEE International Symposium on Service Oriented System Engineering (SOSE) 2010, IEEE, 2010, pp. 27–34.
[10] M. Satyanarayanan, P. Bahl, R. Caceres, N. Davies, The case for vm-based cloudlets in mobile computing, IEEE Pervasive Computing 8 (4) (2009) 14–23.
[11] D. Huang, Mobile Cloud Computing, IEEE COMSOC Multimedia Communications Technical Committee (MMTC), E-Letter, 2011.
[12] Y. Chen, Modeling and simulation for and in service-orientated computing paradigm, Simulation 83 (1) (2007) 3–6.
[13] R. Barr, Z.J. Haas, R. vanRenesse, Scalable wireless ad hoc network simulation, in: Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad hoc Wireless, and Peer-to-Peer Networks, Auerbach Publications, Boston, MA, 2005, pp. 297–311.
[14] S. McCanne, S. Floyd, The Network Simulator – ns-2. <interrefhttp://www.isi.edu/nsnam/nsurlhttp://www.isi.edu/nsnam/ns> (11.03.11).
[15] A. Varga, The OMNeT++ discrete event simulation system, in: Proceedings of the European Simulation Multiconference (ESM'2001), 2001.
[16] OSGi Alliance, OSGi Service Platform Release 4 Version 4.2 Core Specification, 2009. <http://www.osgi.org> (08.12.10).
[17] IEEE, IEEE Standard for Modeling and Simulation (m&s) High Level Architecture (HLA) – Framework and Rules, 2010.
[18] P. Gustavson, C. Tram, L. Root, Moving towards a service-oriented architecture (SOA) for distributed component simulation environments, in: Proceedings of 2006 Spring Simulation Interoperability Workshop, Simulation Interoperability Standards, Organization, 2006.
[19] B. Moller, C. Dahlin, A first look at the hla evolved web services api, in: Proceedings of the 2005 Euro Simulation Interoperability Workshop, Simulation Interoperability Standards, Organization, 2005.
[20] B. Moller, S. Lof, Mixing service oriented and high level architectures in support of the gig, in: Proceedings of 2005 Spring Simulation Interoperability Workshop, Simulation Interoperability Standards, Organization, 2005.
[21] Q. Xiang, G. Chen, Y. Wang, Distributed simulation based on web enabling hla, in: 2nd International Conference on Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC), IEEE, 2011, pp. 2062–2064.
[22] H. Zhu, G. Li, L. Zheng, Introducing web services in hla-based simulation application, in: Proceedings of the 7th World Congress on Intelligent Control and Automation (WCICA 2008), IEEE, 2008, pp. 1677–1682.

[23] W.T. Tsai, C. Fan, Y. Chen, R. Paul, Ddsos: a dynamic distributed service-oriented simulation framework, in: Proceedings of the 39th Annual Symposium on Simulation, IEEE, 2006, pp. 160–167.
[24] S. Mittal, J.L. Risco-Martin, B.P. Zeigler, Devs/soa: a cross-platform framework for net-centric modeling and simulation in devs unified process, Simulation 85 (7) (2009) 419–450.
[25] K. Al-Zoubi, G.Wainer, Performing distributed simulation with restful web-services, in: Proceedings of the 2009 Winter Simulation Conference (WSC), 2009, pp. 1323–1334.
[26] H. Sarjoughian, S. Kim, M. Ramaswamy, S. Yau, A simulation framework for service-oriented computing systems, in: Proceedings of the 2008 Winter Simulation Conference (WSC), 2008, pp. 845–853.
[27] D.R. Hild, H.S. Sarjoughian, B.P. Zeigler, Devs-doc: a modeling and simulation environment enabling distributed codesign, Systems, Man and Cybernetics Part A: IEEE Transactions on Systems and Humans 32 (1) (2002) 78–92.
[28] M.A. Muqsith, H.S. Sarjoughian, A simulator for service-based software system co-design, in: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools '10). Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST), 2010, pp. 54:1–54:9.
[29] M.A. Muqsith, H.S. Sarjoughian, D. Huang, S.S. Yau, Simulating adaptive service-oriented software systems, Simulation 87 (11) (2011) 915–931.
[30] F. Bause, P. Buchholz, J. Kriege, S. Vastag, A framework for simulation models of service-oriented architectures, in: Proceedings of the SPEC International Workshop on Performance Evaluation: Metrics, Models and Benchmarks, Springer-Verlag, 2008, pp. 208–227.
[31] H. Neema, A. Kashyap, R. Kereskenyi, Y. Xue, G. Karsai, Soamanet: a tool for evaluating service-oriented architectures on mobile ad-hoc networks, in: 2010 IEEE/ACM 14th International Symposium on Distributed Simulation and Real Time Applications (DS-RT), IEEE, 2010, pp. 179–188.
[32] S. Mittal, Agile net-centric systems using devs unified process, Intelligence-Based Systems Engineering 10 (2011) 159–199.
[33] T. Kim, M.H. Hwang, D. Kim, Devs/ns-2 environment: an integrated tool for efficient networks modeling and simulation, The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology 5 (1) (2008) 33–60.
[34] W. Wang, W. Wang, Y. Zhu, Q. Li, Service-oriented simulation framework: an overview and unifying methodology, Simulation 87 (3) (2011) 221–252.
[35] Y.J. Kim, J.H. Kim, T.G. Kim, Heterogeneous simulation framework using devs bus, Simulation 79 (1) (2003) 3–18.
[36] T. Erl, SOA Principles of Service Design, Prentice Hall, 2007.
[37] The Eclipse Foundation, Eclipse Communication Framework Project. <http://www.eclipse.org/ecf> (03.11.11).
[38] E. Gutman, C. Perkins, J. Veizades, M. Day, Rfc 2608: SLPv2 – A Service Location Protocol, IETF, 1999.
[39] S. Cheshire, M. Krochmal, Dns service discovery (internet draft). <http://www.dns-sd.org> (11.12.11).
[40] J. Harri, F. Filali, C. Bonnet, M. Fiore, Vanetmobisim: generating realistic mobility patterns for vanets, in: International Conference on Mobile Computing and Networking: Proceedings of the 3rd International Workshop on Vehicular Ad Hoc Networks, ACM, 2006, pp. 96–97.