# MetaContainer: decentralized resource orchestration

Michał Zieliński

**Abstract**

MetaContainer is a capability-based decentralized resource orchestration protocol. It aims to provide a common interface for sharing various types of resources, including filesystems, Ethernet networks or USB devices. MetaContainer also provides compute services (container orchestration) with the ability to seamlessly attach remote resources.

## 1 Introduction

In past years there were not many attempts at making resource sharing homogeneous. The most notable system was Plan 9 from Bell Labs [9], all of whose resource were files. Unfortunately, it was never deployed, nevertheless it influenced many modern operating systems. A more recent system is Redox [11], where all resources are represented by URLs. However, these designs require using a new operating system kernel, which makes them very hard to apply in practice.

In this paper, we present MetaContainer, a capability-based decentralized resource orchestration protocol that aims to provide a common interface for sharing various types of resources, including filesystems, USB devices and Ethernet networks. MetaContainer also provides compute services[1] (container orchestration) with the ability to seamlessly attach remote resources.

The reference implementation of MetaContainer protocol runs on unmodified Linux kernel.

Many objects on Unix systems can already be shared via network. For example:

- file systems can be shared via NFS (Network Filesystem) or 9pfs [1]
- Ethernet devices can be shared by bridging them to GRE [12] tunnels
- block devices can be shared using NBD [8] or iSCSI [14]
- audio devices can be shared using RTP [13]
- USB devices can be shared using USB-IP [16]
- desktops can be shared using VNC or RDP
- serial ports can be shared using socat [15]

[1]We use term “compute services” to refer to services that make it possible to remotely execute code on other machines. The term comes from IaaS terminology (e.g. OpenStack Compute Service)

However there is no unified interface for sharing these resources and each requires using different utilities with distinct configuration syntaxes and security properties.

MetaContainer aims to solve these problems by providing secure unified interface for sharing these resources.

Similarly, there are multiple applications that make compute services available via network. For example, OpenStack (an IaaS software) provides centralized control of virtual machines and their storage and network. Another application, Kubertenes implements scheduling and execution of containers on multiple hosts. However, existing container and VM orchestration systems assume that all machines managed by them are owned by a single entity and are relatively homogeneous.

SSH (Secure SHell) is another utility that provides a way to execute code on remote machines. Its permission model is coarse grained and SSH itself only provides rudimentary means of connectivity. MetaContainer compute services are conceptually similar to SSH, but are integrated with the rest of its resource sharing capabilities. The compute services allow starting both virtual machines and containers.

In this paper:

- a *service* refers to a software module that runs in a separate process
- a *virtual machine* refers to a software that emulates whole computer system
- *containerization* refers to a lightweight virtualization method that partitions systems resources while sharing operating system kernel. A *container* is a single partition in such system
- a *byte stream* refers to a communication channel which can be used to send sequences of bytes between two entities
- *piping* refers to an act of copying data from one byte stream to another

## 2 The protocol

### 2.1 RPC and capabilities

The MetaContainer protocol represents objects using capabilities. It uses Cap'n Proto [2] RPC protocol which is modelled after E distributed programming language [7]. A *capability* is a transferable and unforgeable reference to an object together with an associated set of access rights. The concept of capabilities is well described in [3].

We chose to represent resources as capabilities, rather than forgeable strings such as URLs, because we believe capability-based security is conceptually simpler and more powerful than ACL-based (access control list) security.

We chose to represent resources as RPC interfaces, rather than plain files (as in Plan 9). While using plain files is simpler to implement and makes porting to new languages trivially easy, it also leads to creation of ad-hoc RPC and serialization protocols in places when byte stream abstraction is not enough.

Cap'n Proto RPC protocol is only used on the control plane, data plane is implemented using protocols appropriate for the resource being shared, on top of bare TCP and UDP connections.

### 2.2 Backplane—the network layer

In order to simplify the application layer, MetaContainer pushes some work into network layer[2]:

- MetaContainer assumes that all hosts participating in the protocol are directly reachable using IP protocol and
- MetaContainer requires an IP network which prevents address spoofing and provides confidentiality.

