Termite: Emulation Testbed for Encounter Networks

Rodrigo Bruno, Nuno Santos, Paulo Ferreira
INESC-ID / Instituto Superior Técnico, University of Lisbon
{rodrigo.bruno, nuno.santos, paulo.ferreira}@inesc-id.pt

ABSTRACT

Cutting-edge mobile devices like smartphones and tablets are equipped with various infrastructureless wireless interfaces, such as WiFi Direct and Bluetooth. Such technologies allow for novel mobile applications that take advantage of casual encounters between co-located users. However, the need to mimic the behavior of real-world encounter networks makes testing and debugging of such applications hard tasks.

We present Termite, an emulation testbed for encounter networks. Our system allows developers to run their applications on a virtual encounter network emulated by software. Developers can model arbitrary encounter networks and specify user interactions on the emulated virtual devices. To facilitate testing and debugging, developers can place breakpoints, inspect the runtime state of virtual nodes, and run experiments in a stepwise fashion. Termite defines its own Petri Net variant to model the dynamically changing topology and synthesize user interactions with virtual devices. The system is designed to efficiently multiplex an underlying emulation hosting infrastructure across multiple developers, and to support heterogeneous mobile platforms. Our current system implementation supports virtual Android devices communicating over WiFi Direct networks and runs on top of a local cloud infrastructure. We evaluated our system using emulator network traces, and found that Termite is expressive and performs well.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Data communications*; C.4 [**Performance of Systems**]: Reliability, availability, and serviceability

General Terms

Experimentation, Performance, Algorithms

Keywords

Android, Virtualization, Emulator, CloudStack, OpenStack, Wi-fi Direct, Debug, Emulation

1. INTRODUCTION

Today, smartphones and tablets are equipped with network interface technologies, such as WiFi Direct, that enables wireless communication between nearby devices without a pre-existing infrastructure. This capability has prompted new mobile applications that leverage occasional encounters of users in various social contexts, e.g., for mobile social networking [22, 13, 9], content sharing [15, 12], mobile ad-hoc gaming [5, 16], and more [19, 21, 31, 24]. Specialized middleware [3, 32, 25, 18, 7] has also been produced to facilitate the development of such applications. While the general term *ad-hoc network* has been used to designate opportunistically-formed networks of devices (e.g., sensor networks), we use the term *encounter networks* to refer to this special case of ad-hoc networks that target co-located moving personal devices and involve social interactions.

Encounter networks have unique characteristics that make software testing and debugging cumbersome for developers. To test an application, a developer needs to mimic the movement of devices and simulate user interactions in a realistic encounter network using real devices or software emulators. However, this task is error-prone and time-consuming as the complexity of the network increases in terms of number of nodes, dynamism of its topology, and expressiveness of user interactions. Debugging is burdensome too, because it is difficult to replay an execution under the same exact conditions, and to perform step-wise execution in such a distributed environment to probe for errors.

In spite of extensive work on testbeds and developer tools, existing systems fall short at providing adequate support for testing and debugging encounter-networked applications. Stock mobile platforms like Android and iOS ship with development toolkits [1, 4] that include a virtual device emulator and an automated UI testing framework. However, these tools focus on single devices only, and are therefore inadequate to emulate multi-node encounter networks. Various alternative systems target multi-node environments, but are insufficient to address the developers' needs previously mentioned: some provide network simulation capability only (i.e., no emulation support) [14], others provide network emulation capability, but focus only on low-level network protocols or offer no UI automation support [34, 6, 27], and others [8, 2] are testbeds too general to significantly ease the development effort of encounter-networked applications (apps for short) as they lack the appropriate abstractions to model encounter networks.

This paper presents a testbed system named Testbed for Encounter Mobile applications (Termite). Termite provides testing and debugging support for encounter network applications. To test an application, developers can specify a virtual

encounter network, including its topology, how it evolves over time, and synthetic user interactions with the application on virtual devices. Based on this specification, Termite emulates the behavior of the virtual network nodes by leveraging software emulators running on a clustered computer infrastructure (a public cloud for example). During application emulation, Termite tests the application output to detect potential errors or exceptions. Developers can debug the application by placing breakpoints and re-executing the application on the virtual encounter network. On a breakpoint hit, a developer can inspect the current execution state of any virtual node and carry out the application execution in a step-wise fashion.

Termite makes three main technical contributions. First, it defines a novel model for specifying encounter networks, T-Net. Although prior networking models, such as ns2's [14], offer great versatility in modeling network topologies, they lack adequate abstractions to represent UI interactions of users. This limitation is particularly impairing for the emulation of encounter networks, in which the users' interactions with an application may affect the movement of nodes and vice-versa. T-Net allows for the specification of such interdependencies by leveraging the expressiveness power of petri nets [30]. Second, Termite features a general and resource-efficient design which enables emulation of applications targeting Android, iOS, or other mobile platforms by implementing Termite-specific extensions for the software emulators of the respective platforms. Third, Termite provides an implementation prototype that includes: (i) a scripting language for modeling encounter networks, and (ii) WiFi Direct emulation support targeting Android platforms (which do not support WiFi Direct).

We evaluate our Termite prototype on a local university cluster, using CloudStack as the underlying virtualization platform. The goal is to gauge Termite's expressiveness power and performance overheads. We implemented a simple micro-blogging application for encounter networks and managed to express various scenarios for synthetic scenarios based on real world situations, showing that Termite is expressive. Performance-wise, the evaluation shows that our system adds negligible overheads to the time latency introduced by the virtualization software.

2. OVERVIEW

Termite is an emulation testbed aimed at assisting developers of encounter network applications to test and debug their apps. As in a typical software development cycle, the programmer starts by coding the app. The next phases are then assisted by Termite: specification, testing, and debugging.

T-Net specification Before executing the program, the developer must provide a T-Net specification (spec). The T-Net spec describes the topology evolution of the encounter network (i.e., how many nodes are part of the network, how they are connected, and how their connections change over time) and the users' UI interaction on the network's virtual nodes (i.e., the input to be simulated at each node, and the output that is expected at each node).

Testing The T-Net spec is then provided to Termite along with the packaged application binary. Termite enters the testing phase, in which software emulator instances emulate the topology evolution and UI interaction of the virtual encounter network specified by the developer. If the app produces unexpected outcome on any virtual node (i.e., contradicting the T-Net spec), Termite generates an exception and the test fails; otherwise the test succeeds.

Debugging To debug the program, a developer can re-execute the experiment in debug mode, possibly after changing the application code or the T-Net spec. The developer can place breakpoints, execute the experiment step-by-step, and inspect the execution state of specific nodes.

In the following sections we describe Termite in detail. Next, we start by presenting the encounter network model that lies at the basis of the experiment specification.

3. MODEL

To motivate our model for encounter networks, we describe an application example.

