A Game Engine-Based Visualizer for ns-3 Simulations

Hyeokjae Lee, Woochan Yoon and Hyogon Kim

***Abstract*—ns-3 is an extremely popular and still evolving simulation platform that enables flexible experimentation for virtually any modern networking and communication systems. One aspect that needs improvement, however, is its visualization utility. The best-known visualizer for ns-3 is NetAnim, but this event-based animator does not provide the most efficient visualization service. In this paper, we discuss a new, frame-based visualizer for ns-3 called UVNS. An interesting feature of the visualizer is that it utilizes Unity, a cross-platform game engine. Owing to the breakneck speed of game technology advancement, the speed and visual components of the engine are highly optimized. By exploiting them, UVNS can greatly facilitate the animation of ns-3 simulation log both in performance and in presentation quality.**

***Index Terms*—ns-3, network simulation, animation, Unity, game engine**

I. Introduction

NS-3 [1] is an extremely popular and continuously evolving simulation platform that enables experimentation of virtually any modern networking and communication systems. One disappointing aspect is its representative visualization tool, NetAnim [1]. NetAnim is a standalone, Qt5-based software executable [2] that uses a trace file generated during an ns-3 simulation to display the topology and animate the packet flows between nodes. Unfortunately, it has multiple drawbacks. First, the simulation events in the log file can only be processed in entirety. It does not have a way to selectively process and visualize particular events. Second, the animation is strictly event-based. Therefore, when there is a large volume of logged events to be shown, NetAnim would severely slow down or, in extreme cases, crash before completion. Third, there is various information it does not show in animation such as queue size fluctuations.

Although the ability to animate large-scale simulated system dynamics is a highly desirable characteristic, it is not easy to achieve, as NetAnim case tells us. It requires the visualizer to handle large memory and computation resources efficiently. We found that game engines that typically deal with a large number of dynamic entities satisfy the requirement well. Therefore, we implemented a prototype visualizer based on Unity [3], a cross-platform game engine. Owing to the breakneck speed of game technology advancement, the speed and visual components of the engine are highly optimized. By exploiting them, our prototype called Unity-based Visualizer for ns-3 (UVNS) can greatly facilitate the animation of ns-3 simulation log both in performance and in presentation quality. Our prototype implementation can animate the simulation log with a large number of nodes and in-flight packets without slowdown, for which NetAnim experiences difficulties.

II. Implementation

Fig. 1 shows how the UVNS interfaces with other elements in the ns-3 simulation. The user provides the simulation model in a C++ script, and ns-3 executes the model. During execution, a log file in XML format can be produced. After the simulation is completed, UVNS dynamically and immediately instantiates animation objects such as nodes and packets while parsing down the log file. Using the Unity engine, UVNS then visualizes the packet-level dynamics in animation. Below, we describe these steps in more detail.

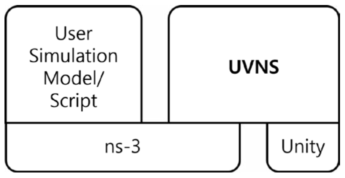


Fig. 1. UVNS

A. Simulation log creation

UVNS utilizes the same module (`animation-interface.cc`) in ns-3 that is used by NetAnim to create the simulation log. The log is in XML format, and consists of tags and attributes, where tags represent the type of information, and attributes represent the content. In the current version, UVNS only shows the information regarding nodes, packets, routing, queue size and the size of TCP socket buffers at each node. Any other information can also be utilized for visualization as long as it is available in the simulation log file, either by modifying the `animation-interface.cc` or by adding tracking functions to the scenario. But it is left for a future work because this paper is focused on showing the qualitative difference game engines can make in the visualization of large-scale ns-3 simulations.

B. Log processing

The fact that NetAnim uses the entire log created by `animation-interface.cc` for animation has a significant issue. Potentially unnecessary information for animation such as each node's IP address and routing details may be included in the log, making the XML log file excessively voluminous. It can cause extremely slow animation in NetAnim.

However, it is easier to address this problem in UVNS, by selectively creating objects that the user wants to examine during load and then by showing them in animation. This feature greatly contributes to the scalability of UVNS.

