VNetOS: Virtualised Distributed and Parallel Sensor Network Operating Environment for the IoT and SHM

Stefan Bosse
University of Bremen, Dept. Mathematics & Computer Science, Bremen, and Institute for Digitization, Bremen, Germany

Abstract — **Dealing with distributed and parallel computing in strong heterogeneous environments, e.g., distributed sensor networks, is still a challenge at the algorithmic, communication, and application levels. Heterogeneity is related to different computer and network (communication) architectures. Virtualization can hide and unify heterogeneity. Besides inter-process communication and synchronization, the unified access and monitoring of computing nodes (devices, computers, processors) is required to handle distributed and parallel systems in a comfortable and easy-to-access manner. Especially in education, the access and control of a large set of computing nodes is difficult, which lowers the learning curve significantly. In this work, a unified distributed and parallel framework and Web tools are introduced using Virtual Machines (VM) and Web browsers to control them. The framework enables the control, monitoring, and study of distributed-parallel systems, especially addressing sensor networks and IoT networks. Nodes can be arranged in a graphical drawing world or script-based. Virtual network nodes are assigned to VM instances that can be created inside the browser using Web worker processes or can be attached to externally running VM instances via a Web control API. New VM instances or processes can be started and controlled instantly. The graphical UI provides access to the internal and external nodes, programming editors, and monitor shells. The VMs can be generic, but in this work there is a focus on JavaScript and Lua. The framework provides augmented virtuality, i.e., a coupling of physical and virtual worlds.**

Keywords — Virtualisation; Virtual Machines; Distributed Data Processing; Sensor Networks; Internet of Things; Network Simulation; Augmented Virtuality

I. Introduction

Virtual network computing has its origins long ago in teleoperation [1]. The aim of virtual network computing is the split of application software and hardware. But communication was commonly limited to pure data exchange and well-defined Remote Procedure Calls (RPC) that could be requested by a thin client on a remote server. There are client processes and server processes that provide an operational service. Virtual networking itself is basically a mapping of physical communication graphs onto logical ones. The Internet is a prominent example, especially concerning IPv6 protocols that hide the physical location of computers. In [2], network virtualization has been proposed as a promising way to overcome the current limitations of the Internet by allowing multiple heterogeneous virtual networks (VNs) to coexist on a shared infrastructure. Studying distributed algorithms and monitoring and evaluating distributed systems is still a challenge, especially in education. The Web browser is today's expressive and powerful software to create network applications (Laboratory in the Web browser [3]) without the requirement of software installation and dependency conflicts.

Simulation of communication on different levels plays an important role in the development and evaluation of distributed algorithms, protocols, systems, and hardware components. Hardware-in-the-loop simulations couple simulated processes and devices with real devices. Examples include ns2 and ns3, as well as more recent frameworks such as Jist, a Java-based simulator, and SimPy, a Python-based simulator [4]. These simulators perform an abstraction of processes and devices.

In this work, Virtual Machines (VM) can be deployed everywhere and connected with a Web browser-based network control software framework. Additionally, most of the VMs can be processed in the Web browser software, too, either directly (any JavaScript-based VMs) or by using VM-in-VM implementations (like for Lua). The VNetOS is the software layer that provides operations and the components to merge internally and externally deployed VM instances. VM instances are processed in worker processes (either WebWorker, threads, or system processes) and can be accessed and controlled by the unified Network Management Protocol (NMP), supporting Web browsers, too. The Virtual Network Operating System (VNetOS) is a collection of protocols and software modules that enable the design and evaluation of large-scale heterogeneous computer networks with an easy-to-use and powerful GUI processed in the Web browser. Applications of the software framework are education, simulation, and real-world system control of embedded system networks. The next sections introduce the principle network and communication architecture, the software framework, a customized multi-computer based on a 5$ tiny embedded system directly controllable and programmable via the Web browser, and an evaluation.

