Network Simulation Bridge: Bridging Applications to Network Simulators

Harikrishna S. Kuttivelil
hkuttive@ucsc.edu
University of California
Santa Cruz, California, USA

Shesha Sreenivasamurthy
shesha@ucsc.edu
University of California
Santa Cruz, California, USA

Lakshmi Krishnaswamy
lakrishn@ucsc.edu
University of California
Santa Cruz, California, USA

Nayan Bhatia
nbhatia3@ucsc.edu
University of California
Santa Cruz, California, USA

Katia Obraczka
katia@soe.ucsc.edu
University of California
Santa Cruz, California, USA

ABSTRACT

Network simulators have been widely used to test and benchmark communication networks, their protocols and services. While new simulators have been developed to account for new technologies and standards, adapting and developing existing, popular network simulators to cope with the ever-increasing number and diversity of distributed applications is considerably more challenging. To fill this gap, this paper introduces the Network Simulation Bridge, or NSB for short, a simple, low-overhead pipeline consisting of a message server and client interface libraries that bridge together applications and network simulators. NSB is application-, network simulator-, and platform-agnostic and uses a "plug-and-play" approach which allows it to integrate any application front-end with any network simulator back-end. We showcase NSB by using it to integrate two different distributed applications with the OMNeT++ network simulator. Our performance evaluation confirms that NSB incurs relatively low overhead and exhibits adequate scalability. We open-source the NSB pipeline and its APIs through a publicly available repository.

CCS CONCEPTS

• **Networks** → **Network simulations**.

KEYWORDS

networks, network simulators, application simulators, simulation

ACM Reference Format:
Harikrishna S. Kuttivelil, Shesha Sreenivasamurthy, Lakshmi Krishnaswamy, Nayan Bhatia, and Katia Obraczka. 2023. Network Simulation Bridge: Bridging Applications to Network Simulators. In *Proceedings of Proceedings of the 19th ACM International Symposium on QoS and Security for Wireless and Mobile Networks (Q2SWinet '23)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3616391.3622771

Q2SWinet '23, October 30-November 3, 2023, Montreal, QC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0368-3/23/10.
https://doi.org/10.1145/3616391.3622771

1 INTRODUCTION

Network simulation platforms have been extensively used by researchers and practitioners to test, validate, and evaluate the performance of communication networks and their protocols. They offer a number of important advantages, including the ability to run reproducible experiments under a wide range of scenarios (e.g., network topologies, workloads, protocols, models of the underlying physical communication medium, etc). Additionally, they are often used as an initial experimental step to test and validate network protocols since, when compared to hardware testbeds, they are more widely accessible, easier to deploy and scale besides offering superior experimental diversity and reproducibility. Network simulators such as OMNeT++ [18] and ns-3 [7] have a large and diverse user and contributor community and have accumulated a considerable body of contributions and collective knowledge ranging from small-scale wired networks to more complex data center networks to ad-hoc wireless networks. Existing application simulators (e.g., vehicle-to-vehicle simulators, distributed machine learning simulators, etc.) simulate standalone components that do not account for the effects of the underlying communication network. However, modern paradigms (e.g., edge computing and edge intelligence systems) require cooperation among the different participating agents. This requires either application simulators to incorporate network simulation, or embed application knowledge in network simulators.

Trying to evolve and adapt network simulators to evaluate distributed applications is challenging due to the the ever-growing number and diversity of networked applications. A widely-used approach to evaluate distributed applications using network simulators is to observe application behavior, model the resulting application workload, and use it to drive the simulated network. Alternatively, network simulation platforms like *ns-3* [7] and OMNeT++ [18] allow users to build their applications atop the simulator. This requires re-implementing the application in the network simulator, which is labor- and time intensive and may introduce potential compatibility issues for each new application. Some network simulators and tools [13, 17] do offer the ability to connect applications directly to the simulated network, but to the best of our knowledge, these methods are not supported on all platforms and are often application- and simulator dependent. We discuss these approaches and other related work in more detail in Section 2.

To address the need to integrate a wide range of applications to network simulators – while being application-agnostic, simulator-agnostic, and platform-agnostic – we propose the Network Simulator Bridge, or NSB. NSB is a simple, low-overhead pipeline consisting of a message server (the NSB Server) and client interface libraries (the NSB Application Client and NSB Simulator Client) that together bridge applications and network simulators. In this paper, we introduce the NSB pipeline, describe its components, and showcase its use bridging two distributed application frontends with the OMNET++/INET'[18] network simulator backend. While, in this paper, we use OMNET++/INET as the network simulation backend, NSB is compatible with most well-known network simulators. Our performance evaluation confirms that NSB incurs relatively low overhead and exhibits adequate scalability.

