Similitude: Interfacing a traffic simulator and network simulator with emulated Android clients

Seth N. Hetu
Dept. of EECS
Massachusetts Institute of Technology
Cambridge, MA 02139
Email: sethhetu@csail.mit.edu

Vahid Saber Hamishagi
Future Urban Mobility
SMART
Singapore, 138602
Email: vahid@smart.mit.edu

Li-Shiuan Peh
Dept. of EECS
Massachusetts Institute of Technology
Cambridge, MA 02139
Email: peh@csail.mit.edu

***Abstract*—Mobile phone apps are increasingly part and parcel of today's intelligent transportation systems (ITS). Evaluating these apps at scale requires modeling of phones and networks, along with vehicles, people and roads. In this paper, we present Similitude, a system comprising a traffic simulator, network simulator, and cluster of Android emulators that has applications in mobile app development as well as modern transport simulation. Apps with their wireless network stack are run on an Android emulator, with network packet delivery modeled in detail via a network simulator. Each phone's location and human interaction elements are obtained through interfacing with a microscopic traffic simulator running driver and pedestrian behavioral models. A prototype of the system is shown to scale well up to 300 simultaneous connected Android emulators, with individual system components scaling upwards of thousands of agents. An ITS app that does road space rationing is used as the case study demonstrating a potential use case of Similitude.**

I. Introduction

Modern mobile phone application (app) developers are increasingly able to rely on GPS location services to develop rich client experiences. At the same time, transport system designers have used mobile apps for a variety of ITS purposes, such as traffic routing[1] and bus prediction[2]. Both parties can benefit from computer simulation: app developers can test their applications at a city scale with credible location and network data, and calibrated models of drivers, passengers and pedestrians. Similarly, transport system designers can estimate the impact of phone apps on transport system parameters such as roadway capacity and travel time improvements.

With this in mind, we present Similitude[3], a system that combines the SimMobility agent-based microscopic traffic simulator with Android virtual machine emulation in QEMU and ns-3, a network simulator. This paper deals with the technical details of the combined system's construction, as well as the performance, scalability, and limitations of the system and a case study of its use with the RoadRunner app, a distributed road-space rationing program compatible with Android [1].

Similitude's distinguishing quality is its focus on valid simulation combined with flexible emulation. Consider the developer of a GPS guidance app who wants to add route guidance based on roadway congestion. Similitude can model the sending of guidance information over LTE, as well as the driver's response to new route information. This in turn will affect other drivers' behaviors, which can have a cascading effect as the size of the study area grows. With Similitude, the app can be modified, tested, and then shipped with accurate knowledge of its expected effect on both individual drivers and the road network at large. Once apps are released to users, local transport authorities may wish to perform additional studies, such as determining the quality-of-service repercussions of upgrading from 3G to LTE or exploring the potential for vehicle-to-vehicle collaboration through local WiFi. Similitude can switch the simulated network protocol easily, and unlike traditional simulation it does not require fully re-implementing the apps' logic in a simulator module.

The remainder of the paper is organized as follows. Section II covers related work. Section III describes the system design and its implementation for a specific case study. Section IV provides a performance and scalability analysis of the system and its components. Finally, Section V concludes the paper.

II. Related Work

Our work draws from existing research in vehicle and transit simulation, wireless and network simulation, massive mobile phone emulation, as well as the combination of simulators with emulators. Clearly, one of our requirements when selecting component simulators is that they are open-source, enabling the integration necessary for Similitude. We will now cover related work in each of these areas in turn

Traffic simulation. Our work requires an agent-based, microscopic traffic simulator. The former refers to a technique of encapsulating driver behavior into independent units called agents, which provides the flexibility necessary for transparent integration with Similitude. The latter refers to the typically short update times of individual simulation components, which is necessary for accurately representing network response time data. In addition, the simulator used must be scalable. SUMO is one such simulator, well-known for its rich tool set and extensible API, written in the Java programming language [2]. Similarly, MATSim is a fast, agent-based traffic simulator that is open source [3]. However, as it is mesoscopic, its time resolution is not suitable for our purpose. The simulator we chose for Similitude integration was SimMobility, a new

[1]http://googlemobile.blogspot.com/2011/03/youve-got-better-things-to-do-than-wait.html

[2]http://www.sbstransit.com.sg/iRIS/overview.aspx

[3]The name draws from **sim**ulation of a multi**tude** of people, as well as having veri**similitude** due to relying on validated simulation models.

