A Generic Emulator Interface for Digital Preservation

Towards a Collaborative Distributed Emulator Registry

| **Rafael Gieschke** | **Klaus Rechert** |
| --- | --- |
| *University of Freiburg**Germany**rafael.gieschke@rz.uni-freiburg.de**0000-0002-2778-4218* | *University of Applied Sciences Kehl**Germany**rechert@hs-kehl.de**0000-0002-2454-4374* |
|  |  |

**Abstract – Emulation frameworks as well as emulation as an access strategy have matured. With more simplified access to emulation and improved workflows, there is still a gap to be closed, primarily the availability of emulators and especially for smaller niches like arcade games or pre-PC computers. In this article we propose methods to simplify emulator preparation for framework integration as well as describing an emulator’s technical capabilities. Both, the technical design and the technical description of emulators will provide a foundation for cooperative work to identify, list, describe and integrate emulators of interest for the digital preservation community.**

**Keywords – Emulation, Metadata, Registry**

**Conference Topics – Innovation, Exchange**

# Introduction

With maturing emulation frameworks as well as wider acceptance and usage of emulation as a strategy within the digital preservation community, the availability of emulators as a crucial precondition and a future risk factor has become a focus of attention. Implementing emulators is a highly technical and complex task which requires significant development resources as well as expertise. Even though considered in the past [1], implementing emulators within the preservation community is currently – and for the foreseeing future – not a realistic options. The IT research and development budgets of memory institutions are already strained due to a variety of challenges posed by the ongoing digital transformation of public administrations and businesses. Furthermore, it is not sufficient to implement just a single emulator but a wide range of platforms, computer systems and use-case are to be supported to cope with the breadth of born digital content from different decades and types.

Fortunately, there is already a very lively emulator ecosystem outside of the preservation community. High quality emulators have been implemented for various purposes, e.g. by enthusiasts and hobby programmers for nostalgic reasons or with support of the (commercial) hardware and software industry to support development and testing. One of the main foundations and success factors of the bwFLA emulation framework [2] was to tap into that pool of emulators and make them accessible and usable for digital preservation workflows through a unified control interface. Continuously integrating further emulators for digital preservation is necessary to address a number of conceptual and technical challenges. Most obviously, increasing the variety of emulators is necessary to cover smaller niches from arcade machines, game consoles to the pre-IBM PC era of mainframes and early home computers such as the BBC Micro or the ZX Spectrum. While plenty of emulators are available as (open source) standalone software, the integration into emulation frameworks currently requires a so-called "wrapper" within the framework to adapt to the emulator’s control interface, i.e., how the emulator is configured and started, to allow integration into generic preservation workflows, e.g., open and render a document in an emulated software setup or simply play a computer game without additional knowledge about the emulator specifics. This additional adaptation work remains a significant hurdle. The already quite limited development capacity has been focused on high volume and high demand workflows and emulators. To improve this, broadening the developer base is of high importance, e.g., by simplifying the adaptation and integration tasks and decoupling the resulting emulator packages technically from the emulation framework, i.e., maintenance of the emulation framework and maintenance of emulator packages not only become independent tasks, but also preparation of emulator packages could be done without knowledge of internals of the emulation framework (e.g. EaaS).

A further problem regarding implementation, usage and maintenance of emulators is that there is currently no overview about emulators currently in use by the preservation community or what kind of emulators do exist and are of potential interest for the preservation community. Additionally, it is necessary to describe their technical capabilities and ideally verify these, e.g., based on relevant use cases. For instance, if there are multiple options for a given artifact, which emulator could be used or is recommended. Ideally, this information is not only shared within the preservation community but is also machine actionable such that tool support can be implemented to support search and automated workflows.

Finally, emulators are contemporary software and, like any other contemporary software, today’s emulators will eventually become obsolete. Preparations are to be made for that event, first and foremost, by knowing if an emulator requires a more recent substitute, which emulated hardware components precisely this substitute should provide, if the substitute is then likely to be compatible or which one is the best match even if not 100% compatible.

In this paper, we have identified, defined and implemented a generic emulator API, such that – ideally – any emulation framework is able to integrate and reuse this work. Furthermore, for emulators to be described technically, we propose extensible metadata using semantic web technologies, to build a shared corpus of machine actionable metadata for emulators.

# Related Work

Emulators for digital preservation, e.g., Dioscuri [1] or the more abstract Universal Virtual Computer (UVC) [3] have been proposed and developed with the idea to produce durable implementations of machines or machine code interpreters. In practice, this approach has failed due to the technical complexity of implementing emulators of rather complex computer systems. To achieve compatibility of software or operating systems, not only the architecture’s CPU (supporting its instruction set architecture) needs to be implemented but also a wide range of additional hardware (and peripherals) like graphic cards, sound cards, input devices, and more. Due to the fast technical progress of computer platforms, not only the emulators need to be constantly adapted to a new technical ecosystem but there are also new concepts and components to be integrated.