Contributions. The overall contribution of our Network Simulator Bridge – NSB – is to provide a seamless, "plug-and-play" approach to integrate distributed applications to network simulator back-ends. To the best of our knowledge, NSB is the first application-network simulator bridge that is application-, network simulator-, and platform-agnostic. The contributions of this paper can be summarized as follows: **(1)** We present NSB, its pipeline, and components in detail and introduce a taxonomy to place NSB in context of related work; **(2)** We describe how NSB can be integrated and used to connect application front-ends to network simulation back-ends; **(3)** We showcase NSB by using it to integrate two different applications with the OMNeT++ network simulator [18]; **(4)** We evaluate NSB's performance to confirm that it incurs relatively light computational load and exhibits relatively small memory footprint.

Roadmap. The rest of this paper is organized as follows. Section 2 discusses related work. In Section 3, we introduce NSB and describe its pipeline and components in detail. Section 4 provides a high-level description of integrating and running NSB wtih applications. In Section 5, we showcase NSB by using it to integrate two driving applications – collaborative decentralized learning and autonomous vehicle platooning – to OMNeT++. In Section 6, we discuss NSB's performance and in Section 7, we conclude the paper.

2 RELATED WORK

Testing and evaluating distributed applications and systems to account for the effects of the underlying communication network infrastructure typically use: network simulators, custom purpose-built simulation systems, hardware testbeds, or a combination theeof. We discuss each of these approaches below and put our work on NSB in perspective as summarized in Table 1.

Network Simulators. ns-3 [7] and *OMNeT++* [18] are among the most popular network simulators in use today, and are open-source with large user communities, who have contributed a significant body of extensions and tools.However, these network simulators have been designed and built to evaluate the network itself and not necessarily complex distributed applications. ns-3 and OMNeT++ include *TAP* interfaces [5, 16] to allow integration with "real" hardware devices by interfacing with the device operating system's TAP interface. However, the TAP interface is not supported on all platforms, e.g., support was dropped for MacOS and was never available natively on Windows. While this may not pose an issue for Linux users, many students and researchers outside the networking domain work on other platforms. On the other hand, Mininet [3] and its variants (e.g., Mininet-WiFi [4] are popular network emulation solutions to evaluate networks and networked applications. However, Mininet as an emulator does not provide the option to easily deploy different underlying network stacks and will also require redeveloping the application for the emulation environment which, similar to network simulations, is both time- and labor-intensive and may need to omit key application features/functionality.

Application- and Domain-Specific Simulators. Many simulators are also built specifically for certain application domains, such as UAVs [2], connected vehicles [1], sensor networks [19], and more. While these are useful for specific contexts, there is considerable overhead in adapting them to new use cases. As new applications emerge and proliferate, having to develop adequate domain-specific simulators can pose a barrier to their proper validation and evaluation. Additionally, to be able to account for how the communication infrastructure affects application behavior and performance with reasonable fidelity will require developing realistic models of the underlying network.

Table 1: NSB and Related Work

Tool(s)	***Notes* or Example(s)**	**Features**									
		Universal application interface	Universal simulator interface	Popular platform compatibility	Does not require complete app re-implementation	Works with single application system	Works with distributed applications	Can model various network features	Can model various non-network features	Easily extensible for additional features	Accessible to all
NSB	*Used with a network simulator*	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Network Simulators	OMNeT++, ns-3	✗	✗	✓	✗	✗	✗	✓	✓	✗	✓
Network Simulators with TAP	TapBridge, ExtEthernet-TapDevice, DockEMU	✓	✗	✗	✓	✓	✓	✓	✓	✗	✓
Network Emulators	Mininet	✓	✗	✓	✓	✗	✓	✗	✗	✗	✓
Single purpose simulators	*Used to simulate specific implementations*	✗	✗	✗	✗	✓	✓	✗	✓	✗	✗
Hardware test beds	R2Lab, Orbit, PAWR	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
Connected device simulators	NodeRED	✓	✗	✓	✓	✓	✓	✗	✗	✓	✓

Hardware Testbeds. The need to test distributed applications on environments that closely approximate real-world scenarios has motivated the emergence of several private and public hardware testbeds [6, 12, 14]. However, testbed deployment and experimentation pose a number of limitations. First, hardware platforms have limited access as they are not portable and therefore cannot be deployed anywhere or by any user; even remote access cannot always be guaranteed. The use of physical hardware also presents both physical and virtual limits on deployment, scalability, and diversity, and the network stack options are limited to what is available on the hardware. Additionally, experimental reproducibility is either limited or not possible.