UVNS also exploits software optimizing features in Unity for object creation during load, in particular, *Prefab*. Prefab is originally a construction term, referring to factory-made components that can be assembled on-site to complete a building. In Unity, it is used in a similar sense, because Unity Prefabs are created in advance in the desired form, stored, and can be copied as instances when needed, allowing for multiple reuses. The advantages of using Prefabs in UVNS are as follows:

- Objects can be instantiated from Prefabs without the need to create them beforehand or modify the code. Also, it reduces the memory size to store objects.
- Modifying objects is straightforward. When creating an instance, it copies a Prefab as its parent. Therefore, modifications to the Prefab automatically apply to all instances, making it convenient to change the visuals or structure of objects without modifying each individual object separately.

Therefore, we created Prefabs for nodes and packets and instantiated them in the code, allowing them to be dynamically loaded into animations at runtime.

We added C# scripts as Unity *components* to the Prefabs. Using the scripts, we can exert a fine control over individual objects while exploiting Prefabs, by specifying different values for the public variables in the script. Then, each object instance created through the Prefab can store different data and perform different actions. We apply this feature to objects that behave mostly the same but have slight differences. For example, packets may have the same appearance, origin and destination, but times of departure and arrival are all different. To accommodate such individual differences in attributes, public variables written in the C# script in the components are used to store the attributes from the XML document for each instance when running the animation.

C. Animation

When running animations in Unity, we create animation frames with Unity *deltaTime* as the time gap between consecutive frames. The advantage is that by changing the *TimeScale* parameter value, a feature built into Unity that controls the speed of time in game play, we can adjust deltaTime in UVNS. This allows us to observe the animation in any speed we want and even dynamically change the speed during animation, giving us high flexibility in examining the simulated system.

Once deltaTime is determined by the TimeScale parameter, it controls the animation speed as illustrated in Fig. 2. If there are multiple rendering events (e.g. queue size fluctuation) in one deltaTime, only the last one is rendered while the others are ignored. Note that a rendering event may concur with a simulation event or may be independent. The packet count in the queue is an example of the former. In the event-based visualizer like NetAnim, all rendering events must be displayed. However, in UVNS, only one rendering event is displayed per deltaTime. For example, if the count went up from 0 to 100 in 100 packet enqueue events, all 100 rendering events should be displayed in NetAnim. On the other hand, if the animation speed is set to be high so that all 100 events takes place in one deltaTime, only the last count (i.e., 100) is displayed in UVNS. We call this feature of UVNS "*frame-based* visualization" as opposed to the event-based visualization in NetAnim. Multiple

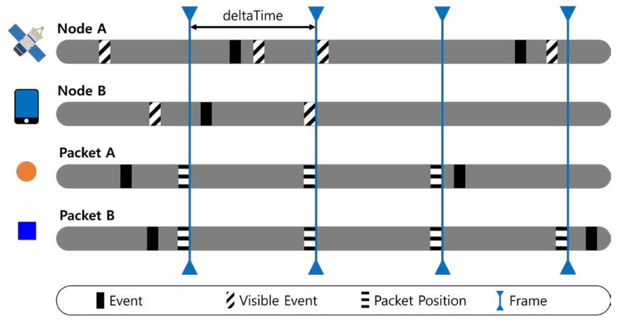


Fig. 2. Optimization in UVNS frame-based visualization

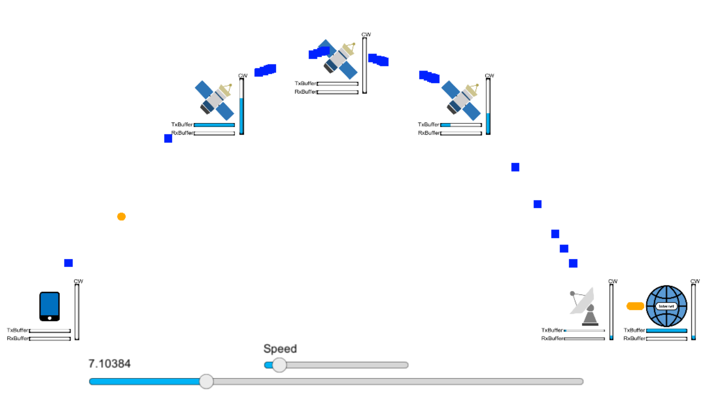
rendering events can correspond to a single simulation event. For example, the movement of a packet along a link requires multiple rendering events although there is only one packet transmission event. Again, many of these rendering events can be ignored if deltaTime is large, i.e., animation speed is high.