Packaging freely available emulators to become usable as tool for preservation purposes and to some degree portable has been implemented within the KEEP Emulation Framework (KEEP EF) [4], [5]. Emulators became available through a simple "emulator archive". The packaged emulators, however, had been setup to run locally on the user’s computer and thus, suffered from compatibility problems and reduced portability. Furthermore, the abstraction of the individual emulator control was rather limited. Abstraction of emulator controls, e.g., as a unified API is however a crucial precondition for the creation of complex workflows, automation, and interoperability. The rise of cloud computing led to a quite similar problem set. The lack of standards, e.g., every cloud provider using its own hypervisor (implementation or configuration) and, thus, hardware abstraction, made moving virtual machines from one provider to another difficult [6]. The DMTF Open Virtualization Format (OVF) defines and standardizes, amongst others, technical metadata describing the virtual machine’s hardware configuration [7]. libvirt[[1]](#footnote-0) is an example for a generic control API implementation for virtualization systems [8]. The main goal is to provide a stable interface for arbitrary hypervisors, implementing wrapper code (drivers) and an independent API front-end. We aim for a similar but more lightweight architecture abstracting specific control components (similar to the libvirt driver code) for disk, network, video, etc., components with the API endpoints being integrated within the emulator packages. For this, these approaches from the cloud context provide valuable guidance on an abstract level, however, these projects are targeted to complex, well documented contemporary systems and quite difficult to extend to a technically diverse emulation landscape. Additionally, preservation workflows are less focused on performance aspects, i.e., require less fine granular controls but a higher degree of abstraction to cover a wider variety of hardware components.

For the purposes of framework integration, developing an emulator package reaches beyond a container-based portable software setup (e.g. Docker image, a Snap package or similar). Technically, the emulator package must not only be self-contained but also be self-describing, such that emulators can be found and selected based on their technical capabilities, i.e.,the specific emulated hardware that is available and the potential hardware-related configuration options. With the KEEP metadata proposal [9] and later TOTEM registry and metadata schema [10] an initial tool-set for describing hardware and software became available. The proposed schema modeled hardware dependencies in a very detailed way but the schema has not yet found wider adoption yet mostly due to its static design. Entity fields were descriptive strings or user provided identifiers. This made an automated, tool-based generation, refinement, search and comparison of entities difficult. Digital preservation and in particular emulation-based workflows need to be highly automated to scale with breadth and amount of digital objects. There are now tools and concepts to identify file formats [11], software dependencies [12] and suggest software setups [13], but there is little to no support to identify the relevant technical platform, e.g., based on binary code analysis or operating system information, which technical platform (e.g. x86 PC, Apple PowerPC, Macintosh m68k, etc.) is required. The main reason so far is that there is no registry yet with documented available emulator hardware as well as documented connections of these hardware components to software, such as operating systems, libraries, drivers, or other digital artifacts.

With PREMIS v3.0 [14] the concept of environments has been introduced [15]. While environments can be described now in a rather flexible way, e.g. as an *Intellectual Entity*, the focus remains on descriptive elements, since hardware is considered as a physical object. Therefore, it is further necessary to resolve the ambiguity of emulators representing one or many abstract physical concepts as well as being a software object with versatile configuration options and potentially multiple hardware platforms. Relating for instance a software stack (operating system, rendering software) and an digital object to an abstract hardware object additionally requires the relation to an emulator and more specifically information which of the emulator’s configured hardware features are required. Describing the required hardware configuration and especially a "representation" in the form of a configured emulator remains difficult.

# Describing Emulators

In order to list, maintain and (re-)use emulators for digital preservation purposes, technical description of their capabilities are required. A practical and rather general description of an emulator might be sufficient to describe the fact that an emulator emulates one or multiple guest platforms, e.g., computer systems or a combination of computer and operating system. These emulated systems are composed of multiple (default) hardware components with additional configurable or optional hardware elements. For instance, a typical computer setup always contains a specific CPU model but the CPU model is not sufficient to define or describe a specific computer system. A specific CPU might be used by multiple platforms, e.g., the MOS Technology 6502 CPU has been used in such diverse platforms as, among many more, the Apple II, the Commodore VIC-20, as well as the original Nintendo Entertainment System. A description at a computer system or platform level implies a specific set of components, e.g., a game for the Commodore VIC-20 will expect both the platform’s MOS Technology 6502 CPU as well as its VIC graphics processor to be present and will not run on a Apple II using the same CPU.