Integrating Applications with Network Simulators. A few solutions have been proposed to address the need to seamlessly integrate distributed applications with network simulation platforms. A notable example is DockEMU [13, 17]. While it served as an inspiration to our work, it relies on two limiting factors – tight integration with the *ns-3* framework and reliance on the *TapBridge* interface, only available within *ns-3*. As such, it cannot be used to integrate other network simulators. More importantly, it relies on the host operating system having a TAP interface, which, as discussed before, is currently not supported on two of the three major consumer operating systems.

3 THE NETWORK SIMULATION BRIDGE

NSB is designed to be application-, simulator-, and platform-agnostic, employing a "plug-and-play" approach that allows it to integrate any application front-end with any network simulator back-end. NSB uses a client-server architecture where the application front-end and the network back-end are the clients of the NSB server. The NSB clients are kept functionally simple and "slim" and use well-defined interfaces to interact with the NSB server which receives, stores, and forwards messages between the application and the network simulator. As shown in Figure 1, NSB consists of three functional components – the NSB Server, the NSB Application Client, and the NSB Simulator Client – which are described in Section 3.1. These components communicate with each other exchanging custom messages – NSB messages – using a well-defined TCP-socket based interface. NSB messages are described in detail in Section 3.2. The application(s), NSB server and network simulator can run in any configuration – local, remote, or distributed – as long as both the application(s) and the network simulator are able to communicate with the NSB server. The *application(s)* can refer to a single application (such the vehicle platooning simulator described in Section 5.2) or a collection of same or different applications (such as separate processes, containers, or machines, e.g., the containerized application introduced in Section 5.1).

3.1 NSB Components

3.1.1 NSB Server. The NSB Server handles the interaction between the application and the network simulator by acting as a relay. At its core, NSB receives, stores and and forwards application messages between the *NSB Application Client* and *NSB Simulator Client* using a set of *message queues.* As the server, it does not push or pull messages, but rather

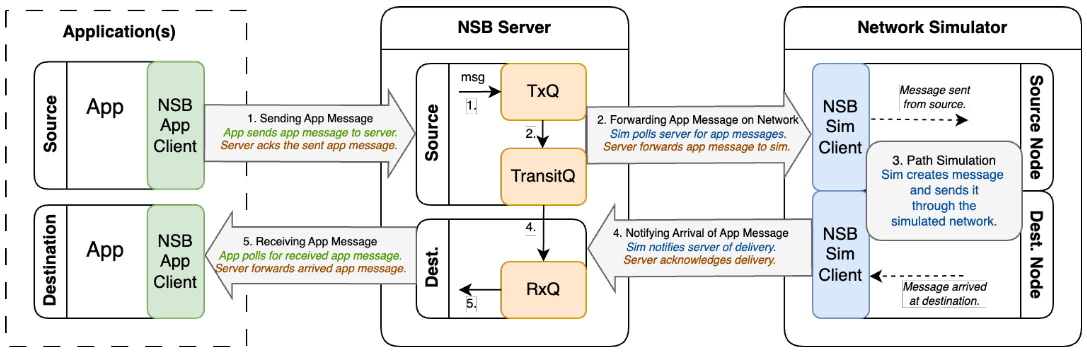
Application(s)
Source
App
NSB App Client
Destination
App
NSB App Client
1. Sending App Message
App sends app message to server.
Server acks the sent app message.
5. Receiving App Message
App polls for received app message.
Server forwards arrived app message.
NSB Server
Source
msg
1.
TxQ
2.
TransitQ
4.
Dest.
RxQ
5.
2. Forwarding App Message on Network
Sim polls server for app messages.
Server forwards app message to sim.
4. Notifying Arrival of App Message
Sim notifies server of delivery.
Server acknowledges delivery.
Network Simulator
NSB Sim Client
Message sent from source.
3. Path Simulation
Sim creates message and sends it through the simulated network.
NSB Sim Client
Message arrived at destination.
Source Node
Dest. Node

Figure 1: NSB components and application message lifecycle.

passes messages (e.g., application messages, messages from the network simulator) by replying to requests from the application- and network simulator clients. This allows the clients to remain slim with low overhead, as they just need to push and pull messages to/from the server, while the server handles message storage, queuing and management. Every application endpoint is represented by a network node in the simulator. The node IDs assigned by the application may be different from the network node IDs assigned by the network simulator. Therefore, the NSB server also maintains a mapping between the two IDs.