Fig. 3 shows the speed bar and the video progress bar at the bottom of UVNS animation window (which is the "game window" in Unity). On the left is the Non-Terrestrial Network (NTN) scenario simulation based on Low Earth Orbit (LEO) satellite constellation where the movements of hundreds of satellites along with packet transmissions are simulated. On the right is TCP congestion control and pacing model running on a dumbbell topology. The speed bar sets the aforementioned TimeScale parameter in UVNS animation so that the animation speed (i.e., deltaTime) can be dynamically changed. Therefore, by separating the notions of a simulation event and a rendering event, UVNS can achieve its efficiency.

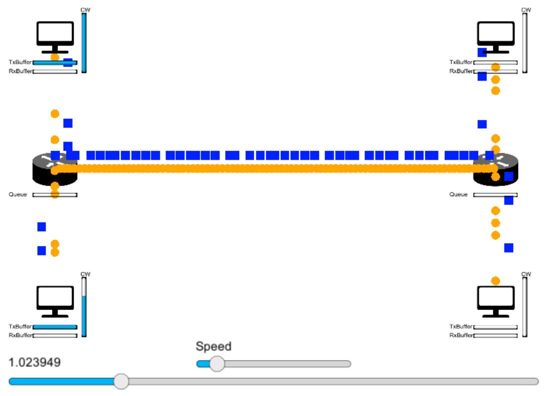
In addition to raw animation speed, consistency is another strength of UVNS animation. In NetAnim, the animation speed is directly affected by viewport selection. If there are only a small number of objects (e.g. nodes) in the viewport, the simulation speed approaches its potential maximum because the actions regarding the objects not selected in the viewport need not be rendered. If there are many objects, however, it becomes slow because of many events that should be rendered. Therefore, it is impossible to provide a consistent animation speed that a user may want to set at. In UVNS, however, the simulation speed is consistent over all viewport sizes and locations, as the progression of animation is flexibly controlled by the TimeScale parameter. Combined with the efficiency, it makes UVNS simulation much more consistent than NetAnim.

D. Simulation inspection

For inspecting individual objects during animation, UVNS exploits Unity *inspector window* that is originally used for



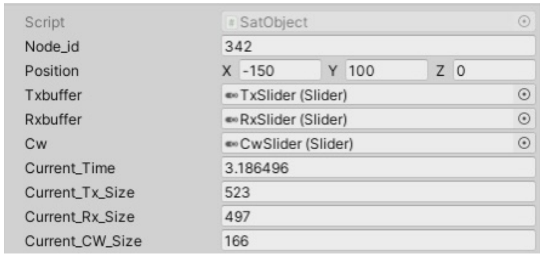
(a) LEO-based file transfer in 3GPP NTN scenario



(b) TCP pacing for congestion control in dumbbell topology

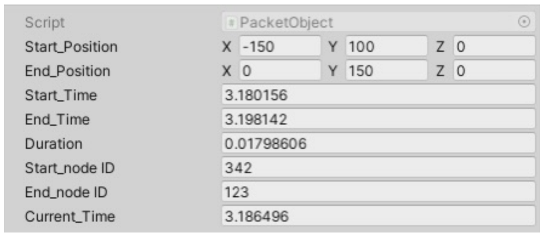
Fig. 3. Screen captures of UVNS animation examples; circles = data segments, squares = acknowledgements

monitoring and editing individual objects in a game program. Through inspector windows, UVNS can show the information regarding a selected node or packet that appear in the animation window. In Unity-based games, we can monitor any objects to which we attached a script. Therefore, we will extend the inspector facility to other objects than nodes and packets in future versions of UVNS. Fig. 4 shows that UVNS can display the details of selected node and packet in the inspection window. Currently, they appear on the right side of the animation window when activated. In the future, we will implement the inspection windows to appear inside the animation window for more direct presentation.