3.1 Motivating Example: NearTweet

Figure 1 illustrates a simple experiment running on Termite; it is a basic yet representative mobile application. The goal of this app, which we call NearTweet, is to provide a location-aware microblogging service. Essentially, it provides a simple chatting service that enables unrelated users to communicate when clustered together in spaces such as stadiums, malls, traffic jams, concerts, and restaurants.

A simple NearTweet interface is depicted in Figure 1 (left-hand side). There is a single screen that pops up a message in a *status panel* whenever other users enter or leave the communication range of a device. While connected, the user has access to an *input panel* where she can type in a message and a *post button* to broadcast that message to all nodes reachable over the encounter network currently formed. In addition, there is an *output panel* that displays the messages sent by other users currently connected.

Consider the experimental scenario represented in Figure 1 (right-hand side). There are two (virtual) users—Alice and Bob—carrying their own devices. At time t_0 the experiment starts with both users located apart from each other. At time t_1, they approach each other and their devices connect, each device popping a "Connected" message. Immediately after (t_2), Alice sends a message introducing herself—"Hello, I'm Alice"—to which Bob replies—"Hello Alice.". Because Alice is in a hurry, she steps away at t_3 and devices disconnect; the communication is interrupted, and message "Disconnected" is displayed on each device.

3.2 Model Requirements

Despite their simplicity, both the application and the experimental scenario represented in Figure 1 suffices to illustrate the key events that need to be modeled in more complex applications and usage patterns. In particular, there are *network* and *UI* events. Network events comprise: joining a group, leaving a group, and exchanging messages. UI events consist of: providing user input, and checking specific output. It is easy to see that scenarios featuring a larger number of devices and more complex communication patterns would share the basic network events of the previous example. NearTweet's in / out texting events could also generalize to I/O events previously issued by more complex UI layout elements.

To specify encounter networks such as the one described, the Termite model must satisfy three requirements as follows:

R1. Expressiveness To serve a large number of applications and usage scenarios, developers should be able to represent complex scenarios, featuring arbitrary network topologies and UI events. Network topologies could change over time, with nodes joining, and leaving groups. Developers must control user input and check output conditions, and express causality dependencies between events (e.g., message is shown on Bob's device only after being sent by Alice).

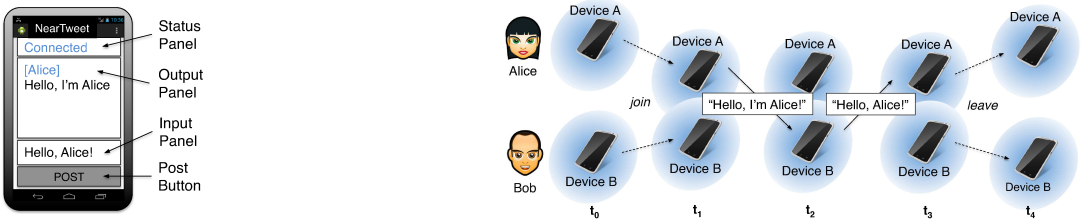

Figure 1: Example of NearTweet—a chatting mobile app—running on a simple encounter network. The application screen is shown on the left, and the network topology evolution on the right.

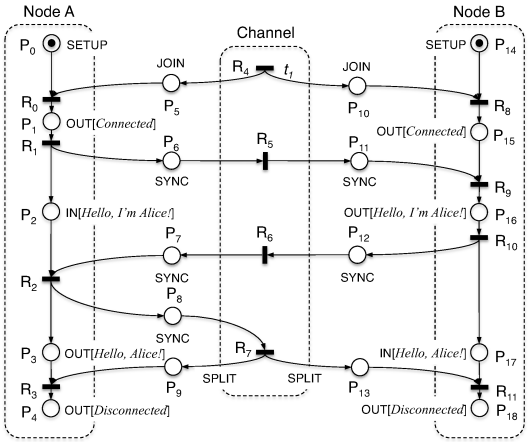

Figure 2: T-Net modeling example from Figure 1.

R2. Reproducibility To allow for proper program debugging, Termite experiments must be reproducible, i.e. running the same experiment multiple times must produce the same outcome.

R3. Execution control As in any debugger tool, the developer must perceive the emulation process as a fine-grained sequence of events that can be controlled by placing breakpoints, allowing for stepwise execution, and exposing the runtime state of apps.

To address these requirements, we leverage the *Termite Net* (T-Net), which we address now.

3.3 T-Net Model

Termite Net (T-Net) is a novel Petri Net variant to specify encounter networks. Petri Nets are powerful constructs that exist in different variants and have been applied in various contexts, namely concurrent systems [30]. To introduce T-Net, we first explain how the example of Figure 1 can be modeled using a T-Net, and then present T-Net's formal definition.

Figure 2 presents a T-Net spec of the NearTweet encounter network introduced in Section 3.1. It comprises three concurrently executing *components*: two *nodes*—representing Alice and Bob's virtual devices —and a *channel*—representing the wireless group they formed. Places represent relevant system events and can be classified as: *setup*, *IO*, and *network* places.

Setup places correspond to bootstrapping events. Every node component must start with a SETUP place responsible for the component's initialization. For this reason, tokens are initially placed in SETUP places. *GUI IO places* represent I/O events and can be of two types: input or output. Input places IN[x] are used to emulate a user providing specific input; x is a developer-defined parameter that defines the input to be provided when that specific place gets marked. In Figure 2, input places emulate Alice typing message "Hello, I'm Alice" (P_2), and Bob typing message "Hello Alice" (P_{17}). Output places OUT[x] validate the generated output against a developer-defined condition x. In Figure 2, output is validated to check messages "Connected" (P_1 and P_{15}), "Hello, I'm Alice" (P_{16}), "Hello Alice" (P_2), and "Disconnected" (P_4 and P_{18}). *Network places* represent network events and can be of three types: join, leave, and sync. JOIN and LEAVE places represent nodes joining and leaving a wireless network group, respectively. In Figure 2, there are two join places (P_5 and P_{10}) and two leave places (P_9 and P_{13}). The SYNC place is used across nodes to synchronize the arrival of a specific message from the network. For example, P_6 and P_{11} allow us to ensure that Bob only types a response once it receives the hello message from Alice.

In T-Net, transitions are associated with specific components and represent emulation steps. In other words, they represent synchronization points that are relevant to the developer. In fact, by carefully inspecting the transitions of Figure 2's T-Net, we can detect an almost one-to-one correspondence with the expected step sequence described in Section 3.1. In some cases, transitions are enabled by a single place only (e.g., R_1). In others, transitions are enabled by multiple places (e.g., R_0). Transitions can also depend on time. In particular, the developer can associate a time t with a transition to make sure that, even if the transition is enabled, it fires no earlier than t. This time is compared against a global clock that counts the time since the emulation has started. Transition R_4, for example, is set to fire no earlier than $t = t_1$, which is the time Alice and Bob meet.