For practical purposes, configurable or optional hardware components are of greater interest, e.g., the presence of a cassette/tape drive or a floppy drive of the VIC-20. A game might run from one or the other. This is even more apparent when using PC sound cards like the Sound Blaster 16 or the AdLib Music Synthesizer Card as games might very well only be compatible with one or the other. For an emulation setup, it is thus far more important – and easier, since these options are explicitly exposed by the emulator software as configurable elements – to describe configurable or optional components than to describe implicit or default components of a given platform. Even if in some cases an optional component might be required, e.g., an emulated PC will generally have to have at least one (optional) storage device to be able to load any software, it can still be regarded as optional in the sense of not being implied by the emulated platform.

We therefore propose technical emulator metadata able to describe the emulator’s supported platforms (e.g., QEMU is able to emulate the IBM PC platform, certain types of Apple Power Mac systems, and many more) and for each of these platforms the configurable components the emulator is able to emulate. The platform’s detailed description, i.e., all required hardware components, and thus, elements an emulator has to implement to support a given platform, can be outsourced to public knowledge bases like Wikidata, DBpedia, or similar. Configurable and optional hardware components can be of different types. While emulated platforms differ greatly from each other, all of them share similar concepts like storage devices, input devices, video devices, audio devices, other output devices and possibly network devices. Some of these devices will have associated properties, e.g., a storage device has an inserted medium whereas an (Ethernet) network card has a MAC address.

## Describing Hardware Components

As a first step towards a machine actionable emulator description, the (optional) components of an emulated platform have to be assigned an identifier so that they can be referenced from and included into emulation environments. While from a technical perspective, it would be enough to enumerate "component 1", "component 2", etc., additional descriptive and technical metadata about the components is crucial not only for users to able to make an informed choice about which optional components are necessary in an emulation environment but also for automated or assisted selection of a suitable emulator, comparing emulators, finding substitutes for emulates and others.

For some hardware components, the community already has collected machine-readable linked open data describing their properties, e.g., the NE1000 network card[[2]](#footnote-1), which is a member (a subclass) of the more general concept "network card"[[3]](#footnote-2) and of which Wikidata provides an image. Hence, it might be tempting to describe hardware emulated by the emulator directly using, e.g., entity URLs from Wikidata. However, this would not really be accurate as, e.g., QEMU does not provide a NE1000 network card directly but a specific implementation thereof. Other emulators might implement the NE1000 network card slightly differently such that there are cases in which it is not compatible with the QEMU implementation, e.g., QEMU might emulate a NE1000-compatible card connected to the ISA bus, while other emulators might connect it to the PCI bus. Furthermore, it might later be discovered that the emulated network card is not really a NE1000-compatible but actually a NE2000-compatible card.

Keeping existing environment metadata stable in this case while adding additional knowledge makes it necessary to describe the emulation environments using a two-step approach: The only guaranteed (and verified) information at creation time of emulation environments is the exact chosen configuration of the emulator package. While the developer of an emulator package might not understand initially all consequences of options passed to the emulator will have to the final hardware configuration, e.g., will the QEMU option -hda create an IDE or AHCI drive, will its storage controller be attached via PCI, the developer can assign a (unique) identifier to every device type corresponding to a configuration option of the emulator it supports. It is only important at this step to assign (at least one) unique identifier to every component the developer of the emulator packages wishes to expose. Identifiers should then stay stable over future versions of the same emulator package, i.e., the same identifier should reference the same emulated component. We expect this condition to be feasible as the source code of the emulator package is available and only updated for new emulator versions. For emulators themselves, developers or maintainers of emulator packages might have to scan the changelog or release notes for changes regarding, e.g., default options. We expect that this can be supported by, potentially automatically, trying to run existing emulation environments with the new version of the emulator package. For example, if the QEMU option -hda were to change from emulating an IDE drive to emulating an AHCI drive, existing emulation environments containing Microsoft Windows would stop to run completely, which could easily be spotted. The existing component with its existing identifier would then have to pass further command-line options to QEMU and a new identifier could be introduced for the new default behavior. Linking these identifiers to, e.g., real-world devices can then be done as a second step, possibly at a later time, without modifying existing environments. In the same way, wrong assumptions about emulated devices (does QEMU really emulates a 80486 CPU for the IBM PC platform by default) can be corrected retroactively.