II. Network Architecture

The general architecture consists of a generic network graph $G=\langle N, P, C\rangle$ with VM nodes N that can process a textual programming language L, a set of communication ports P attached to nodes, and communication connections (links) L between ports. JavaScript, e.g., can be processed directly in the Web browser or by an external VM like node.js. Each VM *vm* typically supports one control path γ and is executed in a dedicated worker thread or process. New VM instances can be created at run-time from a root node. External VM instances can be controlled by JSON-based Web Remote Procedure Call (RPC) services that can be accessed via HTTP, HTTPS, WebSocket (WS), or WSS IP protocols from the browser or other nodes. HTTP and WS without certificate-based security level are relevant for embedded systems since certificate management is not suitable for embedded systems. Instead, key-based encryption can be used for secure communication. Additionally, a worker that was forked from a root node can be controlled via the root node too, e.g., in the case the worker hangs in an endless loop. Because most VMs are strictly single-threaded, they cannot handle IO requests (such as signals or worker termination) and computation at the same time (at least not at a high level).

Firstly, three different VMs are considered in this work:

1. JavaScript (using V8/Spidermonkey/jerryscript [7] engines);
2. Lua (C-Lua, eLua, LuaJit [8], Fengari Web/VM-in-VM [6]);
3. JavaScript Agent Machine (JAM, programmed in JS, VM-in-VM, [5]).

Secondly, three different host computers are considered:

1. Generic desktop and mobile laptop computers (x86,x64, 2 cores, 2GHz clock) supporting all VMs;
2. Embedded system Raspberry PI (Zero, 3, Arm, 1-2 cores, 1GHz clock) supporting all VMs;
3. Tiny embedded system ESP32 (Tensilica, 2 cores, 240MHz clock) supporting Lua primarily and JS VMs secondarily.

A node provides control and generic communication ports for JSON-based RPC communication. Worker instances can access the node's communication port via a multiplexer. Each worker instance provides a control port, too. Internal nodes are connected via virtual connection links (VC) handled by the JS main loop. External nodes provide IP-based communication ports. All link pair combinations are possible: Internal-

internal, internal-external, and external-external.

Arbitrary network topologies can be created, including star and mesh grids, typically using generic IP-based protocols like HTTP or WebSockets (WS). The use of secured (SSL) connections is difficult due to certificate provision on each node, but it is fully supported. The graphical front-end (or any script-based network configuration) connects the nodes automatically (if the nodes are reachable).

There are two classes of VMs used in this framework:

A. A root meta VM that is the main process providing a Web RPC API to create and control worker processes;

B. The real target VM (JS, Lua, FORTH, ...) that is executed in a worker process, providing an RPC service (especially for IO), too.

Although, class A can execute the target programming language, too, only worker VMs are used for code processing. The main process as well as the worker processes execute their own VM instance. A worker can be an isolated operating system process (primarily in the case of JS and JAM) or a lightweight thread process (Lua). In both cases, socket-based communication channels and shared memory are used for inter-VM communication. The main VM is responsible for creating and controlling worker instances via NMP described in the next section. Remotely created workers live until they are explicitly terminated by the remote side or if the NMP connection of the remote side is dead.

Each node provides generic communication ports (aside from the RPC control service) that can be used by user programs to communicate between nodes. The communication ports can be linked ad hoc and provide JSON-based channels. Each port provides an IP port listener, except in the case of internal Web browser nodes, discussed in the software framework section.

III. Software Framework

The VNetOS software framework consists of the following parts:

1. The Web browser GUI application with a 2D graphical network world consisting of graphical node entities with communication links between nodes, code editors with syntax highlighting, process monitor and interactive shell windows, and external node controllers;
2. Internal VMs that can be embedded in the Web browser, i.e., can be provided in JavaScript or WebAssembly;
3. External VMs with a Web RPC service that have virtual shadow nodes in the Web GUI;
4. A set of programming modules supporting parallel and distributed programming (like CSP modelling, sensor access, RPC; for each target VM language there is an implementation).

The general architecture is shown in Fig. 1a with internal and external VMs, and generic communication links. The embedding of VMs in the Web browser is used for the simulation of networks, optionally connected with external nodes, or as part of the computational network, providing a coupling of phyiscal and virtual worlds (virtual augmentation). Some VMs like Lua or Python require VM-in-VM implementations.

A target VM is always processed in a forked worker process. All worker processes with independent VM instances communicate via (socket-based) channels with the main process.