Emulating a T-Net consists simply of tracking the execution path of the system along the flow graph of transitions connected by places. There is a start state defined by places P_0 and P_{14}, and a finish state defined by places P_4 and P_{18}. The emulation is successful if it reaches the end state, otherwise it fails. The end state is not reached if a place has failed to fire. This can happen if an exception was thrown by the application, or if the place validation has failed (e.g., Bob received an unexpected message in P_{16}). Note that, once an output event gets marked, if the corresponding output condition is false, the pointed transition will not be enabled and the emulation aborts.

T-Net naturally allows debugging by allowing the placement of breakpoints. A breakpoint is a stop condition defined by a *breakpoint transition*, i.e., a transition r_b that forces the emulation process to stop once r_b is enabled. For example, to suspend the emulation immediately after Bob's device receives Alice's message, but before this message is displayed, we would define breakpoint $b = R_9$. To enforce this breakpoint, transition R_9 would no longer be fired once it has become enabled. To resume the execution, it is only necessary to fire the transition.

Figure 3 summarizes what has been said so far by providing a more formal definition of T-Net. A T-Net consists of a finite set of components, places, and transitions. Components can

A T-Net is an 8-tuple $N = \{C, T, P, R, F, M_s, M_f, B\}$ where:

- $C = \{c_1, c_2, \ldots, c_k\}$ is a finite set of components where $type(c_i) \in \{NODE, CHANNEL\}$,
- T is the maximum duration of execution,
- $P = \{p_1, p_2, ..., p_m\}$ is a finite set of places where $type(p_i) \in \{SETUP, IN[x], OUT[x], JOIN, LEAVE, SYNC\}$,
- $R = \{r_1, r_2, ..., r_n\}$ is a finite set of transitions where $owner(r_i) \in C$ and $time(r_i) \rightarrow \{-\} \cup [t_0, \ldots, t_T]$,
- $F \subset (P \times R) \cup (R \times P)$ is a set of arcs (flow relation),
- M_s: $P \rightarrow \{0,1\}$ is the starting marking,
- M_f: $P \rightarrow \{0,1\}$ is the final marking,
- $B = \{b : b \in \varnothing \vee b \in R\}$ is a finite set of breakpoint transitions,
- $P \cap R = \varnothing$ and $P \neq \varnothing$.

The rule for transition enabling and firing is as follows:

1. A transition r is enabled if each input place p is marked with a token, and p is not marked as final, i.e., $m_f(p) = 0$.
2. An enabled transition r will fire if: (a) $time(r)$ is either undefined "-" or greater than or equal to the current time, and (b) r is not set as a breakpoint transition, i.e. $r \notin B$.
3. Firing an enabled transition r removes the tokens from the input places of r, and adds a token to each output place of r.

Figure 3: T-Net definition.

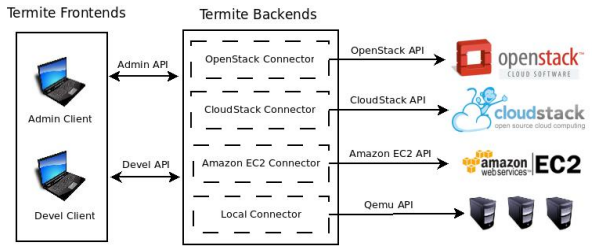

Figure 4: Termite architecture.

be nodes or channels. Transitions are associated with nodes or with channels. Places connect transitions via a flow relation. There are three types of places: setup, I/O, and networks. The developer is free to specify all these elements (including the parameters to input places) and to define breakpoint transitions. Emulating a T-Net follows the transition enabling and firing rules indicated in Figure 3.

The T-Net model satisfies the requirements stated in Section 3.2. As we can see, T-Net offers an expressive framework for developers to model complex encounter networks (**R1**). In addition, T-Net ensures that the same experiment can be repeated by forcing components to execute along the flow relation specified by the developer (**R2**). Finally, by allowing for the specification of synchronized execution steps through transitions and the definition of breakpoints, T-Net provides adequate mechanisms for fine-grained control (**R3**).

4. DESIGN

4.1 Architecture

Termite is designed with three main requirements in mind. First, it must provide an emulation service based on T-Net. Second, in doing so, resources must be managed efficiently by enabling sharing the hosting infrastructure across multiple developers. Third, Termite must be general and easily support the emulation of arbitrary mobile platforms through the incorporation of platform-specific extensions.

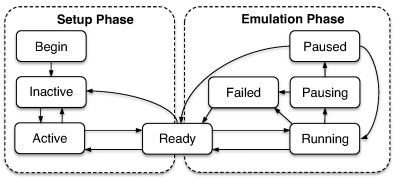

Figure 5: State machine of a devel session.

Termite comprises two *frontends* and multiple *backends* (see Figure 4). Termite frontends consist of two software clients: one to be used for admin sessions (creation of Emulator Disk Images, manage account settings) and the other for developer (devel for short) sessions (described in Section 4.2). Both frontends can be configured dynamically to use one of the available backend connectors. Each of the available backend connectors must respect both the developer and administrator APIs (the minimum set of operations necessary to enable developer and administrator sessions to operate with Termite).

Termite aims at supporting multiple backends. It uses virtualization to deploy emulators and, instead of implementing a distributed virtualization system, Termite allows the addition and configuration of connectors. By using such connectors, users can take advantage of already existing and well-known cloud platforms such as: OpenStack, CloudStack, or even Amazon EC2. Each Termite backend connector has the responsibility of processing Termite API operations into cloud platform specific operations.

Termite is internally designed to support seamless emulation of mobile nodes possibly spanning multiple backend cloud platforms. To do so, it requires only a platform-specific Emulator Disk Image (EDI) to be provided, and the enhancement of the EDI with Termite extensions. These extensions enforce the T-Net model on all emulator instances booted from that EDI. Section 4.3 describes these extensions in detail.

4.2 Devel Session Operations

Devel sessions allow developers to test and debug their encounter networked apps on Termite. Developers have access to these sessions through specific commands issued on the devel client. The devel client implements these commands in coordination with Termite's backend components.

The lifecycle of a devel session is best described by the state diagram shown in Figure 5. Two main phases can be identified: *setup phase* and *emulation phase*. In the setup phase, the developer loads the T-Net for his experiment, instructs Termite to allocate the necessary instances in the hosting infrastructure, and to deploy both the T-Net specification and the application package to the emulator instances. At this point the experiment is set up and emulation can start. The execution phase allows the developer to emulate the previously configured encounter networked app. The developer can either test the application by running the experiment and check if it has terminated successfully, or debug the application by issuing typical debugger commands like setting breakpoints, pausing execution, resuming execution step by step, etc.