978-1-4799-4449-1/14/$31.00 ©2014 IEEE

microscopic traffic simulator based on MITSIMLab [4] which incorporates agent-based simulation and multi-modal route choice [5]. In addition to being an agent-based simulator, SimMobility was written with multi-threaded performance in mind, and runs at a time resolution of 100 ms increments.

Network simulation. Regarding network simulators, we required a scalable, open source simulator that could model a variety of networking standards. After careful consideration, we chose to use ns-3, due to its modular structure, wide range of supported protocols, and reasonable performance [6]. The field of network simulation is well-established, so many other network simulators would also be suitable, such as OpNet[7], OmNet++[8], and others.

Network + Traffic simulation. Tellingly, many new studies focus on integration rather than extension. NCTUns is one such simulator; it allows the running of apps to generate real network traffic. Only GNU/Linux apps are supported, but the simulator makes use of the actual operating system's TCP/IP protocol stack, as a short circuit around traditional network simulation [9]. Similarly, [10] uses a coupled traffic simulator (SUMO) and network simulator (JiST/SWANS) to evaluate inter-vehicle communication techniques as well as providing a hybrid real-world test bed. Similitude distinguishes itself from these systems by focusing on mobile phone application emulation; this allows developers to capture more detailed emergent phenomena such as congestion mitigation via distributed road-space rationing.

Phone emulation. We require an open source emulator that can multiplex Android operating system instances per server node. Simulating large numbers of mobile phones has received some attention recently, mostly in the interest of quality assurance testing. [11] performs automated analysis of sensitive data leakage in Android apps. The authors mention running multiple apps in parallel, but do not explore the performance implications of parallel execution. [12] forgoes parallelization altogether, gaining performance by switching between Xen virtualization and full-system emulation based on certain criteria. Although novel, this approach is not suitable for Similitude, as we require the accuracy of full-system emulation for all agent interactions. By far the most prominent work on emulator scalability is the MegaDroid project at Sandia National Laboratories. MegaDroid provides a simple interface to manage large numbers of QEMU snapshot-based instances, along with supplemental tools to aid setting up a software-defined network that it can communicate with from any single node. Initial tests with MegaDroid scaled up to 300,000 simultaneous Android instances [13]. We use an open source, minimalist variant of MegaDroid called "minimega" that was developed by the same authors.

Finally, there is some existing work that attempts to unify simulators and emulators. An example is [14], which pairs the S3FNet network simulator with a version of the OpenVZ emulator modified to support virtualized time; in addition, timestamps on network packets are modified to ensure consistency. The goal is to study distributed attacks on smart (networked) power meters. This approach is limited by the simple nature of power meter applications; Android apps are far more complicated and cannot be controlled solely through network traffic manipulation. In addition, the relationship between simulator and emulator is tightly coupled such that adding in another simulator (in our case, a traffic simulator) is not a clear-cut process. Another approach sometimes used is called "hardware-in-the-loop" simulation, in which physical units are integrated into a running simulation [15]. Although prohibitive for large-scale Android deployments, the ability to replace several emulators with actual devices has merit for comparative studies; thus, Similitude supports physical devices in addition to emulators.

TABLE I. COMPONENT FEATURE SPACE

Component	*Primary Use*	*Additional Features*
SimMobility	• Driver locations • Coordinates human behavior and app interactions	• Cars, buses, pedestrians • Behavior models: route choice, car following, etc. • Scales to city-sized populations
QEMU	• Emulates Android	• VM snapshots • Hardware-assisted virtualization • Runs Android APKs as-is
minimega	• Start, stop, control VMs in batches • Automates networking and VLANs	• Auto-discovery of minimega instances • Seamless interaction with dnsmasq, Open vSwitch • Remote viewing through noVNC
ns-3	• Network data movement	• Supports Wi-Fi, WiMAX, LTE, 802.11p • Modular, extensible

III. SYSTEM DESIGN AND IMPLEMENTATION

A. Components and Organization

Similitude comprises three primary interacting components: the SimMobility short-term simulator, the Android emulators, and the network simulator ns-3. The SimMobility short-term simulator was used to centralize all additional communication, and acted as a server both to the emulators and ns-3. Android devices were emulated using QEMU, a well-known open-source machine emulator that gains added performance from virtualization technologies such as Intel VT and AMD-V. Each device was a snapshot of Android 4.2 (Jelly Bean) from the Android-x86 project[4]. The minimega management tool was used to provide easier management of virtual machines and networking between them. Minimega's primary feature is the ability to push commands to ranges of virtual machines with minimal required configuration. Finally, ns-3 was optionally used to provide accurate network performance estimates for all agent-to-agent messages. If ns-3 was omitted, SimMobility would simply forward messages to their destination agents as soon as they were received from the sender. Table I lists these components and any additional relevant features.