Communication between the Web controller and external nodes is established via the Network Management Protocol (NMP). NMP is session-based (but loosely coupled with the reconnect feature without losing state) and enables input and output redirection and worker process control. Each virtual representation of an exter-

nal node in the Web GUI uses NMP to access external nodes and to create new VM instances (or agents in the case of JAM) and communication ports. NMP communication (e.g., polling for standard output and error streams from the VM) is event-based and dynamic to reduce the communication costs (which can be significant if the Web browser application is connected to hundreds of external nodes).

Generic data communication ports on external nodes as well as communication links between ports can be created via the Web browser application via NMP, too. A communication port on an external node with full network API (processed, e.g., by node.js) creates an IP listener that receives messages from other nodes. Internal nodes in Web browser workers cannot provide a listening port, but they can be connected to external nodes via push-pull communication (or by using bidirectional WebSockets), which is established from the browser side. Forwarding messages from internal to external nodes uses direct network requests, receiving messages from remote ports uses time-limited blocking network requests that are completed if there are messages for the browser port endpoint (except for WebSocket communication). WebSocket communication introduces a significant code overhead for HTTP(S) upgrades that are not available or suitable for tiny embedded systems. Therefore, HTTP(S) is primarily used. Communication between external nodes can use UDP/TCP ports, too. All VM worker instances created by the same root VM share communication ports. Each communication port is a local message multiplexer (CMUX), too, and messages sent from workers via a communication port are passed to the root VM process, which forwards the message to all registered connections and all other attached VM worker instances (see Fig. 1b). Depending on the underlying process model (threads versa system processes), the forwarding process (from worker to multiplexer and vice versa) can produce a significant extra overhead, as shown in the experimental section. The CMUX also implements IP-port forwarding between different nodes, including Web browser node communication. Each VM worker instance provides a port set mirror of the node's port set. If a port is accessed within a VM worker, the access is routed up to the parent VM and the message multiplexer (see Fig. 1b).

The 2D graphical world is a virtual view of the physical VM network and consists basically of graphical nodes (rectangles) associated with either internal or external VM nodes, e.g., a computer with a specific IP address. Each node representation is organised into slices. Each slice is assigned to a VM instance forked from the root node. Internal and external associations provide nearly the same operational set: Adding or removing code editors, creating worker instances with IO monitors, and interactive shells with IO monitoring. Finally, generic communication ports and links between ports can be added. The entire graphical network representation, including code, can be saved and loaded in JSON format. Links can be established between internal nodes (using virtual channels), between internal and external nodes, and between external nodes. Ports and links are controlled on remote nodes via NMP. NMP is a lightweight protocol using generic IP communication (HTTP, e.g.,) that can be implemented even on low-resource embedded systems.

Besides computational nodes, there are synthetic sensors and generic data sources that can be attached to internal nodes. These sensors inject data at specific time points or upon defined events. The data is either computed by an analytical function or consists of pre-recorded data. A Monte Carlo simulation can be applied to introduce noise and sensor model variances.