## Example

Figure 1 shows the possible description of the QEMU emulator in an emulator package. It is able to emulate both the IBM PC as well as the Power Mac platform. For the IBM PC platform, the emulator package can emulate a hard disk drive and a network interface card. For the Power Mac platform, it can only emulate a hard disk drive. In both cases, this does not necessarily mean that QEMU itself is not able to emulate any other devices or platforms but that the emulator package only exposes (yet) the described devices using the generic API. The metadata is further enhanced with knowledge not directly used by the proposed API, e.g., a human-readable title for the platforms and devices, the information that the emulated hard disk drive uses (Wikidata property P2283) the PATA interface (Wikidata entity Q230360) in case of the emulated IBM PC and the SCSI interface (Q220868) in case of the Power Mac, or the fact that QEMU emulates a PowerPC G4 CPU (Q430856) as implicit (non-optional) component for the platform by default. The non-optional, implicit information is not necessary for starting and interacting (technically) with the emulator but might be very useful to determine if an emulation environment using this emulator package could potentially also run on another emulator, i.e., an emulator also emulating a PowerMac with a PowerPC G4 CPU and a SCSI hard disk drive.



Figure 1 Emulator description.

Of equal importance, is that the metadata could be used in much broader applications, e.g., querying the list of all emulator supported computer systems produced by Apple or, the other way round, determining if any relevant computer systems or optional components are still missing from being supported. It is important to stress that, as with all semantic data, providing the information in a machine-readable format facilitates all kinds of potential uses of the data, many of which were not yet anticipated. Allowing to enhance the emulator description with custom metadata is made possible by use of JSON-LD as an established metadata format as well as assigning (referenceable) identifiers (i.e., URIs in the form of URLs) to each distinct component an emulator package supports.

## Emulator Configuration

While the emulator metadata describes which components the emulator is able to emulate, an emulator configuration describes which components have been selected and should be emulated to represent and instantiate a specific *environment*. We re-use the term environment with similar semantics as in PREMIS to describe fully configured computer setup. For example, a user might want to combine a QEMU IBM PC configured with a Sound Blaster 16 and a NE2000 network with a hard disk drive containing a software setup including Windows 95 as its operating system together with additional software installed as well as an (emulated) optical drive with an ISO image attached. The configuration can only reference concrete platforms and components defined in the respective emulator description and, generally, not arbitrary emulator options, intentionally reducing the number and granularity of supported options. This two-step approach allows to both decouple the emulator configuration from the emulator and, thus, improve its maintainability but also to describe emulation environments in a generic format, allowing the emulation framework to concentrate on this generic format instead of having to include specific knowledge about each emulator.

Figure 2 shows a further example of a configured QEMU emulator emulating an Apple Power Mac with a hard disk drive and a CD-ROM drive. The chosen JSON-LD format again allows the surrounding emulation framework to record additional information about the configured components, e.g., the data source of the hard disk drive. For the CD-ROM drive, no data source is present and it could be assumed to only be present without an ISO image attached. The "index" property is proposed as a generic way to signal in which order an emulator package should add the respective emulated devices to the emulated computer. This order might have an effect, e.g., on the Microsoft Windows operating system family for the assignment of "drive letter" to hard disk and optical drives. It is hoped to be more versatile than, e.g., inventing platform- and interface-specific ways to assign drives to individual ports as these ways would have to be developed for each emulator individually and most probably not be interoperable anyway.

The properties "path", "nativeConfig", and the "frameworkComponents" are technical properties that form part of the interface between the emulation framework and the emulator package and are described in more detail in the next section. "path" allows to provide an associated path, e.g., the path at which the emulation framework will provide the emulator the data, e.g., a disk image. "frameworkComponets" are components that are technically required to access the output of the emulator (i.e., the emulation software) but are not emulated components of the target emulation environment. "nativeConfig" is a simplification for users to provide (non-semantic) unstructured additional configuration options to an emulator package in order to support experiments with the emulator package. When using any "nativeConfig" properties, the user, however, cannot expect the emulation environment to be compatible with any future versions of the emulator package or be interoperable with any other emulators.



Figure 2 Emulator configuration.

To simplify configuration work, emulator configuration templates, i.e., preconfigured and tested combinations of concrete components for a given emulator and platform are useful. These are identical to emulator configurations but, e.g., in case of hard disk drive, leave out configured data sources. They can support users in choosing a sensible start configuration, e.g., for a given target operating system, like including a Sound Blaster 16 and a NE2000-compatible network card in an emulator configuration template targeting Windows 95. This can also be target operating-system specific, as, e.g., while the Linux’s kernels default x86 32-bit configuration has support for NE2000-compatible network cards (a popular hardware from the early x86 era), its default x86(-64) 64-bit configuration only has support for Intel E1000-compatible network cards (a popular hardware from the x86-64 era). Templates could be developed by users independently of the emulator package and also provide the equivalent of default components for a given context, e.g., a default sound card for Windows 95. Templates further help to minimize the number of different configurations of a given emulator package and thereby easing future migrations to another emulator, where only very few configurations would have to be tested to have confidence that all emulation environments created from the same emulator configuration template remain working. On the contrary, it is much easier and more elaborate to collect such "default device" knowledge outside of the emulator package, e.g., also taking into account specific operating systems.