Connecting SimMobility to the Android instances was straightforward. First, a proxy agent was created in SimMobility for each emulated Android instance. These proxies expose enough external information for any other SimMobility agent to interact with them seamlessly. In addition, the proxies synchronize state with the Android clients every time step, ensuring that the location of each agent (and all other properties) are always up-to-date. Large numbers of clients are problematic for certain cluster configurations due to the hard limit on the maximum number of simultaneously open files in Linux (1024). To work around this, we provide a simple relay script which multiplexes messages between clients on the same cluster node, allowing our system to scale past this limit.

[4]http://www.android-x86.org

Once connected, Android instances proceed in lock-step through the entire simulation run. Unlike some techniques, we require minimal changes to the Android application itself in order to ensure valid results. This typically performs better than adding time synchronization code to existing emulators, and allows us to maintain a generic interface —real-world phones running the same app binary can interface with Similitude. The extent of these changes is described in the following section. Generally speaking, inter-app communication need not be modified, as SimMobility will simply relay these messages through ns-3.

B. Android App Modifications

All apps require some rudimentary modifications to work with Similitude. To ease integration efforts, we provide a simmobility4android (sm4and) Java library, which is outlined in Figure 1. Integration proceeds as follows. First, all timing functions must be replaced with the software timer equivalent in sm4and. For example, we call `SimMobilityBroker.postDelayed` instead of `Handler.postDelayed` to post a message after a time offset; this correctly uses the software timer, which is kept in sync with SimMobility as described previously. Second, one must use the sm4and location updates instead of the typical Android `LocationManager`. All code that requests the current location or location update notifications must do so through the `SimMobilityBroker`. Third, some apps will have a small number of custom messages required for their operation. For example, an app tracking carbon emissions may request the grade of the current roadway, which is not normally sent to emulated entities. The sm4and library provides a simple interface for defining new message types, and SimMobility itself is extensible on the server side. Finally, all apps will require some custom workarounds for inconsistencies that arise during emulation. For example, the `eth0` network interface that Android-x86 instantiates takes longer than expected to acquire a DHCP lease. The app used in our case study was modified to avoid connecting to the SimMobility server until a short, fixed delay had elapsed; such a solution may be generally applicable and incorporated into sm4and in the future. At present, the nature of these modifications is such that they must be done manually. However, many of them may be automated in future releases of sm4and, as we have intentionally kept API calls parallel to Android's when possible.

IV. Performance Analysis

We performed several tests of the individual components in order to determine their scalability. Following that, we profiled the combined system's performance, and performed an analysis of the types of setups that scaled the most effectively. We end off with a case study of an ITS app.

A. Experiment Setup

All tests were performed on a randomized virtual road network covering an area 6km by 2.5km. This road network was generated using the SUMO[5] abstract network generator, with the following command:

```
netgenerate --rand -o out.xml
  --rand.iterations=200 --random -L 2
```

In studies featuring driver agents, one of two sample sets was used. The *random* sample set was used to stress the simulators (SimMobility and ns-3) in isolation. It contained up to 100,000 randomly selected agents from the list of all possible origin to destination pairs, with subsequent agents on the same origin offset by 1s to prevent overlap. The *longest* sample set was used to test setups with Android emulators. It contained the four longest origin/destionation trips (diagonals) repeated and offset by 1s to prevent overlap, up to a total of 300 agents. This was designed to keep as many connected clients in the system for as long as possible; the *random* sample set featured a number of short trips that would artificially limit the maximum simultaneously connected clients.