Script	SatObject				
Node_id	342				
Position	X -150		Y 100		Z 0
Txbuffer	TxSlider (Slider)				
Rxbuffer	RxSlider (Slider)				
Cw	CwSlider (Slider)				
Current_Time	3.186496				
Current_Tx_Size	523				
Current_Rx_Size	497				
Current_CW_Size	166				

(a) Node



Script	PacketObject		
Start_Position	X -150	Y 100	Z 0
End_Position	X 0	Y 150	Z 0
Start_Time	3.180156		
End_Time	3.198142		
Duration	0.01798606		
Start_node ID	342		
End_node ID	123		
Current_Time	3.186496		

(b) Packet

Fig. 4. UVNS inspector windows

The code for the initial UVNS prototype is available on Github [4].

III. Performance Evaluation

We compare the speeds of NetAnim and UVNS for the log processing that takes place during initial loading and for the animation execution itself. When UVNS reads in the simulation log, it creates only the objects of user's interest and does it dynamically as they appear in the log. In NetAnim, however, it creates all objects associated with recorded events. In case there are a large number of events in the simulation, NetAnim can create huge overhead even though the animation does not need all objects because the user wants to see only a fraction of them. The NTN scenario in Fig. 3(a) is a good example, so we use this case for the performance comparison in this section. In the scenario, more than 630 satellites exist in the system, whose positions constantly change. Although only six nodes are involved in actual communication, NetAnim spends excessive amount of time to compute and render the movements of the satellites. We found that NetAnim animation takes incomparably long time. So, for the sake of simply making the comparison feasible between NetAnim and UVNS, we pre-process the log file for NetAnim so that NetAnim can only show the selected six nodes. We stress that this pre-processing step has been performed separately by us, as it is not provided in NetAnim.

A. *Load speed*

Since all the scripts for parsing XML files and generating the necessary objects in UVNS is written in C#, the added pre-processing for NetAnim was also programmed in C#. With pre-processing for NetAnim, the load times become comparable for NetAnim and UVNS (Fig 5). Across simulation times, UVNS still exhibits slightly shorter times. With the load speed comparable and relatively small, the animation time becomes the decisive factor.

B. *Animation speed*

Fig. 6 shows the stark differences in the wall-clock animation times for 5, 10, 20, and 30 seconds of simulated LEO constellation system. For a mere 30 seconds-long simulation, NetAnim takes as long as 6,800 seconds (1.9 hours) to play the animation. We found that the time for NetAnim without pre-processing is at least four times longer than with pre-processing. In contrast, it is only 30 seconds in UVNS, i.e.,

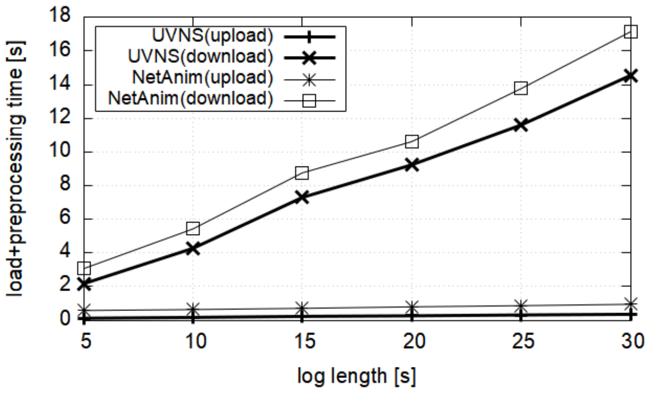


Fig. 5. Load time comparison; pre-processing applied to NetAnim

more than two orders of magnitudes smaller. In fact, UVNS plays the animation along exactly on the simulated time axis.

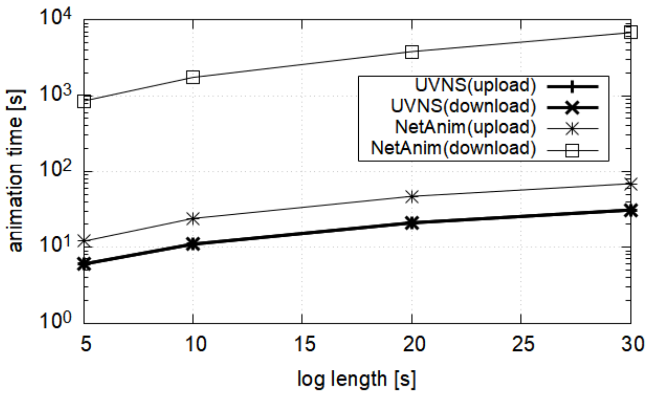
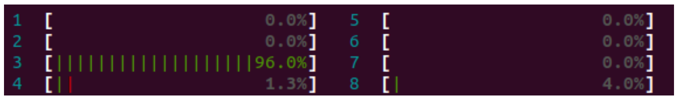