The end goal of the setup stage is to prepare the experiment to be executed by the developer, and finishes when the devel session reaches state READY. After authentication, a session starts in state INACTIVE. Next, the developer must execute three steps as follows:

1. Provide a T-Net specification The developer must provide a specification (spec) of the T-Net to be emulated

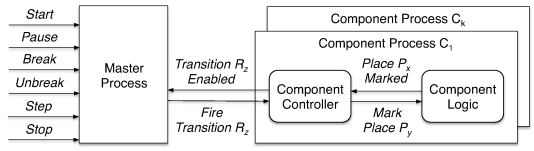

Figure 6: Actors involved in T-Net emulation.

(see Section 5.2 on T-Net specification). To produce such a spec, the developer can be assisted by an implementation-dependent language (see Section 5.2) or by a GUI tool. After analysing the T-Net spec, the devel client creates a profile of the emulator instances to be created: determines the number of emulator instances, their EDI types, and EDI parameters provided by the developer (e.g., memory size).

2. Bootstrap emulator instances: Based on the emulator profile determined previously, the developer must now request the instantiation of the emulators on the hosting infrastructure through the platform connector. As soon as all instances finish bootstrapping, the connector notifies the client. At this point, the client's state is updated to ACTIVE.

3. Deploy app and T-Net spec The last step of the setup phase consists of deploying the app package and T-Net spec to the allocated emulator instances. The client executes it bypassing the platform connector and performing the transfer directly to each emulator instance. After completing the deployment on every instance, the client changes its state to READY, and the emulation phase can start.

4.3 T-Net Emulation

As soon as a developer completes the setup phase of an experiment, the emulation phase can start, allowing the developer to emulate the T-Net either completely (testing) or in a stepwise manner (debugging). Termite emulates the T-Net as described in Section 3.3 by relying on a distributed algorithm executed between the devel client and emulator instances. To explain this algorithm, we start by describing it using more abstract concepts, and then map these concepts to the Termite components that effectively implement them: devel client and emulator instances.

The algorithm to emulate a T-Net is carried out by the interplay of the conceptual modules shown in Figure 6. Each node or channel of the T-Net is emulated independently by a *component process*. Internally, a component process consists of two sub-modules: *component logic*, and *component controller*. While the former emulates the T-Net's application, the latter checks whether the application executes according to the transition sequence specified for that component. Component processes are globally coordinated by a *master process*; it makes sure that all processes are synchronized according to the T-Net spec, and implements the control commands issued by the developer. These commands allow for starting the emulation, pausing it, resuming it, placing or removing breakpoints, etc. For example, to emulate the T-Net shown in Figure 2, we require one master process and three component processes: one for Alice's node, one for Bob's node, and one for the existing channel. Next, we explain the role of each module to the overall emulation algorithm.

Master process (MP). The MP coordinates the overall emulation process by implementing the T-Net fire rule. The master implements the state diagram illustrated in the emulation phase of Figure 5. State Ready is the initial state. When the developer issues command start, the master sends an initialization instruction to the controller of every component process, resetting it to the initial transition set (i.e., marking the initial places), and the master enters state Running. From this point onwards, the master is notified by the component processes whenever transitions are enabled. If no breakpoints are set, the master instructs the corresponding component process to fire the transition. This procedure is repeated until the transitions reach the end state or until an error occurs. An error occurs if the component controller enables a transition that is not specified by the T-Net or if it sends a failure event (e.g., an exception has been generated by the component). If a failure has occurred, the master switches to state Failed, allowing the developer to issue commands to inspect the state of the system. If a breakpoint has been hit, the master starts a mechanism to pause the entire emulation. This consists of entering state Pausing and wait until all transitions pointed to by currently marked places have been enabled. (The set of places currently marked correspond to the current execution state of the system.) Once all of such transitions have been enabled, the master has paused the system in the current execution state and it changes state to Paused. Again, the developer can issue commands to inspect the current state. Resuming the execution is done by firing all the enabled transitions and changing state to Running. The developer can terminate the entire emulation at any point. Once paused, stepping consists of adding a breakpoint to a transition pointed to by the transition where the master has currently stopped, and resuming execution. The pause command consists of pausing the system in the current execution state.

Component controller (CC). The component controller notifies the master whenever a transition of its local component has become enabled and fires that transition when instructed by the master. The component controller detects when transitions are enabled by checking when all input places of a transition are marked. To detect when input places get marked, the controller receives events from the component logic.

Component logic (CL). The component logic is responsible for the emulation of the component-specific functionality associated with places. Such functionality includes, e.g., testing the user interface output for the OUT place, or sending a network join event for the JOIN place. The component logic receives instructions from the controller to mark a place, and notifies it once the place gets marked. Marking a place corresponds to executing the place's associated functionality successfully. If the execution has failed (e.g., because an exception has been triggered), the component logic notifies the controller of the failure.

4.4 T-Net Resource Management

When using public clouds to support T-Net emulation, for example Amazon EC2, developers must support the associated costs. This is a severe problem when debugging large scale networks. For that reason, Termite provides network I/O minimization through smart resource allocation. By using adequate resource allocation strategies, it is possible to co-locate emulators with high communication requirements. This allows network I/O minimization which reduces the cost of the emulation.

By analyzing the T-Net specification and, more particularly, the nodes and interactions described in it, Termite devel client computes which groups of nodes have high communication requirements and therefore should be deployed as close as possible (in the same physical host, for example).

The first step for determining a better resource allocation

strategy is to convert a T-Net into a weighted graph. To do so, each node and channel are mapped to graph vertexes. Then, each message between any node and channel is mapped to a graph edge. Repeated graph edges are reduced to one weighted graph edge (the edge weight equals the number of times the edge is used for exchanging messages). After this step, a weighted graph holds information about the network interactions between virtual emulators over time, i.e., some edges or vertexes may only exist for a period of time.

In the second step, the goal is to partition the weighted graph and split it into multiple sub-graphs. The attentive reader might notice two types of traffic between component processes: i) control traffic between every component process and the master process, and ii) application traffic between nodes and channels. Since application traffic always goes through a channel, partitioning can be done around channel vertexes. Therefore, each channel vertex belongs to a different partition and every other vertex goes to one of the existing partitions. The weight between nodes is used as criteria to choose the partition to where a node is sent.

For the final step, partitions created in the previous step are handed to the platform-specific connections, which, depending on the platform support, will enforce the partitions. For example, platforms such as OpenStack and CloudStack allow users to co-locate virtual machines on the same physical host and/or cluster. Thus, a good approach is to have one partition per physical host and/or network of hosts.

5. IMPLEMENTATION

We implemented a prototype of Termite in Java. Currently, it can emulate encounter networks of virtual devices that run Android $\geq$ 4.0 and communicate using WiFi Direct. We chose Android for its popularity and openness, and WiFi Direct for two reasons: when compared with Bluetooth, it allows devices to communicate over wider areas, and it is compatible with the standard WiFi network stack. Our prototype includes a T-Net scripting language to enable the specification of encounter networks by the developers.