Experiments were carried out on a cluster of Intel Xeon X5650 nodes running Ubuntu 13.10 (Linux kernel 3.11). Each node contained 2 sockets with 6 physical CPU cores each at 2.67 GHz and 48 GB of RAM. Hyperthreading was enabled, leading to 2 logical cores per physical, or a total of 24 processors per node. Nodes existed on the same switch and exhibited low latencies; typical ping values were $<$5 ms. When used, SimMobility was compiled in Release mode with agent update output disabled. Where Android interactivity was required, we used RoadRunner, a distributed road-space rationing application, as our representative application. RoadRunner is designed to eliminate traffic bottlenecks by "rationing" commonly congested areas through the use of tokens. Tokens are exchanged via phone-to-phone communication (802.11p), which falls back to LTE requests to a cloud server if no clients are in range. Any vehicle that does not accrue the requisite tokens for a specific rationed area (a "Region" in RoadRunner's parlance) must re-route or incur a penalty token. By restricting the initial token set, traffic planners can disincentivise overcrowding and improve average vehicle speed. RoadRunner has been tested in limited real-world deployments; its need for detailed traffic and communication information at runtime makes it an ideal candidate application.

B. Individual Components

The performance of each of the simulators (SimMobility and ns-3) in isolation was stressed with a number of agents from the *random* sample set. Samples of 1,000 up to 100,000 agents were chosen and run for 30 minutes of simulation time. The total elapsed time was measured using the Linux "time" command. For each scenario, the simulation was run five times in succession. Traces from SimMobility were used to generate demand in ns-3.

The average elapsed times are shown in Figure 2. Given ns-3's poor scalability, its results above 20,000 connected agents are not shown. For the sake of comparison, ns-3 took 10 hours to complete the 100,000 agent simulation. With the exception of low agent counts, SimMobility outperformed ns-3 in single-threaded mode, and improved in performance as the number of threads increased. The poor performance of ns-3 is compounded by its inability to be run on more than 1 thread. Although work such as [16] has attempted to address this, it might be worthwhile in the future for Similitude to switch to a simulator with multi-threaded performance as a primary feature.