These two requirements are often not fulfilled by traditionally configured IP networks. Using some type of overlay network is the most convenient way to obtain these properties. For example:

- ZeroTier SDN [18] with 6PLANE addressing
- CJDNS mesh network [4]
- WireGuard VPN [17]
- LAN with switch enforced IP addressing
- public IPv6 network with IPsec

## 3 The data plane

For every supported resource, MetaContainer defines a Cap'n Proto interface. Possessing a capability implementing this interface is equivalent to possessing the resource. We postpone the description of how the capabilities of remote resources can be obtained to Chapter 5.

### 3.1 Streams

The basic type of resource is a bidirectional byte stream. Its interface is a simple wrapper around TCP, which ensures that only authorized peers may connect.

The `Stream` interface has the following methods:

- `tcpListen(remote :Metac.NodeAddress, remotePort :Int32)-> (local :Metac.NodeAddress, port :Int32, holder :Metac.Holder)`

  Requests the node owning the stream to listen accepting exactly one TCP connection from `remote:port`. Connections from other addresses will be dropped. Caller of this method should bind port `remotePort` before making this call—this ensures that no one unauthorized will be able to connect.

[2]One could argue that well designed network protocol should provide confidentiality and direct connectivity. Current deployments are not well designed, because of backwards compatibility. It is then natural and elegant to fix this problem with an overlay network.

- `bindTo(other :Stream)-> (holder :Metac.Holder)`

  Pipes all data between this and `other` streams. Returns holder object representing this action – the streams will be bound to each other as long as this object is kept alive.

## 3.2 Block devices

A *block device* is a fixed-sized byte array supporting random access. MetaContainer uses Network Block Device protocol to access block devices over network. We chose it instead of more popular iSCSI, because of its simplicity.

Block devices can be used, for example, as virtual disks for VMs or containers started by the compute services.

The `BlockDevice` interface has the following methods:

- `nbdSetup()-> (stream :Stream)`

  Request Network Block Device connection, returns reference to a `Stream` representing this connection.

## 3.3 Filesystems

*Filesystem* object represents files or directories. Filesystem object can be, for example, mounted on local directories.

MetaContainer uses v9fs (Plan 9 Filesystem [1]) to access filesystems remotely. We chose it instead of more popular NFS, because it operates on plain TCP connections, is much simpler and behaves better on high latency links.

The `Filesystem` interface represents a directory and has the following methods:

- `getSubtree(name :Text)-> (fs :Filesystem)`

  Returns a subtree of this filesystem. `name` may include slashes, but may not contain symbolic links. This procedure must be implemented carefully to avoid TOCTOU (Time of Check, Time of Use) attacks. Reference implementation uses `openat` syscall with `O_NOFOLLOW` flag.

- `getFile(name :Text)-> (file :File)`

  Returns file object by name. It doesn't need to exist, the object only represents a path in this filesystem. Again, `name` may contain slashes, but may not contain symbolic links.

- `v9fsStream()-> (stream :Stream)`

  Opens this filesystem using v9fs (also called 9p).

The `File` interface represents a regular file and has the following methods:

- `openAsStream()-> (stream :Stream)`

  Opens the file for reading as a stream.

- `openAsBlock()-> (device :BlockDevice)`

  Opens the file as a block device.

## 4 Network devices

`L2Interface` represents a layer 2 network interface (e.g. physical Ethernet port). We chose to tunnel layer 2 (Ethernet) ports, because they can be easily attached to virtual machines and L2 mechanisms (such as bridging) generally require less configuration than L3 (IP) mechanisms.

VXLAN is used to tunnel this L2 traffic over the network. VXLAN is the industry standard for tunneling L2 traffic over IP networks.

`L2Interface` has the following methods:

- `setupVxlan(remote :Metac.NodeAddress, dstPort :UInt16, vniNum : UInt32)-> (local :Metac.NodeAddress, srcPort :UInt16, holder : Metac.Holder)`

  Sets up a unicast VXLAN connection between `remote` and node hosting this L2Interface. `dstPort` and `srcPort` should be unique and not used for any other UDP communication—this is achieved using iptables `owner` module in the reference implementation. `vniNum` is used as an additional 24-bit secret value (this is not strictly necessary, but does no harm).