The prototype adheres to the design guidelines introduced in Section 4. Some of its parts are platform-independent, namely the platform connectors, which are part of the admin and devel clients. The clients are console-based tools implemented in Python. Connectors use HTTP-based communication (REST) to reach cloud platform frontends.

Other parts of the prototype are specific to the platform currently emulated, i.e., Android. First, since the Android emulator does not currently emulate WiFi Direct, we implemented a library that interposes between the application and the Android system that emulates the relevant calls of Android WiFi Direct API (we do not extend our discussion on our WiFi Direct library because of space constraints). Second, to drive the execution of the application under testing on the virtual nodes, it was necessary to implement Android-specific *place enforcers*, i.e., place sensors and actuators. Lastly, we implemented a T-Net script compiler that translates scripts written in our T-Net scripting language into T-Net specs to be interpreted by Termite.

5.1 Android-Specific Place Enforcers

As presented in Section 4.3, place enforcers are responsible for implementing place marking events of a given T-Net spec. In particular, they control bootstrapping, GUI I/O, and networking events of the app running on an emulator instance, which represents a virtual node. Normally, such operations

```
1  tester ntweet = TestNTweet;
2  const m1 = "Connected",  m2 = "Disconnected",
3    m3 = "Hello, I'm Alice", m4 = "Hello, Alice!";
4  channel c = { z1[t=3], z2, z3, z4 };
5
6  node n1 = {
7    join(c1,z1)&
8    out(statpanel,[m1])&
9    in(inpanel,m3)&
10   {
11     out(outpanel,[m3])|
12     {
13       send(c1,z2)&
14       recv(c1,z3)
15     }
16   }&
17   out(outpanel,[m4])&
18   split(c1,z4)&
19   out(statpanel,[m2])
20 };
21 node n2 = {
22   join(c1,z1)&
23   {
24     out(statpanel,[m1])|
25     recv(c1,z2)
26   }&
27   in(inpanel,m4)&
28   {
29     out(outpanel,[m4])|
30     send(c1,z3)
31   }&
32   split(c1,z4)&
33   out(statpanel,[m2])
34 }
35
```

Figure 7: Script representing T-Net of Figure 2.

are highly dependent on the emulated mobile platform.

In our Termite prototype, we implemented place enforcers specific to Android applications. Essentially, they are placed in Java classes that are packaged along with the application. Some of these classes are included in the WiFi Direct emulation library, others are wrapper classes to the application code. The classes included in WiFi Direct emulation library detect the reception of join, leave, or synchronization events from the component controller and reflect those events to the application throught the emulated WiFi Direct API. The classes included in the wrapper classes use an instrumentation framework called Robotium [33] (similar to jUnit) that controls the bootstrapping of the application and interacts with the app's UI thread to emulate user input and test the app's output. The application is also packaged with a general component controller that interfaces with both wrapper and networking classes, and instructs them to fire or notify marking events. These requests are served by the place enforcers.

5.2 T-Net Specification

In order to emulate a given application, Termite requires a T-Net spec to be provided by the developer. The most obvious aspect of this specification is the T-Net graph itself (see Section 3.3). In addition, there is a second aspect of the T-Net spec that is platform-specific. In fact, in order to emulate the GUI I/O user events, the I/O place enforcers need to know how to interact with the application under test in order to perform input operations and test output values, e.g., which button to press when performing input, or which text panel to read when checking output. Since this interaction is application specific, the developer also has to indicate Termite how this is to be performed.

For a typical developer, specifying the T-Net graph could be complex and tedious. To alleviate this burden, we develop a Termite scripting language to help developers specify the T-Net graph using high-level constructs. Figure 7 shows an example of a Termite script that produces the T-Net graph for the NearTweet test case shown in Figure 2. The most relevant parts of the script concern the code blocks that specify the nodes' behavior: lines 6-20 for node A, and lines 21-34 for node B. Each block consists of a set of statementes (e.g., `join`, `out`, etc.) that must be executed sequentially (when separated by `&`) or in parallel (when separated by `|`). Statements can also contain nested blocks of more statements, for example in lines 10-16, indicating that this entire block must be performed between the `in` and `out` statements or lines 6 and 17, respectively. The GUI related statements (i.e., `in` and `out`)

```
1  public class TestNTweet extends termite.Test<NTweet>
2  {
3    public TestNeartweet() {
4      super(NTweet.class); super.setUp();
5    }
6
7    @Override
8    protected void guiIn(ID id, Param p) throws E {
9      super.actuatorGuiIn(p); Solo s = getSolo();
10     switch (id) {
11       case Id.inpanel:
12         EditText t = (EditText) s.getView(R.id.In);
13         s.clearEditText(t); s.enterText(t, p[0]);
14         s.clickOnButton("Send"); break;
15     }
16   }
17
18   @Override
19   protected void guiOut(ID id, Param p) throws E {
20     super.guiOut(p); Solo s = getSolo();
21     switch (id) {
22       case Id.statpanel:
23         TextView o = (TextView) s.getView(R.id.S);
24         assertEquals(p[0], o.getText().toString());
25         break;
26       case Id.outpanel:
27         TextView o = (TextView) s.getView(R.id.Out);
28         assertEquals(p[0], o.getText().toString());
29         break;
30     }
31   }
32 }
```

Figure 8: Java code of NearTweet tester class.

map naturally to the T-Net's GUI I/O places. The network related statements (i.e., `join`, `split`, `recv`, and `send`) map to network places established between the node and synchronization points of given a channel (e.g., `z1` of channel `c`). Regarding the channels, the developer only needs to name existing synchronization points (which map directly to the channel's T-Net transitions) and optionally specify a synchronization time (e.g., `z1` fires at $t = 3$ seconds). The Termite compiler can automatically generate the T-Net graph from this specification.

To properly parameterize the GUI I/O events of the Termite script, the developer has to implement a *tester* class (see Figure 8). This class, named `TestNearTweet`, overloads two methods `guiIn` and `guiOut` that specifies how a virtual user provides input to the application and test output from the application. For the NearTweet application, input is performed by typing text in the input panel and pressing a button (lines 11-14), and output validation requires testing either the status panel (lines 22-25) or the output panel (lines 26-29). Because the input and output places are parameterized by the T-Net graph, the tester methods receive an identifier (id for short) to differentiate in/out operations, and a parameter list of values to be used by the tester methods. Both the id and the parameter list are specified by the developer in the Termite script (see the GUI statements of Figure 7).

After specifying the Termite script and implementing the tester class, the developer: (i) runs the Termite script compiler, which generates the T-Net graph in a Java class, (ii) compiles the T-Net graph code, the tester class, and the application code, and (iii) creates a test package, which includes the generated binaries along with the WiFi Direct emulation library and the Termite wrapper code. The resulting package is then fed to Termite to be emulated on the system.