[5]Simulation of Urban MObility, http://sumo-sim.org

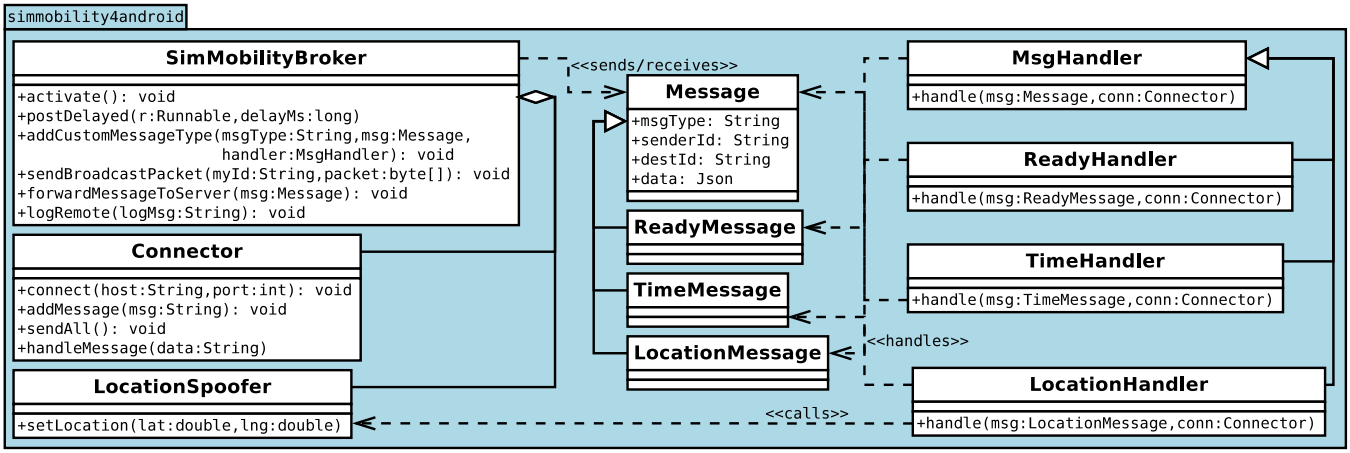

Fig. 1. UML diagram of sm4and library

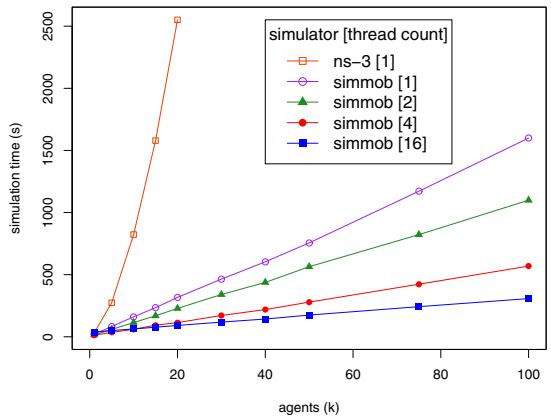

Fig. 2. Performance of SimMobility and ns-3

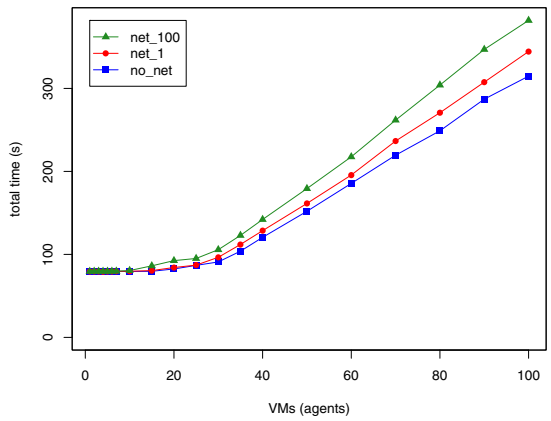

Fig. 3. Performance of Android emulators

The scalability of the QEMU instances themselves was estimated with agents from the *longest* sample set. For each agent, a full trace of messages received was re-run through the Android application in offline mode. All agents were located on the same node, and the agent count was increased to stress the system. Following this, the same scenario was re-run, but with a trace of the sent messages pushed to another node over TCP. No synchronization or communication took place; the remote server simply read and then dropped every message, and the local Android instances sourced the replies from the full message trace rather than the server. These two scenarios —called *no_net* and *net_1* respectively— were designed to test the scalability of the QEMU instances and the clients in general rather than the absolute performance. A third scenario, *net_100*, multiplied the amount of data sent by each agent each time tick by 100. This was intended to approximate the increased amount of communication that larger agent populations would typically generate.

Figure 3 depicts these results. All three sample sets incur minimal overhead for each additional emulator up to 30, at which point each new emulator incurs a fixed additional cost. The highest agent count (100) is far above the node's core count (24), indicating that QEMU, as configured by minimega, scales quite well. The cost of *net_1* over *no_net* is roughly the same as the cost of *net_100* over *net_1*, although such differences start to exacerbate as large numbers of agents compete for network hardware. Thus, we observe that the ideal number of agents (VMs) per core should not exceed 1.5 to 2.0 times the number of cores.

C. Combined System

The performance of the combined system was measured across a variety of agent and node counts. Up to 288 agents were chosen from the *longest* sample set; the simulation was run until all agents exited (approximately 9 minutes). In order to remain as consistent as possible, the Linux nanoscale timer was used to trim the first and last 100s of simulation time from the total execution time.

Figures 4 and 5 depict the results with ns-3 disabled and enabled, respectively. In both cases, as the total number of agents in the system increases, better performance is achieved by increasing the number of nodes. The time saved by using multiple nodes is partly offset by the cost of communicating with a single SimMobility broker. For agent counts less than

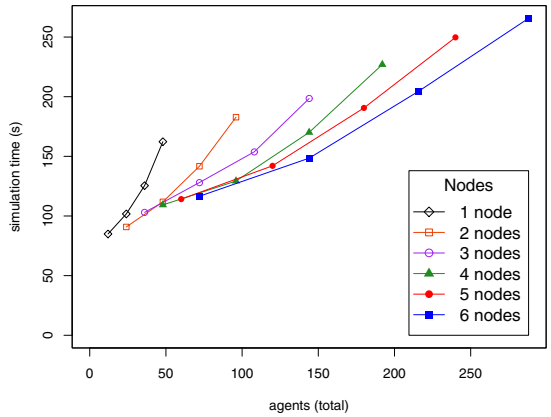

Fig. 4. Scalability (Android + SimMobility)

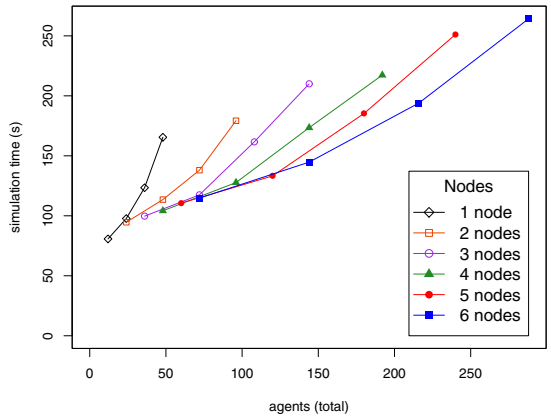

Fig. 5. Scalability (Android + SimMobility + ns-3)