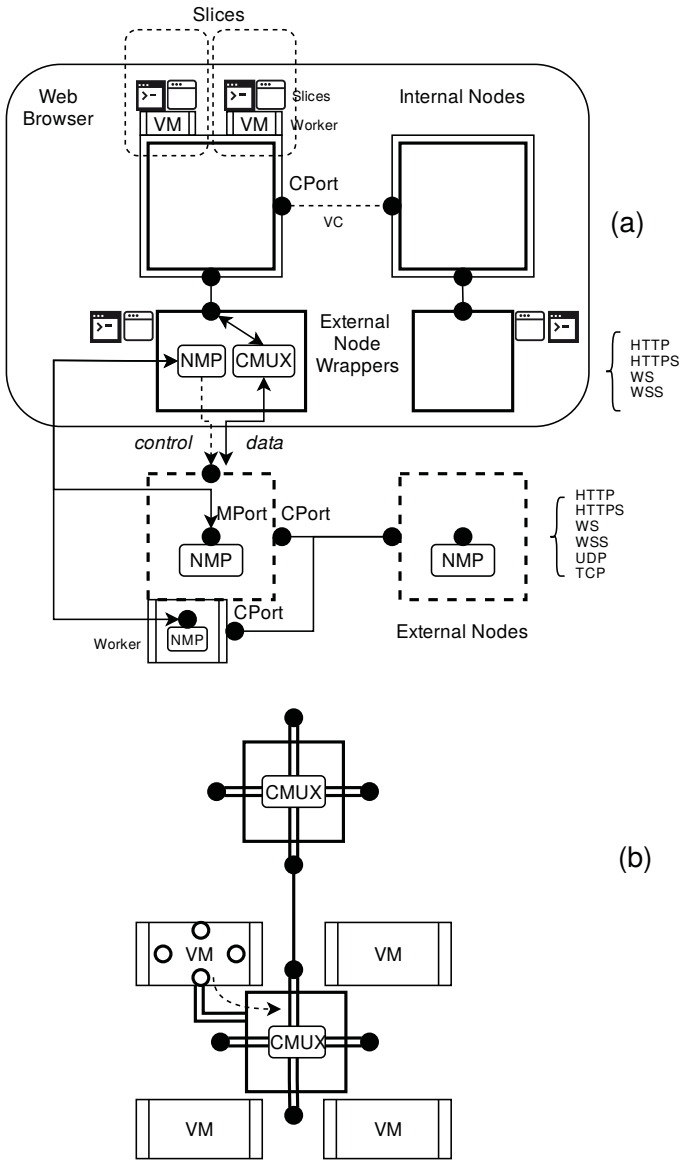

Figure 1. (a) General software framework and communication architecture with internal (double outline), external mapper (single outline), and external (dashed line) nodes. There are management communication ports (mPort) for connecting Web controllers with external nodes and generic communication ports (cPort) for inter-node communication (b) Message multiplexer architecture

To summarise, these different node classes are distinguished:

I-Node. Internal node with an embedded meta VM processed by the browser JS VM.
P-Node. External node processed by a native VM.
V-Node. Virtual wrapper (twin) of a p-node in the Web browser with NMP access and control.

IV. Multicomputer Platform

Besides generic computers, embedded systems play an important role in the investigation and education of heterogeneous networks, including the Internet-of-Thing domain. One of the embedded devices is the raspberry PI Zero microcomputer. It is running with a full operating system (Linux Raspberry OS) and network protocol stack, but it is mostly still a generic computer. One scale level down, the tiny embedded ESP32 computer is used. An ESP32 provides 512kB RAM and 16MB ROM with two Tensilica 32-bit processor cores. This is a pure application-specific device without an initial operating system and process shells.

Furthermore, single embedded systems connected by VNetOS provide only loosely coupled distributed communication. Parallel computation can be exploited by multicore or multi-computer devices. To study distributed-parallel (clustered) computation, a multi-computer was designed with a set of ESP32 tiny embedded devices. The network control of each single ESP32 computer by the Web browser application is performed via WLAN/WiFi by the Web application, but the interconnect of the computers is realized by high-speed serial links on the development board, shown in Fig. 2. An ESP32 has two freely usable UART devices with DMA capabilities and a maximal bit-rate of 5Mb/s. All ESP32 devices are connected to a central multi-port and parallel network switch implemented with a FPGA. Device-to-devices communication is handled by the switch. To increase the bandwidth and to lower the communication latency, two serial links for each device are used in parallel, as shown in Fig. 2.

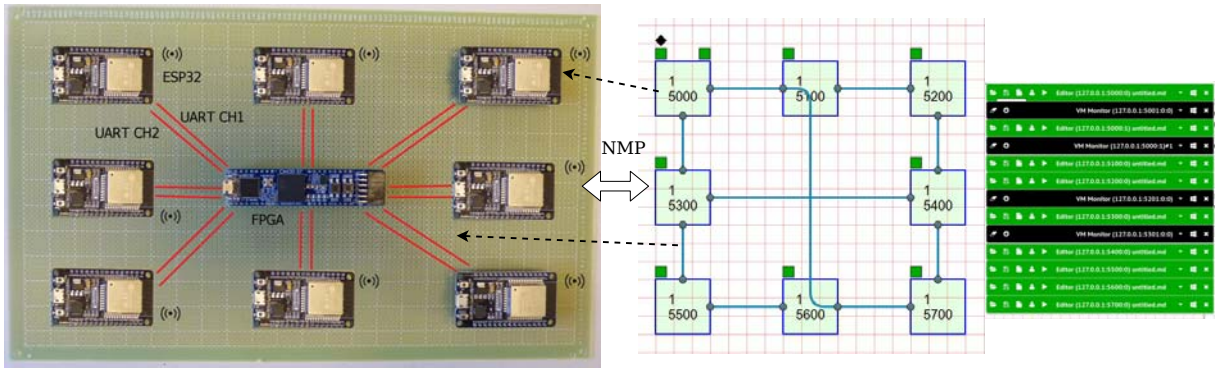

Figure 2. (Left) Prototype of the distributed-parallel multicomputer with 8 embedded ESP32 nodes and an FPGA-based internal network switch (Right) VNetOS GUI representation of the network (green box: editor, black rombus: VM instance monitor)