- `bindTo(other :L2Interface)-> (holder :Metac.Holder)`

  Pipes all traffic between this and `other` interface.

## 5 Persistence

### 5.1 Services

Capabilities are normally obtained by making method calls on a previously owned capabilities. First capability in a program needs to be obtained in a different way. MetaContainer defines two bootstrap capabilities. `NodeAdmin` can be obtained by `root` users by connecting to an Unix socket (`/run/metac/*/socket`)—this capability grants access to everything running on this host. `Node` can be obtained by anyone by connecting to TCP port `901`.

All functionality is implemented as separate services (e.g. `fs`, `network` or `vm` services)—this approach enables modularity and privilege separation. `ServiceAdmin` capabilities provide bootstrap interfaces for specific services e.g. for `fs` service, the capability implements a interface with `rootFilesystem` method which returns `Filesystem` object for the root (`/`) directory.

The `NodeAdmin` is a bootstrap interface for superusers and it has the following methods:

- `getServiceAdmin(name :Text)-> (service :ServiceAdmin)`

  Returns admin bootstrap interface for a service named `name`.

- `registerNamedService(name :Text, service :Service, adminBootstrap :ServiceAdmin)-> (holder :Holder)`

  Registers a new named service. The bootstrap interfaces can be later retrieved using `getServiceAdmin`.

- `getUnprivilegedNode()-> (node :Node)`

  Returns unprivileged bootstrap interface for this node (the same that can be obtained by connecting on TCP port `901`).

The `Node` is a bootstrap interface for unprivileged users. It has the following methods:

- `address()-> (address :NodeAddress)`

  Returns canonical address of this node.

- `getService(name :Text)-> (service :Service)`

  Returns unprivileged bootstrap interface of service named `name`.

## 5.2 The `Persistable` interface

The lifetime of Cap'n Proto capabilities is limited to the duration of the RPC connection. It is often useful to serialize capability as a string (e.g. in order to save it to database or print in human readable form). The serialized form of capability is called a *sturdy reference*.

`Persistable` interface is implemented by objects that can be have a sturdy reference created. It has a single method that creates an unguessable reference to this object:

`createSturdyRef(rgroup :Metac.ResourceGroup, persistent :Bool)-> (id :Metac.MetacSturdyRef)`

If `persistent` is true, then the reference should be valid even across reboots.

The `ResourceGroup` interface represents a resource group object that can be used to implement accounting. The interface does not have any methods, objects can be used by an implementation of `Persistable` in an arbitrary way.

For example, in a hypothetical service offering MetaContainer services to multiple users, each user could be associated with a resource group. When user removes his account, all sturdy references associated with his resource group can be freed.

`MetacSturdyRef` is an unguessable (possibly persistent) capability identifier. It has the following fields:

- `node` – address of a node this capability resides on
- `service` – name of a service storing this capability (it typical scenario, this is `persistence` service, described in next section)
- `objectInfo` – arbitrary data used for restoring the capability

The `Service` interface is implemented by all services. It has a single method that restores a `SturdyRef` associated with this service:

`restore(objectInfo :AnyPointer)-> (obj :AnyPointer)`

### 5.3 The persistence service

Requirements for `createSturdyRef` for all services are similar. To avoid code duplication, MetaContainer contains `persistence` service which handles creation and maintenance of sturdy references.

The default `persistence` service uses a Sqlite database for storing persistent capabilities. It it possible to create different implementations of this service, storing data, for example, in a fault tolerant database.

The bootstrap interface for the persistence service defines method

`getHandlerFor(serviceName :Text)-> :ServicePersistenceHandler`

Client services use it to request a persistence handler at startup.

`ServicePersistenceHandler` has the following methods:

- `registerRestorer(restorer :Restorer)`

  Client service should call `registerRestorer` immediately after startup. The persistent service will use the passed restorer instance to restore saved capabilities by calling its `restoreFromDescription(description :CapDescription)-> (cap :AnyPointer)` method.