300, ns-3 support does not negatively impact similitude's performance. From our earlier profile of ns-3 alone, we expect ns-3 to become the primary performance bottleneck for deployments greater than 1000 agents. At some point, multiplexing too many agents per node will cause the performance to drop, apparent in the non-linear sloping of each line in Figure 4. This limit is roughly equal to the processor count, or 24 agents-per-node. Figure 5 indicates that ns-3 has no significant effect on total execution time.

Expanding on this, Table II presents the simulation length divided by the total number of agents in the simulation, to arrive at the cost per agent in each scenario. The lowest cost is achieved with 6 cores and 48 agents per node.

TABLE II. COST PER AGENT (S)

agents per node	nodes					
	1	2	3	4	5	6
12	7.08	3.78	2.86	2.28	1.90	1.62
24	4.24	2.33	1.78	1.35	1.18	1.03
36	3.48	1.97	1.42	1.18	1.06	0.95
48	3.38	1.90	1.38	1.18	1.04	0.92

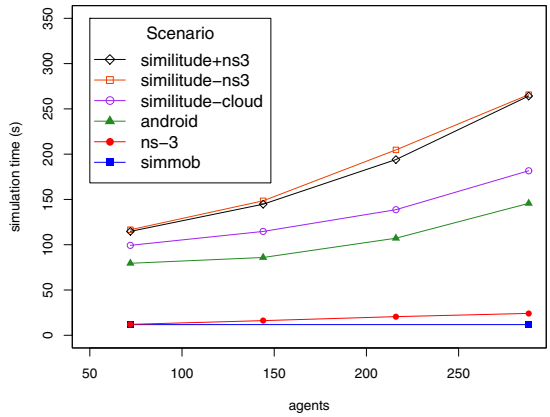

Fig. 6. Best-Case Performance Comparison

Finally, Figure 6 puts the relative costs of each component and combination in perspective. The cost of SimMobility alone (*simmob*) and ns-3 alone (*ns-3*) is minimal and does not vary much for small agent counts. The cost of Android alone (*android*) was calculated by dividing the agent count by the total number of available nodes (6) and then interpolating from the two nearest data points in *net_1*. It scales well, and scalability can easily be improved by adding more nodes. Android combined with SimMobility but not ns-3 (*similitude-ns3*) incurs a higher cost as more agents are simulated. The main reason for this degradation, as discussed earlier, is the increased time spent synchronizing messages sent from SimMobility to the Android clients. The cost of all three components (*similitude+ns3*) does not differ significantly from the cost with ns-3 disabled. This is also likely susceptible to some amount of false synchronization. Finally, the *similitude-ns3* experiments were re-run with cloud token requests disabled (*similitude-cloud*). This approximates the performance gain of rewriting external cloud applications as Sim Mobility modules. Similitude also supports using existing cloud applications as-is, but this introduces significant overhead. Cloud applications are typically small and easy to encapsulate as Sim Mobility modules; thus, *similitude-cloud* represents a realistic best-case performance of similitude.

D. Performance versus Physical Phones

An attempt was made to put the performance of similitude in context with respect to physical Android devices. Ten Samsung Galaxy Notes (1.5 GHz dual-core Snapdragon S3 with 1GB RAM) and ten Samsung Galaxy S4s (1.9 GHz quad-core Krait 300 with 2GB RAM) were run in several configurations with network access provided either over local Wi-Fi or through USB reverse tethering. Figure 7 depicts the results. A *baseline* was obtained by running 10 virtualized clients on a single Intel Xeon X5650 node (which can comfortably support more than 20 VMs without performance impact as shown in Figure 3). These clients suffered a performance penalty from virtualization, but had the benefit of access to the same switch as the Sim Mobility server. The *note* and *s4* categories refer to the Galaxy Notes and the Galaxy S4s, respectively. Similarly,