Implementation. The NSB Server is functionally a *multi-connection server* in which each connection is serviced by a separate and parallel process. Every client connection in NSB is persistent, i.e., under normal operation, stays connected throughout the duration of the experiment. While connections and services run in parallel, the NSB Server maintains a single collection of *nodes*, its internal representation of the devices on both the application and the simulated network. Because of this, in a network of n nodes, the NSB server maintains at most $2n$ active and parallel connections, as every application endpoint has to connect with the NSB Server and then to its corresponding simulator endpoint. The true management of parallelism within the server depends on the underlying operating system. Services on the attached connections access this shared collection of nodes to add new messages, move them between queues and between nodes (e.g., moving from the transit queue of a source node to a receiving queue of a destination node), and remove completed messages.

Queues. Within each of the *nodes* in the NSB Server, there is a set of queues – transmit (or TX), transit, and receive (or RX) – which are different message buffers through which an application message stored in the NSB Server moves as it goes through its lifecycle. The stored message consists of: (1) the message ID (generated incrementally); (2) the application endpoint's source and destination addresses; (3) the application message itself; and (4) the message size. When the NSB Server receives a message from the NSB Application Client, it stores the message and the metadata in the transmitting node's TX queue. It then moves the message to the node's transit queue when the NSB Simulator Client pulls the message from the NSB Server. The message is moved from the transmitting node's transit queue to the receiving node's RX queue when the NSB Simulator Client pushes a delivery notification to the NSB Server. The NSB Server then removes the message from the receiving node's RX queue and from the server completely when the NSB Application Client pulls the application message.

3.1.2 *NSB Application Client.*

The NSB Application Client is the interface between the application and the NSB Server. It is implemented as a library that applications can link. It is responsible for: (1) packing and pushing outgoing application messages to the NSB Server; and (2) pulling and unpacking received application messages from the NSB Server once they are transmitted through the simulated network. By design, the NSB Application Client API is meant to act like simple RPC calls that hide actual socket interface between the NSB Application Client and the NSB Server.

3.1.3 *NSB Simulator Client.*

The NSB Simulator Client handles the interactions between the network simulator and the NSB Server and is implemented as a library that the network simulator backend can link. It is responsible for: (1) pulling application messages from the NSB Server for the network simulator to send over the simulated network; and (2) notifying the NSB Server when messages have been delivered to their destination through the network simulator.

3.2 NSB Messages

As described in Figure 1, the different NSB components use messages to communicate and synchronize.

NSB Application Client ↔ NSB Server. messages are used by the NSB Application Client to send application messages and query their status. Header fields include the length of the payload, its type, message source and destination, and a variable-length payload (i.e., the original application message). To broadcast message to all reachable nodes, the destination address is set to $0xFFFFFFFF$.

NSB Simulator Client ↔ NSB Server. messages are used to send and receive application messages between the network simulator and the NSB server. The header includes the type of message, the length of the payload, source, destination, the message ID (referring to the relevant application message stored in the NSB Server) and a variable-length payload. The source and destination IDs specified are those that are assigned by the network simulator to the nodes in the simulator network. The NSB bridge will perform the translation between the network node IDs and the application node IDs when accessing TX and RX queues and populate the translated values in any response messages, too.

3.3 Application Message Lifecycle

Figure 1 shows both the steps of the application message lifecycle and the corresponding transfers that occur in the NSB Server queues.

NSB Application Client → NSB Server. Upon receiving a message from an application endpoint (the source) that needs to be sent through the network to another application endpoint (the destination), the NSB Application Client encapsulates the application message within an *AB_SEND_MSG* message with the source and destination addresses and sends it to the NSB Server. When the NSB Server receives a message from an NSB Application Client, it creates a new entry in the corresponding TX queue. The NSB Application Client awaits the *AB_SEND_MSG_ACK* response, which comes with a return code to indicate success and a message ID for tracking purposes. This message ID can be used with the *AB_MSG_GETSTATE* message to request the state of the specified application message, to which the NSB Server responds with *AB_MSG_STATE* indicating the application message state, either MSG_STATE_QUEUED (message is in the sender's TX queue) or MSG_STATE_NOTFOUND (message is not found, implying it was already sent over the simulated network).