5.3 Platform Connectors

As presented in Section 4.1, Termite deploys virtual emulators in a wide variety of virtualization platforms. Termite relies on platform-specific connectors, i.e., software components that know the platform communication API. These software components work as plugins for the admin and devel clients. Each connector is implemented in an isolated Python module and is shipped with the client Command Line Interface (CLI). Adding more connectors does not require code changes to the client CLI, which searches for connectors (modules) on startup.

The job of a Termite connector consists on accepting commands from the admin and devel APIs (Termite API) and issue the adequate commands for the virtualization backend using a platform specific API (CloudStack REST API for example). From the cloud provider point-of-view, Termite clients are regular clients deploying virtual machines with customized EDIs. To implement a new connector, one has to implement both Termite APIs: i) the admin API which manages EDIs, account preferences, and instances, and ii) the client API accepts T-Net specifications, and all the commands related to orchestrating the emulation process.

6. EVALUATION

6.1 Evaluation Setup

To evaluate Termite, we use a real test deployment scenario: a local CloudStack installation comprising 50 nodes each one using an Intel(R) Core(TM) i5-3570 CPU @ 3.40GHz (4 cores with no hyperthreading) with 8GB of RAM. All nodes have Debian 7 with Linux 3.16 installed.

Physical nodes are connected through a two-layered switch network. Each of the leaf switches connect at most 20 nodes. A collector switch connects all leaf switches to other servers (CloudStack servers in particular) and to the Internet. All network connections have a limited capacity of 1Gbps.

In our deployment, CloudStack 4.4.1 was used. We installed one management server and one NFS server (serving as both primary and secondary storage). Both servers are in fact two virtual machines installed on one of our virtualization servers. The CloudStack management server uses 2 cores @ 2.4GHz, each with 4GB of RAM, while the NFS server uses 4 cores @ 2.4GHz also with 4GB of RAM. The available free space for virtual machine storage is 3,6 TB.

For emulating real Android devices, we used an Android x86 [17] 4.4-r2 image, a native Android system for x86 architecture (this is specially important since it enables us to use common virtualization systems such as KVM). Accordingly, we built a virtual machine template with 2GB of disk, 1GB of RAM and 1 core @ 1GHz. All virtual machines have a network IP in the same network. Isolation for the virtual machines' network is assured through virtual networks (VLANs).

6.2 Dataset Generation

Termite is targeted to help developers test their apps for encounter networks of different sizes and dynamics. To help with this task, we provide with Termite a dataset generator, i.e. a simple program that generates emulator location logs.

The dataset generator is a simple Python script which generates logs of emulators' location and creates or destroys groups (encounter networks) based on proximity. The script starts by spawning a predefined number of emulators in a predefined area. Then, for each round and for each emulator, it decides either to move or to stay. For each move, each emulator can walk at most 1 meter in any direction. When two or more emulators are close enough (a configurable parameter), a group creation is logged. On the other hand, when nodes are separated, the group termination is also logged.

Table 1: Termite generated datasets.

Metric	Dataset 1	Dataset 2	Dataset 3
Emulators	50	125	250
Groups (min)	10	13	16
Groups (avg)	12	16	16
Groups (max)	16	16	16
Group size (min)	2	3	12
Group size (avg)	3	7	15
Group size (max)	9	13	28
Neighbors (avg)	36	112	240

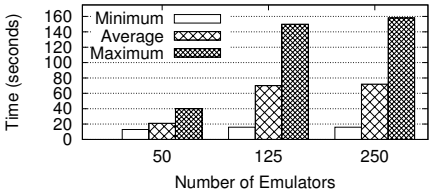

Figure 9: Emulator deployment time.

For evaluation purposes, we generate 3 datasets. Each dataset complies with the rules just described. To simulate a large in-door event (such as a conference, or a festival) we use a fixed 100x100 meter area where emulators are spawned. The minimum distance of interaction is 32 meters in every direction (32 meters is a reasonable value according to the fixed area dimension to enable both small and large groups of devices). Therefore, as soon as two emulators are close (32 meters or less) a group is created. For each dataset, each emulator can move up to 100 times (this is configurable). These datasets can be based on real device traces.

Table 1 shows some statistics about the generated datasets: i) the number of emulators, ii) the number of groups (minimum, average, and maximum), iii) group size (minimum, average, and maximum), and iv) the average number of neighbors at each moment(i.e., number of nodes with at least one neighbor). Based on these datasets, T-Net scripts are generated. These scripts conduct the encounter network emulation, which is described next.

6.3 T-Net Deployment

Emulation deployment, starting emulators and deploying the application, is one of the main steps in deploying a T-Net simulation. Therefore, in this section, we analyze both operations and try to point out how the system scales and its bottlenecks.

Starting emulators is the most costly operation. For each emulator, a 2GB disk file must be fetched from the NFS server and cloned. After cloning the disk, we still have to wait until the virtual machine boots and Android starts. On the other hand, application deployment is faster: the application image is uploaded to the device which installs it and then starts it. For each of these operations, we used 15 parallel requests to speed-up the process.

Figure 9 shows the performance results for creating new virtual devices. This data was acquired as follows: i) 15 parallel requests to create a new emulator are launched, ii) for each request, the start and finish time is recorded, iii) as soon as a request finishes, a new one is launched. The number of parallel requests is configurable and it is only limited by local resource capacity (mainly network and CPU). The default amount of parallel requests, 15, was enough to consume 100% capacity of one CPU @ 1GHz. Figure 9 shows the minimum, average, and the maximum values that we measure through several runs.

It is important to notice that the time needed to deploy a new emulator is not constant. Thus, as the number of requested emulators increases, the average time to deploy an emulator also increases. We found out that two factors contributed for this to happen: i) physical hosts start to be overloaded (with more virtual resources than those physically available), and ii) saturated network bandwidth (our infrastructure supports at most 1 Mbps). Through our analysis, network bandwidth is the most limiting factor; Figure 10 confirms it. For deployments with more than 50 emulators, network bandwidth is a serious bottleneck, and even limiting the number of parallel requests to 15, our experiment shows that CloudStack continues to use network bandwidth on the background, for each newly created virtual machine. It is worth to note that our Android image does not perform any network I/O after booting.

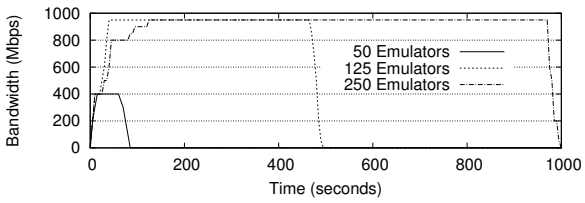

Figure 10: Deployment network bandwidth.

Table 2: Application deployment time.