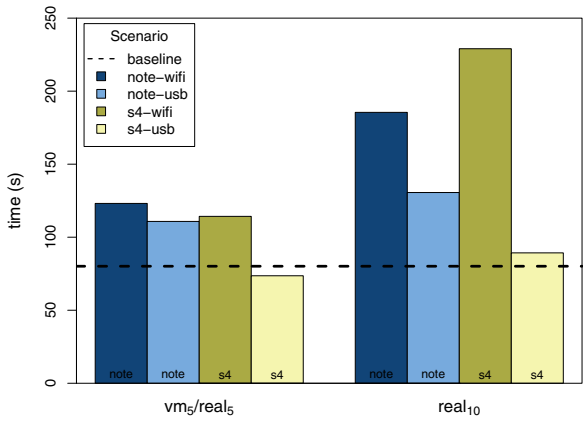

Fig. 7. Performance comparison of physical and virtualized phones

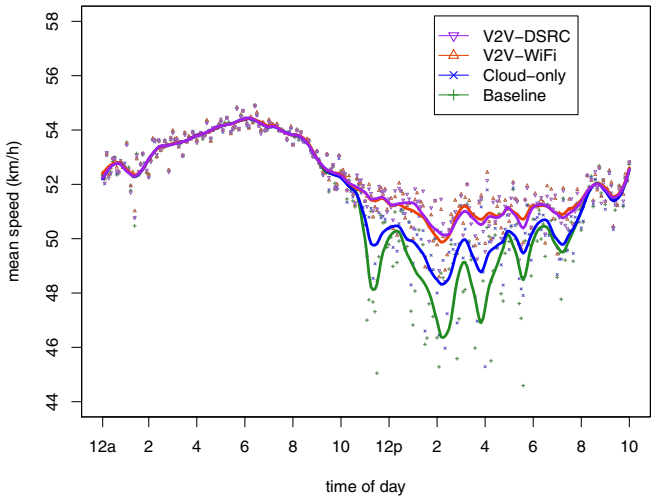

Fig. 8. Mean speed in simulation with 100 tokens. (Note. Adapted from [1].)

phones in the *wifi* category accessed the network over local Wi-Fi, while those in the *usb* category used USB reverse-tethering. Two scenarios were tested, in addition to the baseline. The real_{10} scenario featured 10 physical devices (either Notes or S4s), while the $\text{vm}_5/\text{real}_5$ scenario featured 5 physical devices and 5 virtualized (minimega) clients.

Several interesting trends stand out. First, moving from 5 virtual and 5 physical to 10 physical clients worsened performance within each category. This phenomenon is far more pronounced for phones using Wi-Fi, which might mean that physical clients are bound by network latency rather than the performance of the phones themselves. Second, USB reverse tethering is always faster than Wi-Fi, and in the case of the Galaxy S4s, it is actually faster than the baseline in the $\text{vm}_5/\text{real}_5$ scenario. The real_{10} scenario performs worse than the baseline, probably due to the overhead of switching 10 phones on a single USB controller. Third, the Notes unexpectedly outperform the S4s on Wi-Fi with real_{10}. This was possibly due to firmware differences in the Notes (running Cyanogenmod) and the S4s (running a stock kernel), but was not relevant to the current study, as we are concerned only with the best possible performance. In summation, we observed that the virtualized clients perform comparably to physical Android phones. Future work could explore the potential benefit of physical nodes with lower-latency access to the Sim Mobility server, such as Android devices with built-in ethernet adapters.

E. Potential Transit Design Implications

To put Similitude in perspective, Figure 8 presents results from an early study of the RoadRunner app, where a token set is used to limit road usage on a crowded roadway in Singapore. Loop detector counts were used to estimate network demand over the course of a full day of simulation time. For the *baseline* case, no congestion control algorithm was employed, causing average vehicle speed to drop off sharply during peak periods. The *cloud-only* case used LTE to coordinate token exchange; this required all messages to go through a central communication channel, but it was nonetheless effective at improving average speed. The *v2v-wifi* and *v2v-DSRC* cases featured direct token exchange between vehicles, and were the most effective at smoothing average speed.

V. Conclusion

We presented Similitude, a system comprised of the SimMobility traffic simulator coupled with Android emulators and an optional network simulator (ns-3). This system was designed to meet the emerging needs of traffic systems designers and mobile app developers when faced with pervasive apps with ever-increasing connectivity. We profile the individual components of Similitude, showing that all components can scale to the 288 connected clients that our current cluster supports. Beyond that limit, SimMobility was shown to scale well up to 100,000 agents, gaining performance through multi-threading. The Android emulators performed well in isolation, scaling quite well up to 1.0 to 1.5 times the CPU count. The current bottleneck is with ns-3, which was shown to exhibit poor scalability and performance above 20,000 agents. Combined, Similitude exhibited acceptable performance by distributing the Android emulators across multiple nodes.