NSB Server → NSB Simulator Client. The NSB Simulator Client polls the NSB Server for outgoing application messages containing self address as source address using *SB_RECV_MSG* message. The NSB Server checks the corresponding TX queue for application messages with the specified identifiers. If found, the NSB Server moves the application message from the TX queue to the transit queue. It then replies to the NSB Simulator Client with the application message and relevant information in a *SB_RESP_MSG* message containing either empty (no messages) or with the application message, source, destination, and message ID.

NSB Simulator Client → NSB Server. When the application message arrives at the destination in the network simulator, the NSB Simulator Client sends a delivery notification to the NSB Server with a *SB_DELIVER_MSG* with the source, destination, message ID, and optional additional information. Upon reception of this notification, the NSB server moves the stored application message from the transit queue to the RX queue. The NSB Simulator Client awaits the *SB_DELIVER_MSG_ACK* response that includes an error code, indicating success or failure.

NSB Server → NSB Application Client. When NSB Application client tries to receive an application message, it polls the NSB Server by sending a *AB_RECV_MSG* request with its own address as the destination. The NSB Server searches for the message in the destination RX queue and sends a *AB_RESP_MSG* reply back to the NSB Application Client containing either empty (in case of no messages) or the application message along with the original source and destination. The NSB Application Client unpacks the message and delivers it to the application endpoint. The message will be removed from the NSB Server as it would have completed its lifecycle.

4 INTEGRATING NSB

In this section, we describe how to integrate applications and network simulators using NSB. Consistent with NSB's overall design principles, integrating applications and network simulation platforms using NSB is done through its simple, well-defined interface. Both the application- and simulator-side interfaces consist of a small set of function calls and involve adding only a few lines of code to connect the application(s) and network simulator to each other via the NSB server. The interfaces are also designed to be as general as possible, consistent with NSB's goal of being application- and network simulator agnostic, aligned with its "plug-and-play" usage model. Using NSB requires applications to connect to the NSB Server using its IP address and port number through the NSB application interface. The current interface provides the *Initialization* method to launch and initialize the *NSBApplicationClient*, *Send* and *Receive* methods to send and receive messages over NSB, and the *State* method to poll the status of a message being queued for transmission over NSB. The *Initialization*, *Send*, and *Receive* methods are required for using NSB, while the *State* method is optionally invoked for user benefit. Complementary to the NSB application interface, the NSB network simulator interface is used to connect the application front-end with a network simulator back-end through the NSB Server. Integrating this interface into the network simulator follows a similar approach to the applcation-NSB Server integration. The current interface provides *Initialization*, *Fetching*, and *Delivery* methods. The *Initialization* method which must be invoked by each node being simulated so that each one of them maintains its own persistent connection to the NSB Server. *Fetching* is called when a node needs to send a message over the network, while *Delivery* is invoked when a message arrives at a destination node.

Integrate the NSB Application Client and NSB Simulator Client into the application front-end and simulator back-end respectively. Then, follow these steps:

(1) Launch the NSB Server with `python nsb_server.py` to allow connections from both Clients.
(2) Start the network simulator back-end to connect to the NSB Server via the NSB Simulator Client, initiating message polling.
(3) Launch the application front-end to send messages to the simulated network via the NSB Server.

When the simulation experiments are completed, the application(s) and the network simulator can be terminated. The NSB server will continue to run until it is terminated (e.g., through the command line). This will end all connections to the NSB Server, disconnecting it from both the application(s) and the network simulator.

5 NSB USE CASES

Our work on NSB was initially motivated by a number of applications and use cases that need to account for how the underlying communication network will affect the driving application performance.which without NSB, would be labor-intensive and limited. In this section, we discuss two of these applications – *collaborative decentralized learning* and *autonomous vehicle platooning* – built with different implementations of NSB.

5.1 Collaborative Decentralized Learning

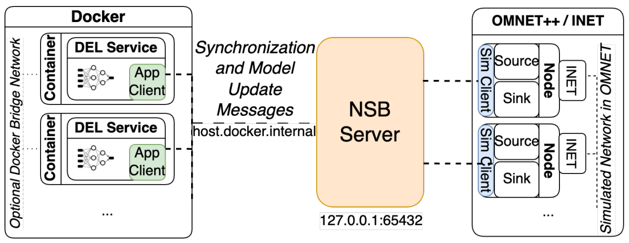

Figure 2: Using NSB to bridge containerized decentralized learning with the OMNeT++ and INET network simulator.