- `createSturdyRef(group :Metac.ResourceGroup, description : CapDescription, persistent :Bool, cap :AnyPointer)-> (ref : Metac.MetacSturdyRef)`

  Requests a sturdy reference for a given capability. This method is used by services to implement `createSturdyRef` method of `Persistable` interface.

  Description is a structure that represent information necessary to restore the capability:

  - `description.runtimeId` – unique identifier for `cap` (used to avoid saving one capability multiple times)
  - `description.category` – type of this capability (transparently passed to the restorer)
  - `description.description` – arbitrary description used to restore the capability after reboot (transparently passed to the restorer). The description may contain references to other capabilities, they will be persisted automatically. This feature (recursive persistence) makes it easy to persist complex object hierarchies across multiple machines.

  `persistent` should be true if this capability should persist restarts.

  `group` is a resource group, unused by the current reference implementation.

## 6 Command line interface

Apart from Cap'n Proto interfaces, which can be used in languages supported by Cap'n Proto (Python, Java, C++, JavaScript, Go), MetaContainer provides command line interface (CLI).

CLI is provided by the `metac` binary. Each type of resource is handled by a separate command (`metac fs`, `metac net` etc). These commands provide more subcommands on their own. The most general subcommand is `export`, which creates a sturdy reference to a given resource. Most subcommands have `--persistent` flag, which, when enabled, causes the effect of the command to persist reboots (e.g. by toggling `persistent` argument to `createSturdyRef` to `true`).

The commands often operate on URIs that describe the resource. The exact form of URIs depends on the service, but `ref:` URIs, describing a sturdy reference, can almost always be used. For a given `MetacSturdyRef` $x$ they are in the following form: `ref://[x.node]/[x.service]/[serialized x.objectInfo]`.

CLI interfaces for individual services are described in later section. The following example shows the general design of the command line interface.

- `$ metac fs export local:/mydir`
  `ref://[fdca:ddf9:5703::1]/MQEaAftm...AB7eJsg`
  Retrieves sturdy reference of the local path `mydir`.
- `$ metac fs export --persistent ref://[fdca:ddf9:5703::1]/MQEaAftm...AB7eJsg`
  `ref://[fdca:ddf9:5703::1]/Qjr1jPQD...ktjffu2`
  Re-exports the sturdy reference, making it persistent between reboots.
- `$ metac fs mount local:/mydir /mnt`
  Is analogous to `mount --bind /mydir /mnt`.
- `$ metac fs mount --persistent ref://[fdca:ddf9:5703::1]/MQEaAftm...AB7eJsg /mnt`
  Mounts remote filesystem specified by a sturdy reference, persisting the mount between reboots.

## 7 Compute services

There are three types of compute services provided by MetaContainer:

- 'raw' virtual machines
- containers based on Linux Containers
- containers based on virtual machines

Raw virtual machines are comparable to services provided by libvirt or Hyper-V. Both types of containers share the same API and are comparable to Docker [6].

### 7.1 Virtual Machines

Virtual machine service is responsible for spawning and managing virtual machines. `VMLauncher` represent a capability to launch virtual machines. It's implementation defined where the machines are launched. MetaContainer currently is only able to launch VMs on a single host.

`VMLauncher` contains method that launches virtual machines:

`launch(config :LaunchConfiguration)-> (vm :VM)`

The `LaunchConfiguration` object contains configuration of the virtual machine. This configuration contains capabilities introduced in previous sections (such as block or network devices). A list of all fields of `LaunchConfiguration` follows:

- boot (type: `Boot`): boot method of the virtual machine. It is possible to either boot from a disk (`boot.disk` specifies a disk number) or directly boot a Linux kernel (`boot.kernel` of type `KernelBoot`):

  ```
  struct KernelBoot {
    kernel @0 :File;
    # Kernel binary
    initrd @1 :File;
    # initrd file containing initial filesystem
    cmdline @2 :Text;
    # Kernel parameters
  }
  ```