# Encapsulating Emulators

It is not sufficient for emulator packages to be self-describing, they also need to be self-contained and self-executing, i.e., the package has to include all necessary parts to execute the emulator and be able to translate the generic API to the concrete API of the emulator so that emulation frameworks only need to speak one generic API.

To be able to preserve emulators and be able to run them independently of the host operating system, an emulator and all of its dependencies have to be encapsulated. The general goal is to have a very high forward compatibility: an existing emulator package should (in the existing version) continue to work for as long as possible. If changes are necessary nonetheless, it is preferable to require the same changes for all emulator packages and not require to maintain and adapt every emulator package individually.

One could ensure this, e.g., by archiving the emulator’s source code and compiling it on the host platform as soon as the emulator is used. While this might seem to be useful for guaranteeing independence from the host platform, in practice, the approach is not really feasible, due to both time needed for compilation as well as fragility of the compilation setup in regard to external libraries, used compiler versions and host platform. An emulator which was originally implemented for the x86 architecture, when ported e.g., to the ARM architecture, will usually not be usable without code adaptations. Thus, the most feasible approach is to package and archive binaries of the emulator and all of its dependencies, so that you will be able to at least run the emulator in exactly the same version again in future. For future maintenance, it generally is advisable to also archive the emulator’s source code, for which there are already existing initiatives, which can be relied upon.[[4]](#footnote-3) A generally accepted way to package application in a self-contained package are Dockerfiles[[5]](#footnote-4) and Docker images. The latter are being standardized as OCI Image Format[[6]](#footnote-5) and only include compiled binary files, while a Dockerfile is able to describe the packaging process in a human readable as well as machine actionable format. Even though emulators should be replaced with equivalent contemporary implementations for performance, security and user-convenience reasons, packaging and preserving emulators as containers allows to resurrect these for reference purposes [16].

Besides the emulator and its dependencies, the emulator package also has to include a component implementing the generic API and speaking to the emulator. While the generic API is intentionally designed as simple as possible, so that this component can be implemented in a broad variety of programming languages and its implementation can also evolve over time without breaking existing emulator packages, a template implementation in JavaScript is provided as a basis for new emulator packages.[[7]](#footnote-6)

## Control API

The used Linux containers as defined by the Open Container Initiative (OCI) Runtime Specification[[8]](#footnote-7) provide a solid encapsulation of executed programs. This ensures that executed emulators can only access resources, e.g., files, inside their own container and not communicate with outside system services provided by the host. In the case of emulators, this especially includes graphical input and output, sound, and network. The main primitives provided by the OCI Runtime Specification to communicate with the world outside of the container are the initial processes’s standard input/output streams and files or directories explicitly shared between the container and the host system (technically through so-called "bind mounts"). Every intended way of interaction with the emulator thus has to be implemented using these primitives. Incidentally, this also helps with forward compatibility as it reduces the used interfaces which have to be maintained, preserved and maybe re-implemented drastically.

The component translating from the generic API will take the configuration format described in the previous section and, generally, pass them to and start the emulator using command-line arguments or by generating emulator-specific configuration files. While, e.g., a sound card can run independently, for some categories of devices external data sources might have to be provided, e.g., a hard disk drive might need a disk image. The emulation framework has to prepare and provide this data to the container, e.g., by using the path property for hard disk drives to specify where the emulator package can find the respective disk image in the form of a raw file and, using the OCI Runtime Specification, share this file with the container at the configured path.

During execution of the emulator, further interaction with the emulator inside the container is needed. The most basic ways of interacting with emulators that have been identified for this work are keyboard and mouse input, video and sound output, and network access. We have evaluated each of these areas individually and tried to find a suitable and stable (standardized) interface.

### Video Output

For video output, the most basic interface would be a frame buffer, i.e., a serialization of the pixels shown on an emulated computer screen, starting at the upper left pixel of the first displayed line and ending at the lower right pixel of the last displayed line, using a fixed number of bytes, e.g., 3 bytes with one byte for each of the pixel’s red, green, and blue component. We argue that a frame buffer offers a higher forward compatibility than more pre-processed serializations of video like, e.g., an H.264 video stream.