Edge intelligence [21] is an emerging paradigm of artificial intelligence that is situated in the network edge and built on edge computing principles. Research in edge intelligence is largely driven by constraints in computation, memory, data privacy, and network communication bandwidth. Distributed learning is an emerging type of edge intelligence, and includes centralized federated learning [11], fully decentralized federated learning [9], and clustered decentralized federated learning [15]. While there are a number of experimental platforms to evaluate the performance of different learning approaches, it is imperative to understand how the different decentralized learning paradigms are affected by the conditions of the network edge and how much network overhead these different paradigms generate. In this NSB use case, which is illustrated in Figure 2, we conducted a study comparing the performance of federated, decentralized federated, and clustered federated learning. We use a common process running on each node to implement these different learning paradigms so that participating nodes assume roles depending on the selected learning approach. We use NSB to connect different containerized decentralized learning deployments to the backend network simulator (e.g., OMNeT++ [18]) where we set up relevant network models (i.e., network topologies, link bandwidths, loss rates, etc), to observe the effect of the network on the different decentralized learning paradigms and vice-versa. In our experiments, we assume a wireless network with stationary nodes that use the UDP transport protocol to exchange messages. This scenario has been implemented in OMNeT++ using the INET network stack [8]. Application messages sent over the network include model-sharing messages generated by the decentralized, collaborative learning mechanism as well as process messages. Figure 2 illustrates how NSB is used to run our decentralized learning experiments.

5.2 Vehicle Platooning

According to the U.S. Department of Transportation (USDOT), *platooning* is defined as "a coordinated operation of two or more vehicles via cooperative adaptive cruise control (CACC)" [10]. Vehicles in a platoon follow each other in close proximity which improves fuel efficiency and road capacity by reducing aerodynamic drag and packing more vehicles on the road, respectively. Coordination among vehicles is required to form and maintain platoons which are achieved by periodically transmitting vehicle speed and position information to neighboring vehicles using wireless communication. In order to test and evaluate different platooning approaches in an environment that models real deployments as close to reality as possible, we need: (1) a platform that incorporates a physics engine that models, among other features, realistic forces acting on the vehicles such as friction and stress on joints in addition to vehicle center of gravity and mass distribution. To that end, we use the *Webots* physics simulator [20] due to its ease of use, extensibility, and thorough documentation. To model the underlying wireless network, we would need to implement a complete network stack in the Webots simulator with the necessary protocols and realistic wireless medium models, which would be both time- and labor-intensive. NSB provides us with a significantly simpler alternative, i.e., use an existing network simulator, e.g., OMNeT++[18]. As shown in Figure 3, our vehicle platooning simulation infrastructure consists of three components, namely, Webots and OMNeT++ to simulate, respectively, autonomous vehicles and the underlying wireless network, and NSB to bridge the two. Each vehicle in the platoon is represented by a process in Webots that connects to NSB's client library. Through NSB, every vehicle in Webots is represented by a corresponding mobile node in OMNeT++ which are connected via a wireless network using OMNeT++'s Wireless framework.

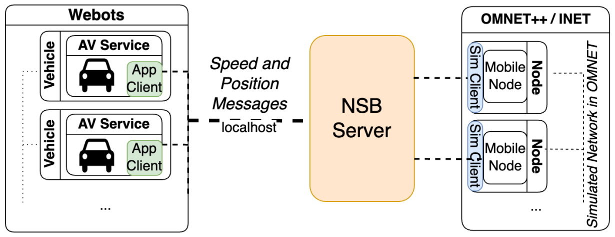

Figure 3: Using NSB to bridge vehicle platooning simulation with the OMNeT++ network simulator.

Average CPU Utilization (%)							
		Number of Nodes					
		5	**15**	**25**	**50**	**100**	**200**
Messages/second	**10**	1.3	2.02	2.59	4.26	38.83	51.14
	30	2.7	3.45	4.06	6.07	41.43	53.08
	50	4.0	4.81	5.35	7.3	44.84	54.38
	100	7.94	7.74	8.69	10.0	45.57	53.06
	300	23.08	15.61	14.68	16.31	46.81	54.71
	500	41.28	28.25	24.55	25.1	45.88	52.46
	1000	58.44	44.34	38.06	41.88	47.5	53.79

(a) Average CPU Utilization.