- memory (type: `Int32`): memory allocated for the VM, in MiB
- vcpu (type: `Int32`): number of CPU cores allocated for the VM
- networks (type: `List(Network)`): list of attached networks. Each item in this list specifies a single network card attached to the virtual machine.

  ```
  struct Network {
    # Attaches a network device.

    enum Driver {
      virtio @0;
      e1000 @1;
    }
    driver @0 :Driver;
    # Driver for the network card.

    network @1 :L2Interface;
    # The network.
    # If null, new network will be created, later available via '
       VM.network' method.
  }
  ```

- drives (type: `List(Drive)`): list of attached drives.

  ```
  struct Drive {
    enum Driver {
      virtio @0;
      ide @1;
    }
    driver @0 :Driver;
    # Driver for the drive.

    device @1 :BlockDevice;
    # The block device for this drive.
  }
  ```

- serialPorts (type: `List(SerialPort)`): list of attached serial ports.

  ```
  struct SerialPort {
    # Attaches a serial port.
  ```

```
    enum Driver {
      default @0;
      virtio @1;
    }

    driver @0 :Driver;
    # Serial port driver

    name @1 :Text;
    # Name of the port (only useful for virtio driver)

    nowait @2 :Bool;
    # Do not wait for connection to this port. May discard initial
         output.

    stream @3 :Stream;
    # The stream connected to this serial port.
    # If null, new stream will be created, later available via 'VM
        .serialPort' method.
  }
```

`VM` interface represents a running virtual machine. It has the following methods:

- `destroy()`

  Destroys the virtual machine.

- `serialPort(index :Int32)-> (stream :Stream)`

  Returns a stream for serial port number `index`.

- `network(index :Int32)-> (l2interface :L2Interface)`

  Returns `L2Interface` for network number `index`.

`network(index :Int32)` and `serialPort(index :Int32)` duplicate functionality of `Network.network` and `SerialPort.stream`. This duplication is intentional – user can pass result of e.g. `serialPort(0)` from one VM to `SerialPort.stream` of another VM, effectively joining two serial port with a virtual cable.

## 7.2 Process launcher

The container services are provided by `ComputeLauncher` capability. It is responsible for launching new processes. It can be thought as a generalization of Python's `subprocess.Popen`. The configuration is separated into environment configuration and process configuration. The main goal of this separation is to make it possible to spawn new processes in existing environments (namespaces), for example, to aid debugging.

`ProcessEnvironmentDescription` structure describes an environment for multiple running processes. The environment includes filesystem and networks. `ProcessDescription` structure describes an environment for a single process.

`ComputeLauncher` contains method that launches a process in a new process environment:

`launch @0 (processDescription :ProcessDescription, envDescription : ProcessEnvironmentDescription)-> (env :ProcessEnvironment, process : Process)`

`ProcessEnvironmentDescription` has the following fields:

- filesystems (type: `List(Mount)`): list of filesystems to mount.

  ```
  struct Mount {
    path @0 :Text;
    # Where to mount this filesystem? Use '/' for root filesystem.

    fs @1 :Fs.Filesystem;
    # The filesystem.
  }
  ```

- networks (type: `List(NetworkInterface)`): list of networks to attach

  The `NetworkInterface` structure, apart from L2 network interface (`l2interface`), contains IP configuration. This is needed, so spawned process can use network without any further configuration.

  ```
  struct NetworkInterface {
    name @0 :Text;
    # Name of the network interface.

    l2interface @1 :Network.L2Interface;
    # If null, new network will be created, later available via '
        ProcessEnvironment.network' method.

    struct Route {
      network @0 :Text;
      # Address of the network (e.g. 192.168.0.0/24)
      via @1 :Text;
      # Gateway. If null, direct route will be created.
    }

    addresses @2 :List(Text);
    # List of IP address for this interface.

    routes @3 :List(Route);
    # List of routes exiting via this interface.
  }
  ```

- memory (type: `UInt32`): memory allocation for this environment (in MiB).