Metric	Dataset 1	Dataset 2	Dataset 3
Emulators	50	125	250
Install App (min)	6 sec	6 sec	6 sec
Install App (avg)	7,6 sec	7,4 sec	7,6 sec
Install App (max)	8 sec	8 sec	8 sec
Total	27 sec	69 sec	131 sec

The installation step is faster than deploying an emulator. We had, however, some problems with the Android Debug Bridge (adb), which is the pre-defined tool to handle Android emulator management. When installing an application on multiple emulators, both in parallel or in serial, the adb tool crashes too often. Since adb uses a small server daemon that keeps some state, using it with many emulators, according to our experience, results in frequent crashes.

To solve this problem, we installed a SSH server, which enables both Secure Shell Login and Secure Copy Protocol commands. Thus, when installing an application, we upload the APK file (application package) to the emulator (using SCP) and then use SSH to invoke local commands (on the emulator) to install and start the application.

According to Table 2, the time to install and launch an application does not depend, individually, on the number of emulators, i.e., the time it takes to deploy it on one emulator does not produce any consequence on other deployments on other emulators. On the other hand, the total amount of time to deploy the application on all emulator depends, as expected, on the number of emulators. Once again, we used 15 parallel requests to speed up the process.

With our experiments, we conclude that deploying an application is fast. This is particularly important since test and debug operations often require many application deployments. It is clear, however, that the size of the APK file may increase or decrease the amount time needed to deploy de application. In our tests, we used an APK file with 1MB.

Figure 11 presents the performance evaluation for deploying each of the presented datasets' T-Net (starting all emulators and installing the application). For each dataset, with 50, 125, and 250 emulators, the minimum, average, and maximum

Table 3: Application execution statistics.

Metric	Dataset 1	Dataset 2	Dataset 3
Emulators	50	125	250
Number of Neighbors (avg)	36	112	240
Number of Messages (avg)	686	6172	28652
Time (avg)	117 sec	189 sec	266 sec

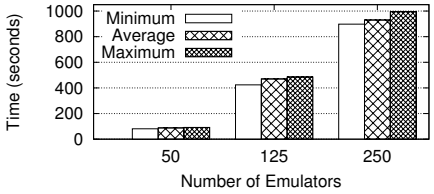

Figure 11: T-Net deployment time.

recorded values are presented. As the number of requested emulators increases, the time to prepare all emulators increases linearly. Subsequent application installations would take less time (see Table 2).

6.4 Application Execution

To test our Termite deployment, we developed a simple yet representative application inspired on NearTweet (see Section 5.2). Using the previously described datasets, devices navigate in a two-dimensional space. Whenever two or more nodes are close, each node periodically sends messages to its neighbors; a node can send up to one message to each neighbor at a time.

The design of the dataset and application has a major impact on the time it takes to conduct the simulation. Nevertheless, we show the performance results from running our prototype application to demonstrate the small overhead associated with Termite.

Table 3 shows several results from running our prototype application. Although the duration of all three datasets is 100 steps, as the number of the emulators grow, the number of messages in the system, per step, also grows. Sending and receiving more messages messages produces more accumulated overhead over time. Moreover, synchronization between emulators (waiting for messages to arrive for example) also introduces more overhead in the system.

6.5 Resource Allocation

As described in Section 4.4, Termite allows platform-specific connectors to take advantage of communication patterns knowledge. Therefore, in our local Termite deployment, we use this knowledge to reduce the amount of inter-host message, reducing the network overhead of emulating a T-Net.

Results are shown in Figure 12 (which presents the number of exchanged messages for both assisted (strategy described in Section 4.4) and random strategies for 50, 125, and 250 emulators). For intra-host messages, i.e., messages between emulators that reside in the same physical host, it is possible to see that assisted resource allocation performs much better (increases the amount of local messages) than a random approach, which is the default approach in CloudStack. Regarding inter-host messages, i.e., messages between emulators in different physical hosts, assisted resource allocation allows Termite to reduce the number of inter-host messages (compared to a random placement).

According to the results, better resource allocation strategies are able to reduce the amount of inter-host messages in 88%, 84%, and 53%, for datasets 1,2, and 3, respectively. The two first results are very good since Termite is able to fit the

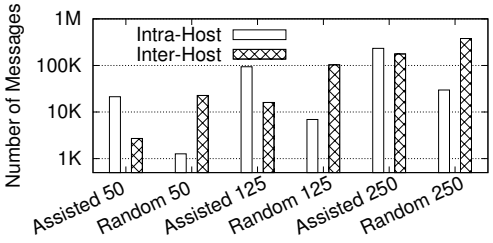

Figure 12: Assisted vs. Random resource allocation.

whole emulator group in the same host. The result for 250 emulators is not as good since the groups' dimension does not fit on the same host anymore.

7. RELATED WORK

Developing and testing of distributed mobile encounter based applications is a difficult task which can be done by: i) relying on real devices, ii) using a simulation environment, or iii) using emulation. As already mentioned, the first two approaches show lack of expressiveness, complex or no full execution control, and ensuring reproducibility is hard; all these are important aspects as a developer needs to mimic the movement of devices and simulate user interaction in a realistic encounter network (either using real devices or software emulators). In other words, solutions based on simulation suffer from lack of fidelity when compared to real world settings and real-word settings are also not a good approach given their high complexity, cost and availability (e.g. real mobile devices may not implement the functionalities required). For this reason, in this section we focus on emulator-based approaches; thus, not considering simulators and real world settings (as is the case of ns2[14], ClickRouter [26], ModelNet [6], and others).

There are several fundamental characteristics along which emulator-based solutions can be analyzed: i) emulator capability to support encounter network topology resulting from users mobility and users input, ii) focus on low-level routing protocols (with no support for end-user applications), and iii) support for several users/topologies in the same hosting infrastructure. In the remaining of this section, we address relevant related work taking into account these characteristics.

vBET [23] is an emulation testbed that uses virtual machine technology to emulate entities such as routers, switches, firewalls, etc. This system is focused on emulating network protocols and topologies such as routing, distributed firewalls and P2P networks. It fails to address the issues specific to encounter networks and the corresponding mobile applications, as it does not provide support for dynamic networks that evolve according to user movement and/or input. Other similar systems such as PlanetLab [8], Quagga [10], Emulab [2], [27] all have the same drawbacks, i.e. they do not support the easy and wide deployment of arbitrary dynamic network topology which has to be considered when addressing the development and testing of encounter-network applications.

Another interesting emulator system is SplayNet [34], which focuses on supporting the evaluation of networked applications. Once again, this system does not address the specific needs of encounter networks as it assumes that network topologies, while several are supported with diverse end-to-end characteristics such as bandwidth, packet loss, etc., do not change depending on users movements and/or input as it happens with Termite.