Each device implements parts of the FreeRTOS operating system in a pure functional API including WLAN/WiFi layers, serial communication, multi-threading, the node-level communication layer, and Lua VMs. Each Lua VM is processed in its own worker thread. The Lua VM (native C version, eLua) is booted with the required Web RPC service using NMP. The Web browser application can directly create and control Lua VMs on each device. Any logical device interconnection network structure created on the Web or by a

host application can be selected by the FPGA network switch (full N:N multiplexer with queuing). The ESP32 devices are programmed via the Arduino IDE with code transfer via USB. The programming of each node is only required once. The user code is passed to and started on the VMs via the Web browser controller.

V. Preliminary Experiments and Results

Three principle experiments were performed:

1. Network of 16/8 internal nodes arranged in a 2D mesh-grid;
2. Network of 16/8 external nodes arranged in a 2D mesh-grid;
3. Hybrid 8 internal + 8 external nodes.

An example configuration is shown in Fig. 3 with four internal and two external nodes (Raspberry PI Zero) connected via WLAN. For each VM instance there is a code editor and an IO monitor shell window. Internal and external nodes can communicate directly via HTTP.

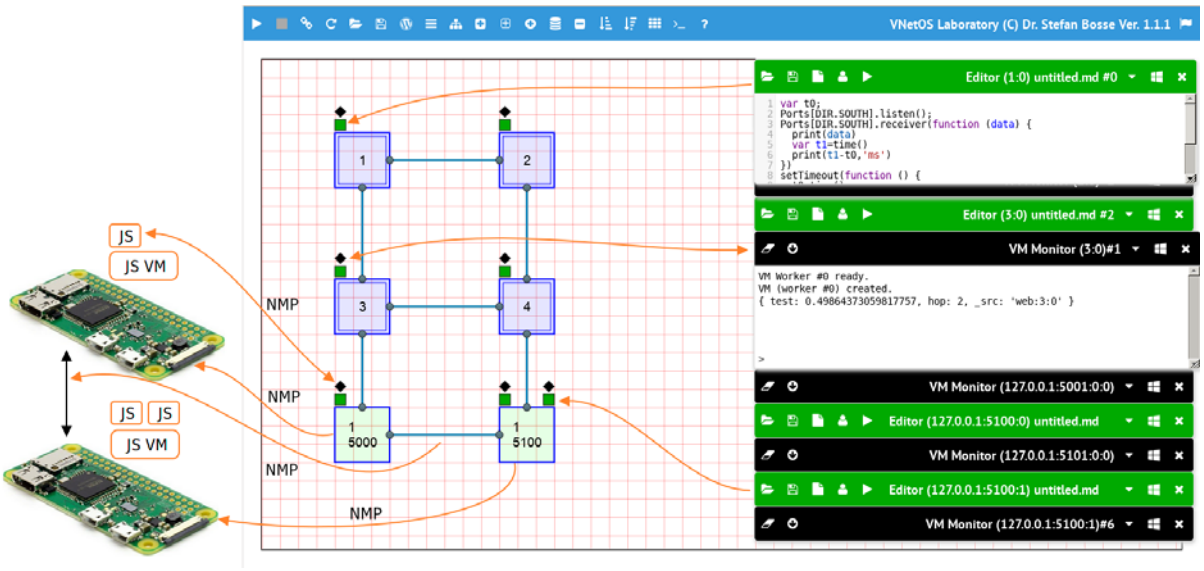

Figure 3. Typical network application using VNetOS, a Web browser, and two external Raspberry PI devices.