Fig. 6. Animation time comparison; note: y-axis is in log scale

For upload, where bandwidth is limited and the number of in-flight packets is small, the performance gap between NetAnim and UVNS is relatively small. However, for download where the number of in-flight packets is large, the performance difference is more than two orders of magnitude. This is because the animation for a packet transmission event is not optimized in NetAnim (for each packet transmission NetAnim always renders the packet position in five times), whereas it is in UVNS, mainly owing to the frame-based animation.

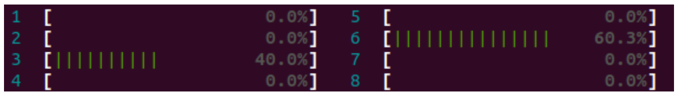
C. Miscellaneous

Other than the inefficiency caused by full event-based animation, we found through extensive experimentation that there are other more obscure reasons why NetAnim is slower than UVNS. First, for graphical user interface (GUI), NetAnim uses Qt5 [2]. Although NetAnim is programmed to support the minimum time gap of 1 μs between consecutive events, Qt only allows a gap of 1 ms. Due to this limitation, NetAnim cannot process more than 1,000 events per second at the maximum. Second, NetAnim is not programmed to use graphical processing unit (GPU). All rendering operations rely on CPU cores. In contrast, being a game engine, Unity actively uses GPU for rendering, offloading UVNS of the rendering computation. Furthermore, load management across CPU cores in NetAnim has room for improvement. For example, Fig. 7(a) shows that only a single CPU core out of sixteen is being exploited in the heavy load NTN scenario whereas two cores are employed in the relatively lighter TCP pacing scenario (Fig. 7(b)).



```
1  [                   0.0%]   5  [                   0.0%]
2  [                   0.0%]   6  [                   0.0%]
3  [|||||||||||||||||||96.0%]   7  [                   0.0%]
4  [||                 1.3%]   8  [|                  4.0%]
```

(a) NTN scenario



```
1  [                   0.0%]   5  [                    0.0%]
2  [                   0.0%]   6  [|||||||||||||||    60.3%]
3  [||||||||||        40.0%]   7  [                    0.0%]
4  [                   0.0%]   8  [                    0.0%]
```

(b) TCP pacing scenario

Fig. 7. CPU core utilization in NetAnim; other 8 cores are not utilized

D. Summary

From the discussions above, we argue that exploiting game engines for animating packet level simulation is highly desirable. The strength of UVNS that can handle large number of objects and events more efficiently is expected to become important in the most advanced communication scenarios such as high-speed networks (where large number of packet events play out at any instant), LEO constellations with a few thousand satellites, and vehicle-to-everything (V2X) communications where hundreds of vehicles typically participate in communication. In the future, we plan to expand the capability of UVNS to provide richer information on the logged simulation trace, while further maximizing the efficiency.

IV. Conclusion

Although ns-3 is a popular, versatile, and evolving simulation platform, there is much room for improvement in terms of its animation utility. This paper introduces an initial prototype of a new animation utility that is based on the Unity game engine platform. Owing to the inherent optimization features of the engine, large-scale network simulation is easily managed with high execution speed and animation quality.

Acknowledgments

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ICT Creative Consilience program (IITP-2023-2020-0-01819) supervised by the IITP (Institute for Information & communications Technology Planning & Evaluation).

References

[1] nsnam, "ns-3 network simulator," [Online]. Available at https://www.nsnam.org.
[2] "Qt Documentation Archives," [Online]. Available at https://doc.qt.io/archives/qt-5.5/.
[3] Unity, "Unity Download Archive," [Online]. Available at https://unity.com/releases/editor/archive.
[4] H. Lee, W. Yoon, H. Kim, "Unity-based Visualization of ns-3 Simulation," [Online]. Available at https://github.com/SeaweedAshes/UVNS.