ReactiveML [11] and VANS [35] (among many similar oth-

ers) are emulator systems that take into account the movement of nodes in the creation of ad-hoc networks. However, they tend to ignore a crucial aspect when developing mobile encounter applications, namely user input/interaction (e.g. touchscreen I/O).

On the other hand, SmartDroid [37], takes into account such user I/O. However, this work focus on addressing the particular issue of revealing UI-based trigger conditions of sensitive behaviors in Android applications. It uses a hybrid static and dynamic analysis method to reveal UI-based trigger conditions in Android applications. Thus, its functionality does not match the one provided by our Termite system.

Regarding mobile test automation, there are several existing solutions besides Robotium. Sikuli [36] accesses the device screen to take snapshots that will be matched against expected snapshots. Therefore, testing with many devices will produce a lot of snapshots that will result in expensive computation to calculate similarities. Moreover, Sikuli needs to access the screen of every device periodically. Besides being network consuming, it is very slow. MonkeyRunner [29] follows a similar approach to Sikuli and presents a low-level API which hampers the creation of tests. Robotium (introduced in Section 5.1), on the other hand, provides distributed testing capability, i.e., tests are performed directly on the device. This enables large performance improvements regarding the other two solutions.

Android x86 emulator is used to provide a versatile approach, allowing the developer to manage the Android image freely, and efficient virtualization with close to native performance. Approaches such as the default emulator (deployed with the Android SDK) or Genymotion [20] also present a viable solution for local testing but are impossible to use with other virtualization infrastructure because of the specific EDI used by these emulators. Manymo [28] offers a web-based approach where devices are launched in a private cloud and only the interface is shown through the browser. Despite the limited versatility of this solution (where developers cannot customize their EDI), this solution provides no easy way to automate tasks through the browser.

To the best of our knowledge, Termite is the first emulator environment that, while supporting researchers to deploy several network topologies at the same time on the same physical nodes over a shared platform (e.g. a cluster), also takes into account the inevitable and fundamental topology changes resulting from user join and leave operations as well as user I/O interactions allowing the testing to be reproduceable.

8. CONCLUSION

This paper presents Termite, a distributed testbed for testing and debugging mobile applications. Specifically, Termite targets scenarios of encounter networks, where devices can opportunistically form wirelessly connected groups, and co-located users can interact with each other. Termite provides a software emulation service that allows for the specification of complex and dynamic encounter networks, including the simulation of user input. Our current prototype, running on a university cloud infrastructure, is being used but approximately 150 users. It supports Android applications and WiFi Direct emulation and our evaluation results confirm the performance and scalability of our system, therefore confirming the contributions of this work.

Acknowledgments. This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013, project TRACE (co-financed by the European Commission through contract n^o 635266), and project PCAS (co-financed by the European Commission through contract n^o 610713).

9. REFERENCES

[1] Android sdk. http://developer.android.com/sdk/index.html?hl=sk.
[2] Emulab - Network Emulation Testbed Home. http://www.emulab.net.
[3] Haggle. http://www.cl.cam.ac.uk/research/srg/netos/haggle.
[4] ios dev center. https://developer.apple.com/devcenter/ios/index.action.
[5] Iron Cube. https://play.google.com/store/apps /details?id=com.glow3d.ironcube.
[6] ModelNet. http://modelnet.ucsd.edu.
[7] Peerhood. http://www2.it.lut.fi/project/ptd/peerhood.html.
[8] PlanetLab. http://www.planet-lab.org.
[9] Proxidating. http://www.proxidating.com.
[10] Quagga Routing Suite. http://www.nongnu.org/quagga.
[11] ReactiveML. http://rml.lri.fr.
[12] SHAREit. www.lenovo.com/shareit.
[13] Speck. http://speck.randomfoo.net.
[14] The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns.
[15] Wifi shoot! https://play.google.com/store/apps /details?id=com.budius.WiFiShoot.
[16] O. Akribopoulos, M. Logaras, N. Vasilakis, P. Kokkinos, G. Mylonas, I. Chatzigiannakis, and P. Spirakis. Developing Multiplayer Pervasive Games and Networked Interactive Installations Using Ad Hoc Mobile Sensor Nets. In *Proc. of ACE*, 2009.
[17] Android x86. http://www.android-x86.org/.
[18] C. Boldrini, M. Conti, F. Delmastro, and A. Passarella. Context- and Social-aware Middleware for Opportunistic Networks. *J. Netw. Comput. Appl.*, 33(5), Sept. 2010.
[19] M. Conti and M. Kumar. Opportunities in Opportunistic Computing. *Computer*, 43(1), Jan. 2010.
[20] Genymotion. https://www.genymotion.com/.
[21] E. Huang, W. Hu, J. Crowcroft, and I. Wassell. Towards Commercial Mobile Ad Hoc Network Applications: A Radio Dispatch System. In *Proc of MobiHoc*, 2005.
[22] N. Jabeur, S. Zeadally, and B. Sayed. Mobile social networking applications. *Commun. ACM*, 56(3):71–79, Mar. 2013.
[23] X. Jiang and D. Xu. vBET: A VM-based Emulation Testbed. In *Proc. of SIGCOMM MoMeTools*, 2003.
[24] C. Journal. Mobile ad hoc networking revamps military communications. http://www.cotsjournalonline.com/articles/view/102158.
[25] D. N. Kalofonos, Z. Antoniou, F. D. Reynolds, M. Van-Kleek, J. Strauss, and P. Wisner. MyNet: A Platform for Secure P2P Personal and Social Networking Services. *IEEE PerCom*, 0:135–146, 2008.
[26] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.
[27] Y. Li, P. Hui, D. Jin, and S. Chen. Delay-tolerant network protocol testing and evaluation. *Communications Magazine, IEEE*, 53(1):258–266, 2015.
[28] Manymo. https://www.manymo.com/.
[29] MonkeyRunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html.
[30] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proc. of the IEEE*, 77(4):541–580, 1989.
[31] NIST. Mobile ad hoc networks. http://www.antd.nist.gov/wahn_mahn.shtml.
[32] A.-K. Pietiläinen, E. Oliver, J. LeBrun, G. Varghese, and C. Diot. MobiClique: Middleware for Mobile Social Networking. In *Proc. of WOSN*, 2009.
[33] Robotium. http://code.google.com/p/robotium/.
[34] V. Schiavoni, E. Riviere, and P. Felber. SplayNet: Distributed User-Space Topology Emulation. In *Proc. of Middleware*, 2013.
[35] M. Shinohara, H. Hayashi, T. Hara, A. Kanzaki, and S. Nishio. VANS: Visual Ad hoc Network Simulator. In *Proc. of ICMU*, 2005.
[36] T. Yeh, T.-H. Chang, and R. C. Miller. Sikuli: using gui screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 183–192. ACM, 2009.
[37] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *Proc. of ACM SPSM*, 2012.