Host	dhry/s	VM	t_{iVM}	m_{iVM}	t_{cmsg}
PC/nodejs	5000k	JS (ext)	140ms	20MB	3ms
PC/Firefox	4200k	JS (int)	100ms	10MB	4ms
PC/plvm	600k	Lua, Parallel LuaJit(+libuv) (ext)	3ms	800kB	0.1ms
Raspberry PI Zero/nodejs	230k	JS	1600ms	20MB	40ms
Raspberry PI Zero/plvm	40k	Lua, Parallel LuaJit(+libuv) (ext)	10ms	800kB	1ms
ESP32/Lua	1k	Lua, FreeR-TOS (ext)	100ms	100kB	5ms

Table 1. Selected experimental results of VM performance (dhrystones measured by the VM)

Each experiment was carried out with JS and Lua VMs. The Lua VM, either Fengari Lua VM for Web browser or the Parallel Luajit VM (plvm), and JAM VMs are extensively used in education and lecture courses (bachelor and master courses). The evaluation of lectures with student exercises of both VMs and the VNetOS (and some predecessors like LuaNetOS and the JAM laboratory) showed a steep learning curve, and even students with low programming skills were able to program and evaluate distributed systems.

A dhrystone benchmark was performed for each target VM for normalized comparison. Performance results are shown in Tab. 1, which depends on the VM implementation, regarding both VM forking and messaging times between two nodes, t_{iVM} and t_{cmsg}, respectively. Lua can be easily embedded and forked using multi-threading, whereas node.js requires system process creation (at least some time ago), resulting in a instance creation time 100 times higher. Communication time is limited due to core bandwidth/latency and by the process/thread scheduling times required for message multiplexer invocation. Lua (LuaJit) shows superior performance compared to node.js/V8-based VMs and is a suitable VM for (tiny) embedded systems. The base memory requirement for each VM instance m_{iVM} is another important parameter, as shown in Tab. 1. Node.js (and Web browser engines) pose the highest start-up times and memory requirements, but also the highest computational power.

VI. Conclusion

A novel distributed virtualization framework for the deployment and control of heterogeneous networks of generic and embedded systems was introduced. The control of the distributed network is performed by a graphical Web browser application (or alternatively, script-based). Via the Web application, each node can be controlled by the NMP protocol. Each physical node has a virtual representation in the Web application (or any other script-based control software). The physical and virtual nodes are connected via NMP. Each root

node supports a programmable target VM (e.g., JS, Lua) and can instantiate (fork) VM worker processes. VM instances can be connected with each other by using generic communication ports. The routing of messages is performed by a message router. Evaluation of the node performance identified VM forking and message routing times as critical, but strongly dependent on the underlying VM (LuaJit forking is 100 times faster than node.js). Even tiny embedded systems can be used for distributed programming and processing. Besides education, simulation and generic distributed network control are core applications.

VII. References

[1] T. Richardson, Q. Stafford-Fraser, K.R. Wood, A. Hopper, 1998. Virtual network computing. IEEE Internet Computing, 2(1), pp.33-38.

[2] N.M.K. Chowdhury, M.R. Rahman, R. Boutaba, 2009, April. Virtual network embedding with coordinated node and link mapping. In IEEE INFOCOM 2009 (pp. 783-791). IEEE.

[3] S. Bosse, , PSciLab: An Unified Distributed and Parallel Software Framework for Data Analysis, Simulation and Machine Learning—Design Practice, Software Architecture, and User Experience , Appl. Sci. 2022, 12(6), 2887; 10.3390/app12062887

[4] J. Lessmann, P. Janacik, L. Lachev, D. Orfanus, 2008, April. Comparative study of wireless network simulators. In Seventh International Conference on Networking (icn 2008) (pp. 517-523). IEEE.

[5] S. Bosse, U. Engel, Augmented Virtual Reality: Combining Crowd Sensing and Social Data Mining with Large-Scale Simulation Using Mobile Agents for Future Smart Cities. Proceedings, Volume 4, ECSA-5 5th International Electronic Conference on Sensors and Applications 15–30 November, 2018 DOI 10.3390/ecsa-5-05762

[6] https://github.com/fengari-lua/fengari, online, accessed 1.6.2022

[7] https://github.com/jerryscript-project/jerryscript, online, accessed 1.6.2022

[8] https://luajit.org, online, accessed 1.6.2022