`ProcessDescription` has the following fields:

- args (type: `List(Text)`): command arguments, including executable name (e.g. `["cat", "/etc/passwd"]`)
- files (type: `List(FD)`): list of open Unix file descriptors

  ```
  struct FD {
    isPty @0 :Bool;
    # Should the file descriptor be opened as a virtual terminal?

    stream @1 :Stream;
    # The stream. If null, new stream will be created, later
        accessible via 'Process.files' method.
  }
  ```

- uid, gid (type: `UInt32`): user and group identifier for this process.

`Process` represents a running process. It has the following methods:

- `file(index :UInt32)-> (stream :Stream)`

  Returns stream for file previously specified in `files` field of `ProcessDescription`.

- `kill(signal :UInt32)`

  Kills the process with `signal`

- `returnCode()-> (code :Int32)`

  If the process has finished, returns its return code.

- `wait()`

  Waits for the process to finish.

`ProcessEnvironment` represents an existing process environment. It has the following methods:

- `launchProcess(processDescription :ProcessDescription)`

  Launches a new process in this environment. This is analogous to `ComputeLauncher.launch`.

- `network(index :Uint32)-> (l2interface :L2Interface)`

  Returns network specified in `ProcessEnvironmentDescription` with index `index`.

# 8 Resource services

This chapter contains a list of services currently specified as a part of the MetaContainer protocol.

## 8.1 The filesystem service

The filesystem service exposes whole filesystem of the node and makes it possible to mount remote filesystems.

The `FilesystemServiceAdmin` has a single method that returns a root filesystem namespace:

`rootNamespace()-> (ns :FilesystemNamespace)`

`FilesystemNamespace` interface has the following methods:

- `filesystem()-> (fs :Filesystem)`

  Returns a filesystem for the root of this namespace.

- `mount(path :Text, fs :Filesystem)-> (holder :Holder)`

  Mounts a filesystem in this namespace at path ‘path’.

Command line interface:

- `file-uri` is an URI in the form:
  - `local:path` – representing a file or a filesystem on local path `path`
  - `ref:XXX` – representing a sturdy reference
- `metac fs export [--persistent] {file-uri}` – create a sturdy reference to a given filesystem
- `metac fs mount {file-uri} {path}` – mount a given filesystem at `path`
- `metac file export [--persistent] {file-uri}` – creates a sturdy reference to a given file
- `metac file cat {file-uri}` – copy given file to stdout

## 8.2 The network service

Network service exposes the network namespace on the node. It allows creation and exporting of kernel interfaces.

`KernelInterface` represents an existing kernel network interface and is an extension of `L2Interface`. It has the following methods:

- `getName()-> (name :Text)`

  Returns name of this interface.

- `rename(newname :Text)`

  Changes name of this interface.

- `destroy()`

  Destroys interface.

- `l2Interface()-> (iface :L2Interface)`

  Returns L2Interface associated with this kernel interface.

`KernelNetworkNamespace` represents a kernel network namespace. It has the following methods:

- `listInterfaces()-> (interfaces :List(KernelInterface))`

  Returns a list of network interfaces existing in this namespace

- `createInterface(name :Text)-> (iface :KernelInterface)`

  Create a new kernel interface. Before this call there should be no existing interface named `name`.

Command line interface:

- `net-uri` is an URI in the form:
  - `local:devname` – representing existing local device named `devname`
  - `newlocal:devname` – representing a new, not yet existing, local device named `devname`
  - `ref:XXX` – representing a sturdy ref
- `metac net export [--persistent] {net-uri}` – create a sturdy reference to a given network device

- `metac net bind {net-uri} {net-uri}` – binds two network devices (e.g. `metac net bind ref:XXX newlocal:remote0` imports remote device as `remote0`)

## 8.3 The USB service

The USB service allows attachment of remote USB devices. The `UsbDevice` interface represents a single USB device. It has the following methods:

- `info()-> (info :Info)`

  Returns information about this device (product ID, bus ID)

- `usbIpStream()-> (stream :Stream)`

  Opens USB-IP stream to this device.

`UsbDevices` interface represents a collection of USB devices. It has the following methods:

- `listDevices()-> (devices :List(UsbDevice))`

  Lists all USB device connected to this host.