At the same time, it typically puts no big computational burden on the emulator package to produce a frame buffer, which will typically be the first step in producing any video output anyway, and not have to further process it. Additionally, due to encapsulation, this post-processing would have to be done in generic software without any acceleration possibly provided by individual host systems. It is thus much preferable to have access to the most basic video output (i.e., a frame buffer) from the emulator. This video output can then be further post-processed by the individual host system with its individual host-specific acceleration capabilities, e.g., encoding it to a compressed H.264 video stream to be sent to a user’s web browser.

A frame buffer alone, though, is more like a generic abstraction and not already a technical realization. While it could be realized by constantly updating a file shared between host and container with the current content of the emulated display or sent as byte-stream through a socket or network, the technical characteristics need to also be communicated. For instance, each pixel could be represented by 2, 3 or 4 bytes per pixel. Furthermore, the dimensions of an emulated computer screen might be fixed and known in advance, for many platforms it can also change during execution, causing the size of the frame buffer to change as well.

With forward compatibility in mind, we propose that the video protocol should be a protocol that is common, well understood, and ideally has a wide range of actively maintained implementations as well as tool support. Potential options include X11[[9]](#footnote-8), Microsoft’s Remote Desktop Protocol (RDP)[[10]](#footnote-9), the Remote Framebuffer Protocol (RFB)[[11]](#footnote-10) used by various VNC programs, and Wayland[[12]](#footnote-11). Evaluating these protocols, RDP does not enjoy wider Open Source tool support and, as a proprietary protocol, it is likely to be more difficult to maintain and support over time. We currently consider Wayland as not fully mature, such that forward compatibility of the current state is not ensured. We still expect that Wayland matures quickly and will become a viable candidate in the future. The RFB protocol is a high-level protocol also offering different video-codecs. Its "Raw Encoding" would be equivalent to the aforementioned framebuffer but few emulators offer direct RFB support, requiring an additional active component inside the emulator container capturing the emulator's output and producing a RFB stream. X11, a very well established and very mature (though somewhat complex) protocol, is already supported by almost all software with graphic output within the Linux/Open Source ecosystem including emulators. It supports access to the frame buffer of an application from any other application, and optionally, by directly using shared memory. While this is nowadays sometimes considered as a problem regarding security, for our purposes it is helpful to access any application output from outside the container. Security is not an issue in this specific case since a separate X11 server is deployed in each container and the container is already separated from all other containers as well as the host system. Additionally, X11 offers native support for accessing the X11 server using UNIX domain sockets represented as (special) files in the file system. These can easily be shared and accessed by the host system.

### Input Events

In order to allow users to interact with an emulated guest, emulated input devices need to be connected to the user’s contemporary input devices. All aforementioned protocols not only offer transport options for the emulator’s video output but also the ability for input events. In case of X11, the XTEST extension[[13]](#footnote-12) allows sending relative and absolute pointer coordinates and keyboard input to the emulator. For keyboard input, sending "keycodes" based on a key’s physical location as well as (indirectly) sending "keysyms" based on a key’s meaning is supported. This is relevant as an operating system in an emulator will typically allow users to configure keyboard layouts itself and thus passing through the location of a key as opposed to its (irrelevant) meaning on the host system is desirable, e.g., the letter "Z" typed on a "QWERTY" keyboard would be interpreted as a "Y" on a "QWERTZ" keyboard and a "W" on an "AZERTY" keyboard. For RFB, allowing to send keys based on their location is only possible using an extension[[14]](#footnote-13), making X11 a more desirable protocol for keyboard input independently of video output as well.

### Audio Output

For sound output, popular options used by emulators within the Linux ecosystem include the Advanced Linux Sound Architecture (ALSA)[[15]](#footnote-14), PulseAudio,[[16]](#footnote-15) and PipeWire.[[17]](#footnote-16) While ALSA is a rather low-level interface mainly used as interface to the Linux kernel, PulseAudio shares many of the useful characteristics of X11, in particular, exposure via a UNIX domain socket as a file in the file system. We consider PipeWire to be in a similar state as Wayland, quite promising but still not matured enough yet. Thus, we chose to use PulseAudio for sound output.