Peak Memory Usage (MB)							
		Number of Nodes					
		5	**15**	**25**	**50**	**100**	**200**
Messages/second	**10**	13.05	13.05	13.01	13.05	13.05	13.38
	30	13.05	12.97	13.02	13.07	13.05	13.38
	50	13.04	13.03	13.05	13.04	12.96	13.38
	100	25.8	13.05	13.05	13.05	13.05	13.33
	300	49.37	30.41	13.03	13.05	13.05	13.35
	500	70.93	58.96	39.97	13.04	13.05	13.36
	1000	83.8	87.3	75.28	13.66	13.42	13.36

(b) Peak Memory Usage.

Table 2: Standalone performance of NSB Server running on an AWS t2.micro EC2 instance.

6 NSB PERFORMANCE

We evaluate the NSB Server's runtime performance measuring its average CPU utilization and memory usage. In these experiments, the driving application, the NSB Server, and the "ghost" network simulator were executed on a *t2.micro* AWS E2 instance, running an Intel Xeon E5-2676 v3 CPU at 2.40 GHz (1 vCPU at 3.2 GHz) and 1 GB of RAM, using Ubuntu 20.04.6 LTS. The "ghost" simulator executes the NSB API function calls that are usually invoked by the network simulator backend. We opted to use this "ghost" simulator instead of an actual network simulator so we can isolate the performance of the NSB Server. To avoid incurring network latency, we had the NSB Application Client, the NSB Server, the NSB Simulator Client and the ghost simulator running on the same EC2 instance. We used Linux *cgroups* to isolate the processes, allocating 75% of the CPU and RAM to the NSB Server and the remaining 25% to the other processes. The Python library *psmonitor* was utilized for calculating the memory and CPU utilization of the server. We varied the number of nodes in the network – ranging from 5 to 200 nodes – and the number of messages sent per second over the entire network – ranging from 10 to 1000 messages per second. The ghost simulator uses a fixed request rate of 10 messages per second per node (running in parallel). Table 6 summarizes the resulting NSB Server's average CPU utilization and peak memory usage averaged over 3 runs with no significant variation between the runs. Despite the experimental platform's modest processing and memory capabilities, the results demonstrated stable performance across the wide range of configurations, i.e., number of nodes and workload. For a fixed number of nodes, as the message rate increases, the CPU utilization increases as expected. Similarly, for a fixed message rate, as the number of nodes increases, the CPU utilization also increases. This indicates that the server is efficiently processing the increased workload by utilizing more CPU resources and as the number of nodes and message rate increase, CPU utilization scales appropriately. Memory utilization also remains stable across a large range of scenarios, but we see more complex behavior in some use cases. We observe that memory usage increases with higher message rate, but at fixed message rates, the memory usage decreases with higher number of nodes, with a base memory usage of 13 MB. This is because the NSB Server keeps track of all messages queued for or currently going through the simulated network. As the message rate increases, the congestion in the simulated network increases, resulting in more messages being stored in the NSB Server. We note that for the same rate of messages over the network with more nodes – resulting in a higher net message intake by the ghost simulator – this congestion is alleviated and memory usage also decreases. Similar experiments driven by the applications mentioned in Section 5.1 and 5.2 using OMNET++ as the network simulator backend [1] yielded similar findings. Overall, the network simulator message intake rate was the main limiting performance factor, and the NSB Server has negligible impact on the overall performance of the entire experimental pipeline. These results were largely expected: NSB acts as a lightweight message relay system, where its main processing and memory needs arise from queue management and messages stored at the NSB Server.

7 CONCLUSION

In this paper, we introduced the Network Simulation Bridge – NSB – a simple, low-overhead, extensible tool that can be used to seamlessly bridge a wide range of distributed applications with different network simulation platforms. This integration allows distributed applications' design and evaluation to account for the impact of the underlying communication infrastructure. The NSB pipeline includes a message server – the NSB Server – at its core, and interface libraries – the NSB Application Client and NSB Simulator Client. NSB's simple interfaces and lightweight operation facilitate their integration with existing applications and network simulators. We showcased NSB by using it to bridge two different distributed applications with the OMNeT++ network simulator [18]. Our performance evaluation confirms that NSB incurs relatively low computation and storage overhead and exhibits adequate scalability. Its easy extensibility and being an open-source tool, NSB has the potential to gain wide adoption by distributed application as well as network simulation developers.

REFERENCES

[1] Md Salman Ahmed, Mohammad Asadul Hoque, and Phil Pfeiffer. 2016. Comparative study of connected vehicle simulators. In *SoutheastCon 2016*. IEEE, 1–7.

[2] Michael A Day, Michael R Clement, John D Russo, Duane Davis, and Timothy H Chung. 2015. Multi-UAV software systems and simulation architecture. In *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 426–435.