- `getDeviceByProductId(productId :Text)-> (device :UsbDevice)`

  Returns device with this `productId`. The object will be returned even if there is no currently connected device or there are multiple devices with the same product ID. This behavior makes the returned interface resistant to USB re-enumerations (which change bus IDs).

Service interface `UsbServiceAdmin` defines the following methods:

- `attach(device :UsbDevice)-> (holder :Metac.Holder)`

  Attaches a new USB device to this host.

- `usbDevices()-> (devices :UsbDevices)`

  Returns a collection of USB devices attached to this this host

Command line interface:

- `usb-uri` is an URI in the form:
  - `local-busid:busid` – local device with bus ID `busid`
  - `local-productid:productid` – local device with product ID `productid`
  - `ref:XXX` – representing a sturdy ref
- `metac usb export [--persistent] {usb-uri}` – create a sturdy reference to a given network device
- `metac usb attach {usb-uri}` – attach a remote USB device to the current host

## 8.4 The sound service

The sound service allows streaming of sound between machines. The `SoundDevice` interface represents a sound device—either a sink (speaker) or a source (microphone). It has the following methods:

- `info()-> (info :SoundDeviceInfo)`

  Returns information about this device

  ```
  struct SoundDeviceInfo {
     name @0 :Text;
     # Name of this device

     isHardware @1 :Bool;
     # Is this real device?

     isSink @2 :Bool;
     # Is this a sink device (as opposed to a source)?
  }
  ```

- `opusStream()-> (stream :Stream)`

  Returns stream outputting/accepting audio in OPUS format. OPUS is used, because it is designed to handle real-time audio compression.

- `bindTo(other :SoundDevice)-> (holder :Metac.Holder)`

  Pipes sound between two sound devices

`SoundServiceAdmin` has a single method `getSystemMixer()-> (mixer :Mixer)` returning the system mixer. `Mixer` interface represents a software sound mixer (e.g. PulseAudio [10] instance). It has the following methods:

- `createSink(name :Text)-> (source :SoundDevice)`

  Creates a new sink. Return source which emits sound played on this sink.

- `createSource(name :Text)-> (sink :SoundDevice)`

  Creates a new source. Audio played on the returned sink will be emitted on this source.

- `getDevice()-> (devices :List(SoundDevice))`

  Returns list of currently connected devices.

Command line interface:

- `sound-uri` is an URI in the form:
  - `localsink:id`, `localsource:id` – local device with PulseAudio ID `id`
  - `newlocalsink:name` – new local sink device named `name`
  - `newlocalsource:name` – new local source device named `name`
  - `ref:XXX` – representing a sturdy ref
- `metac sound export [--persistent] {usb-uri}` – create a sturdy reference to a given sound device
- `metac sound bind {sink-uri} {source-uri}` – bind a sink to a source

### 8.5 The remote desktop service

A *desktop* is a resource that represents a screen together with mouse/keyboard inputs. A desktop capability can be obtained for existing X11 sessions. `Desktop` session could also be obtained for running virtual machines. It has the following methods:

- `vncStream @0 ()-> (stream :Stream)`

  Opens a VNC connection to this desktop.

`DesktopAdmin` bootstrap interface has the following methods:

- `getDesktopForXSession(num :Int32)-> (desktop :Desktop)`

  Retrieves a desktop for a X11 session running on this machine.

Command line interface:

- `desktop-uri` is an URI in the form:
  - `local-x11:N` – representing a local X11 session
  - `ref:XXX` – representing a sturdy reference
- `metac desktop attach {desktop-uri}` – start a graphical VNC client attached to a given remote desktop
- `metac desktop export [--persistent] {desktop-uri}` – create a sturdy reference to a given desktop

## 9 The implementation

The reference implementation of the MetaContainer protocol is available https://github.com/zielmicha/metac. It is written in Nim programming language. Many, but not all, services are implemented:

- filesystem service (`metac/fs.nim`, `metac/fs_*.nim` files)
  - QEMU NBD server is used for sharing block devices
  - diod v9fs server is used for sharing filesystems
- network service (`metac/network_service.nim`)
  - in-kernel VXLAN implementation is used
- VM service (`metac/vm.nim`)
  - QEMU hypervisor is used for running the virtual machines
- compute service, VM version (`metac/computevm_service.nim`)
- sound service (`metac/sound_service.nim`)
  - PulseAudio is used as a mixer
  - OPUS codec is used for audio compression

### 9.1 Implementation of VM-based containers

We implemented `ComputeLauncher` for virtual machines in the `computevm` service. It works by spawning a virtual machine with a predefined Linux kernel and initial filesystem (“initrd”).