### Network

For network access, one has to differentiate between access to the Internet used by the emulator itself (e.g., an emulator might support downloading content from the live Internet) and the emulator emulating a network interface card to the guest platform. The former is not necessarily desirable as the goal of encapsulating the emulator is to eliminate any dependencies on external sources. By enabling network namespaces, the OCI Runtime Specification already covers this by forbidding access to the Internet. Managing network traffic from an emulated network interface card is handled separately. Currently, the only relevant network type supported by any emulators – if there is support for network – is Ethernet, which has to be accepted from and sent to the emulator, preferably via shared (special) files exposed within the local file system. We have chosen the Virtual Distributed Ethernet (VDEv2)[[18]](#footnote-17) [17] project to provide such an abstraction of Ethernet. Particularly, its vdeplug library defines a simple format, i.e., prefixes each Ethernet frame with its length as 16-bit big-endian integer, to turn Ethernet frames into a stream, which can then easily be tunneled over many types of transport. [18]

### Emulator Control

During execution of an emulator, users might want to change its configuration, e.g., changing the media of an emulated CD-ROM drive. Like the initial configuration, this request has to be translated into specific actions passed to the emulator, e.g., QEMU’s QMP monitor. The emulator package thus has to accept generic requests and turn them into emulator-specific requests during runtime.

As easily in a wide range of languages implementable control protocol, we propose a JSON-RPC[[19]](#footnote-18) based protocol sent over the container’s initial process’s standard input/output streams. An example to change the media in an emulated CD-ROM drive can be seen in Figure 3.

Conveniently, the same protocol can be used to transfer both the initial configuration with the request to start the emulator as well as any events originating from the emulator, e.g., a notification that an emulator has exited, errored, or a virtual power button was pressed, back to the emulation framework.



Figure 3 Control protocol.

## Re-usable Implementation

The proposed architecture deliberately is as independent from a surrounding framework as possible. It only requires to run Linux containers according to the OCI Runtime Specification, to share relevant files and directories with the host system. It does not predefine any special control files but relies on sending requests to the container’s initial process via its standard input. For the emulator’s input and output, existing and widely used protocols are used. We hope that the provided emulator packages can thus be useful and re-usable for new and future implementations of emulation frameworks.

The external emulation framework has to provide access to the exposed input and output, e.g., keyboard input and video output, to the user. It can use, e.g., Xpra[[20]](#footnote-19) to allow users access from their web browser.

We propose to use GitHub as a collaborative platform for collecting information about emulators, turning them into emulator packages, and maintaining them. A common "emulation-archive" project facilitates discovery of existing emulator packages, which can exist as one repository per emulator. A template repository can be provided, which can be forked as a basis for new emulator implementations.

# Conclusion

In this article we laid out the technical foundations to improve portability and re-use of emulators. Machine actionable technical metadata will support further automation, interoperability and eventually, a more sustainable emulation infrastructure. But most importantly, the technical design as well as the extensible technical descriptions of emulators provide a framework for cooperative work to identify, list, describe and integrate emulators of interest for the digital preservation community. We have deliberately chosen an open, collaborative way as the emulator landscape is quite scattered as is the detailed knowledge about less common computer systems.

# REFERENCES

[1] J. R. van der Hoeven, R. Verdegem, and B. Lohman, “Emulation for digital preservation in practice: The results,” *Int. J. Digital Curation*, vol. 2, pp. 123–132, 2007.

[2] K. Rechert, I. Valizada, D. von Suchodoletz, and J. Latocha, “BwFLA - A functional approach to digital preservation,” *Prax. Inf.verarb. Kommun. (PIK)*, vol. 35, no. 4, pp. 259–267, 2012.

[3] J. R. van der Hoeven, R. J. van Diessen, and K. van der MEER, “Development of a universal virtual computer (uvc) for long-term preservation of digital objects,” *Journal of Information Science*, vol. 31, no. 3, pp. 196–208, 2005.

[4] B. Lohman, B. Kiers, D. Michel, and J. van der Hoeven, “Emulation as a business solution: The emulation framework,” in *Proceedings of the 8th international conference on preservation of digital objects (iPRES 2011)*, 2011, pp. 167–170.

[5] W. Bergmeyer, “The keep emulation framework.” in *Proceedings of the 1st international workshop on semantic digital archives (sda 2011)*, 2011, pp. 8–22. Available: http://ceur-ws.org/Vol-801/paper1.pdf

[6] S. Ortiz Jr, “The problem with cloud-computing standardization,” *Computer*, vol. 44, no. 7, pp. 13–16, 2011.

[7] DMTF System Virtualization, Partitioning, and Clustering Working Group, “Open virtualization format specification.” Document DSP0243, 2010.

[8] M. Bolte, M. Sievers, G. Birkenheuer, O. Niehörster, and A. Brinkmann, “Non-intrusive virtualization management using libvirt,” in *2010 design, automation & test in europe conference & exhibition (date 2010)*, 2010, pp. 574–579.

[9] D. Anderson, J. Delve, and D. Pinchbeck, “Toward a workable emulation-based preservation strategy: Rationale and technical metadata,” *New review of information networking*, vol. 15, no. 2, pp. 110–131, 2010.