[3] Rogério Leão Santos De Oliveira, Christiane Marie Schweitzer, Ailton Akira Shinoda, and Ligia Rodrigues Prete. 2014. Using mininet for emulation and prototyping software-defined networks. In *2014 IEEE Colombian conference on communications and computing (COLCOM)*. Ieee, 1–6.

[4] Ramon R Fontes, Samira Afzal, Samuel HB Brito, Mateus AS Santos, and Christian Esteve Rothenberg. 2015. Mininet-WiFi: Emulating software-defined wireless networks. In *2015 11th International Conference on Network and Service Management (CNSM)*. IEEE, 384–389.

[5] Abdurrahman Fouda, Ahmed N Ragab, Ali Esswie, Mohamed Marzban, Amr Naser, Mohamed Rehan, and Ahmed S Ibrahim. 2014. Real-time video streaming over ns3-based emulated lte networks. *Int. J. Electr. Commun. Comput. Technol.(IJECCT)* 4, 3 (2014).

[6] Abhimanyu Gosain. 2018. Platforms for advanced wireless research: Helping define a new edge computing paradigm. In *Proceedings of the 2018 on Technologies for the Wireless Edge Workshop*. 33–33.

[7] Thomas R Henderson, Mathieu Lacage, George F Riley, Craig Dowell, and Joseph Kopena. 2008. Network simulations with the ns-3 simulator. *SIGCOMM demonstration* 14, 14 (2008), 527.

[8] INET. [n. d.]. https://inet.omnetpp.org. ([n. d.]). https://inet.omnetpp.org Open-source OMNeT++ model suite.

[9] Anusha Lalitha, Shubhanshu Shekhar, Tara Javidi, and Farinaz Koushanfar. 2018. Fully decentralized federated learning. In *Third workshop on bayesian deep learning (NeurIPS)*, Vol. 2.

[10] Jeff Loftus. 2018. Truck Platooning The State of the Industry and Future Research Topics. *U.S. Department Of Transportation* (Jan 2018).

[11] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*. PMLR, 1273–1282.

[12] Farzaneh Pakzad, Marius Portmann, Thierry Turletti, Thierry Parmentelat, Mohamed Naoufal Mahfoudi, and Walid Dabbous. 2018. R2Lab Testbed Evaluation for Wireless Mesh Network Experiments. In *2018 28th International Telecommunication Networks and Applications Conference (ITNAC)*. IEEE, 1–6.

[13] Erick Petersen, Guillermo Cotto, and Marco Antonio To. 2019. Dockemu 2.0: Evolution of a network emulation tool. In *2019 IEEE 39th Central America and Panama Convention (CONCAPAN XXXIX)*. IEEE, 1–6.

[14] Dipankar Raychaudhuri, Ivan Seskar, Max Ott, Sachin Ganu, Kishore Ramachandran, Haris Kremo, Robert Siracusa, Hang Liu, and Manpreet Singh. 2005. Overview of the ORBIT radio grid testbed for evaluation of next-generation wireless network protocols. In *IEEE Wireless Communications and Networking Conference, 2005*, Vol. 3. IEEE, 1664–1669.

[15] Felix Sattler, Klaus-Robert Müller, and Wojciech Samek. 2020. Clustered federated learning: Model-agnostic distributed multitask optimization under privacy constraints. *IEEE transactions on neural networks and learning systems* 32, 8 (2020), 3710–3722.

[16] Thomas Staub, Reto Gantenbein, and Torsten Braun. 2011. VirtualMesh: an emulation framework for wireless mesh and ad hoc networks in OMNeT++. *Simulation* 87, 1-2 (2011), 66–81.

[17] Marco Antonio To, Marcos Cano, and Preng Biba. 2015. DOCKEMU–A Network Emulation Tool. In *2015 IEEE 29th international conference on advanced information networking and applications workshops*. IEEE, 593–598.

[18] András Varga and Rudolf Hornig. 2010. An overview of the OMNeT++ simulation environment. In *1st International ICST Conference on Simulation Tools and Techniques for Communications, Networks and Systems*.

[19] Aleksandar Velinov and Aleksandra Mileva. 2016. Running and testing applications for Contiki OS using Cooja simulator. (2016).

[20] Webots. [n. d.]. http://www.cyberbotics.com. ([n. d.]). http://www.cyberbotics.com Open-source Mobile Robot Simulation Software.

[21] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. 2019. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proc. IEEE* 107, 8 (2019), 1738–1762.

[1]These results are ommitted here due to space limitations.