Acknowledgment

The authors would like to thank Jason Gao at MIT for providing the RoadRunner app. In addition, we would like to thank David Fritz and Evan Tobac of Sandia National Laboratories for their initial help interfacing SimMobility and minimega.

References

[1] J. Gao and L. S. Peh, "Roadrunner: Infrastructure-less vehicular congestion control," in *21st World Conference, Intelligent Transport Systems*, 2014.

[2] D. Krajzewicz, G. Hertkorn, and P. Wagner, "An example of microscopic car models validation using the open source traffic simulation sumo," in *5th International Conference on ITS*, 2002, pp. 23–26.

[3] R. A. Waraich, D. Charypar, M. Balmer, K. W. Axhausen, R. A. Waraich, R. A. Waraich, K. W. Axhausen, and K. W. Axhausen, *Performance improvements for large scale traffic simulation in matsim*. ETH, Eidgenössische Technische Hochschule Zürich, IVT, Institut für Verkehrsplanung und Transportsysteme, 2009.

[4] Q. Yang and H. N. Koutsopoulos, "A microscopic traffic simulator for evaluation of dynamic traffic management systems," *Transportation Research Part C: Emerging Technologies*, vol. 4, no. 3, pp. 113 – 129, 1996.

[5] K. Basak, S. Hetu, Z. Li, C. Lima Azevedo, H. Loganathan, T. Toledo, R. Xu, Y. Xu, L.-S. Peh, and M. Ben-Akiva, "Modeling reaction time within a traffic simulation model," in *Intelligent Transportation Systems - (ITSC), 2013 16th International IEEE Conference on*, Oct 2013, pp. 302–309.

[6] E. Weingartner, H. vom Lehn, and K. Wehrle, "A performance comparison of recent network simulators," in *Communications, 2009. ICC '09. IEEE International Conference on*, June 2009, pp. 1–5.

[7] S. Mittal, "Opnet: An integrated design paradigm for simulations," *Software Engineering: An International Journal (SEIJ)*, vol. 2, no. 2, pp. 57–67, 2012.

[8] A. Varga *et al.*, "The omnet++ discrete event simulation system," in *Proceedings of the European Simulation Multiconference (ESM2001)*, vol. 9. sn, 2001, p. 185.

[9] S.-Y. Wang and C.-C. Lin, "Nctuns 6.0: A simulator for advanced wireless vehicular network research," in *Vehicular Technology Conference (VTC 2010-Spring), 2010 IEEE 71st*, May 2010, pp. 1–2.

[10] B. Schunemann, K. Massow, and I. Radusch, "A novel approach for realistic emulation of vehicle-2-x communication applications," in *Vehicular Technology Conference, 2008. VTC Spring 2008. IEEE*, May 2008, pp. 2709–2713.

[11] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: Automatic security analysis of smartphone applications," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '13. New York, NY, USA: ACM, 2013, pp. 209–220.

[12] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand, "Practical taint-based protection using demand emulation," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, ser. EuroSys '06. New York, NY, USA: ACM, 2006, pp. 29–41.

[13] Sandia Labs News Releases, "Sandia builds self-contained, android-based network to study cyber disruptions and help secure hand-held devices," October 2012. [Online]. Available: https://share.sandia.gov/news/resources/news_releases/sandia-builds-self-contained-android-based-network-to-study-cyber-disruptions-and-help-secure-hand-held-devices

[14] D. Jin, Y. Zheng, H. Zhu, D. Nicol, and L. Winterrowd, "Virtual time integration of emulation and parallel simulation," in *Principles of Advanced and Distributed Simulation (PADS), 2012 ACM/IEEE/SCS 26th Workshop on*, 2012, pp. 201–210.

[15] W. Dong, "A time management optimization framework for large-scale distributed hardware-in-the-loop simulation," in *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM-PADS '13. New York, NY, USA: ACM, 2013, pp. 265–276.

[16] G. Seguin, "Multi-core parallelism for ns-3 simulator," 2009. [Online]. Available: http://guillaume.segu.in/papers/ns3-multithreading.pdf