[10] J. Delve, L. Konstantelos, A. Ciuffreda, and D. Anderson, “Documenting technical environments for posterity: The totem registry and metadata schema,” *PIK-Praxis der Informationsverarbeitung und Kommunikation*, vol. 35, no. 4, pp. 227–233, 2012.

[11] A. Brown, “Automatic format identification using PRONOM and DROID,” *Digital Preservation Technical Paper*, vol. 1, 2006.

[12] K. Thornton, E. Cochrane, T. Ledoux, B. Caron, and C. Wilson, “Modeling the domain of digital preservation in wikidata.” 2017.

[13] J. Giessl, R. Gieschke, K. Rechert, and E. Cochrane, “Automating the selection of emulated rendering environments for born-digital data-sets,” in *Linking theory and practice of digital libraries - 25th international conference on theory and practice of digital libraries*, 2021, vol. 12866, pp. 106–111.

[14] PREMIS Editorial Committee, “PREMIS data dictionary for preservation metadata, version 3 (library of congress, november 2015).”

[15] A. Dappert and A. Farquhar, “Digital preservation metadata practice for computing environments,” in *Digital preservation metadata for practitioners: Implementing premis*, A. Dappert, R. S. Guenther, and S. Peyrard, Eds. Cham: Springer International Publishing, 2016, pp. 129–138.

[16] K. Rechert, T. Liebetraut, D. Wehrle, and E. Cochrane, “Preserving containers - requirements and a todo-list,” in *Digital libraries: Knowledge, information, and data in an open access society - 18th international conference on asia-pacific digital libraries, ICADL 2016,*, 2016, vol. 10075, pp. 225–230.

[17] R. Davoli, “Vde: Virtual distributed ethernet,” in *First international conference on testbeds and research infrastructures for the development of networks and communities*, 2005, pp. 213–220.

[18] R. Geischke, K. Rechert, and S. Mocken, “Preserving access to web servers. A case study preserving output of collaborative research centers,” 2021.

1. libvirt Virtualization API, https://libvirt.org/ [↑](#footnote-ref-0)
2. Novel Network Card NE1000, http://www.wikidata.org/entity/Q1959993 [↑](#footnote-ref-1)
3. "Network Card" Wikidata entry, http://www.wikidata.org/entity/Q165233 [↑](#footnote-ref-2)
4. e.g., Software Heritage source code archive, https://www.softwareheritage.org/ [↑](#footnote-ref-3)
5. Docker builder reference documentation, https://docs.docker.com/engine/reference/builder/ [↑](#footnote-ref-4)
6. OCI Image Format Specification, https://github.com/opencontainers/image-spec/blob/main/spec.md [↑](#footnote-ref-5)
7. For a preliminary example, see https://gitlab.com/emulation-as-a-service/experiments/qemu-ld/-/blob/main/emulator.js [↑](#footnote-ref-6)
8. Open Container Initiative (OCI) Runtime Specification, https://github.com/opencontainers/runtime-spec/blob/main/spec.md [↑](#footnote-ref-7)
9. Currently maintained by the X.Org project (https://gitlab.freedesktop.org/xorg/proto/) [↑](#footnote-ref-8)
10. https://docs.microsoft.com/en-us/openspecs/windows\_protocols/ms-rdpbcgr/5073f4ed-1e93-45e1-b039-6e30c385867c [↑](#footnote-ref-9)
11. Specified in RFC 6143 (https://www.rfc-editor.org/info/rfc6143) and currently maintained by the community (https://github.com/rfbproto/rfbproto/blob/master/rfbproto.rst) [↑](#footnote-ref-10)
12. https://gitlab.freedesktop.org/wayland/wayland/-/tree/main/protocol and https://gitlab.freedesktop.org/wayland/wayland-protocols [↑](#footnote-ref-11)
13. https://gitlab.freedesktop.org/xorg/proto/xorgproto/-/blob/master/specs/xextproto/xtest.xml [↑](#footnote-ref-12)
14. QEMU Extended Key Event Message [↑](#footnote-ref-13)
15. https://alsa-project.org/ [↑](#footnote-ref-14)
16. https://www.freedesktop.org/wiki/Software/PulseAudio/ [↑](#footnote-ref-15)
17. https://pipewire.org/ [↑](#footnote-ref-16)
18. https://github.com/virtualsquare/vde-2 [↑](#footnote-ref-17)
19. https://www.jsonrpc.org/ [↑](#footnote-ref-18)
20. https://xpra.org/ [↑](#footnote-ref-19)