The initrd contains an agent program that communicates with the `computevm` service over dedicated network interfaces. The responsibilities of the agent program include mounting filesystems and setting up the environment in general.

### 9.2 Implementation of LXC-based containers

LXC-based `ComputeLauncher` will be implemented in the `computelxc` service. It will use Linux namespacing capabilities to implement lightweight containers.

## 10 Conclusion and further work

In this paper we have defined MetaContainer protocol – a homogeneous method for sharing resources. The protocol and implementations still can be improved, for example by adding:

- Distributed sturdy references: the sturdy references could be kept in a distributed, fault tolerant database like etcd [5]. The `MetacSturdyRef` structure could be changed to enable usage of multiple addresses, in order to accommodate the fault tolerance.
- Scheduling of containers: automatic scheduling and supervision of containers running on multiple hosts
- Tahoe-LAFS is a capability-based distributed filesystem that nicely fits MetaContainer model of capabilities. It could be wrapped in a MetaContainer service in order to support truly distributed storage.
- Automatic sturdy reference rotation: it is good practice to periodically rotate secret values such as sturdy references.
- More types of resources: there are obviously more types of resources than can be shared, such as videos, windows, smartcards or layer 3 networks.

## References

[1] 9P: The Simple Distributed File System from Bell Labs, available at: http://9p.cat-v.org/. [Retrieved: 11 Jun 2017].

[2] Cap'n Proto cerealization protocol, available at: https://capnproto.org/. [Retrieved: 11 Jun 2017].

[3] Cap'n Proto: RPC Protocol, available at: https://capnproto.org/rpc.html. [Retrieved: 11 Jun 2017].

[4] CJDNS mesh network, available at: https://github.com/cjdelisle/cjdns. [Retrieved: 11 Jun 2017].

[5] CoreOS etcd, available at: https://coreos.com/etcd/. [Retrieved: 11 Jun 2017].

[6] Docker: Build, Ship, and Run Any App, Anywhere, available at: http://docker.com. [Retrieved: 11 Jun 2017].

[7] E Open Source Distributed Capabilities, available at: http://www.erights.

org. [Retrieved: 11 Jun 2017].

[8] Network Block Device, available at: https://nbd.sourceforge.io/. [Retrieved: 11 Jun 2017].

[9] Plan 9 from Bell Labs, available at: https://pdos.csail.mit.edu/archive/6.824-2012/papers/plan9.pdf. [Retrieved: 11 Jun 2017].

[10] PulseAudio on FreeDesktop, available at: https://www.freedesktop.org/wiki/Software/PulseAudio/. [Retrieved: 11 Jun 2017].

[11] Redox: Your Next(Gen) OS, available at: http://www.redox-os.org. [Retrieved: 11 Jun 2017].

[12] RFC-1701 Generic Routing Encapsulation.

[13] RFC-1889 RTP: A Transfer Protocol for Real-Time Applications.

[14] RFC-3347 Small Computer Systems Interface protocol over the Internet (iSCSI) Requirements and Design Considerations.

[15] socat: Multipurpose relay, available at: http://www.dest-unreach.org/socat/. [Retrieved: 11 Jun 2017].

[16] USB-IP Project, available at: http://usbip.sourceforge.net/. [Retrieved: 11 Jun 2017].

[17] WireGuard: fast, modern, secure VPN tunnel, available at: https://wireguard.io. [Retrieved: 11 Jun 2017].

[18] ZeroTier SDN, available at: http://zerotier.com. [Retrieved: 11 Jun 2017].