Introduction to MDSplus using Docker

Abstract **1**

1. Motivations **3**

2. What is MDSplus **3**
2.1. Connection Modes 3
2.2. User Mapping 4

3. What is Docker **5**
3.1. Docker Images 5
3.2. Docker Containers 5
3.3. Docker Networks 6
3.4. Docker Volumes 6
3.5. Dockerfiles and Building Images 6
3.6. Docker Compose 7
3.7. Limitations 8

4. MDSplus Applications **8**
4.1. Client Tools 8
4.2. Graphical Client Tools 11
4.3. Application Programming Environment 11
4.4. Data Server 12
4.5. Experiment Infrastructure 13

5. Conclusions **14**

References **15**

Acknowledgements **15**

Abstract

With increased use of MDSplus[1] comes an influx of new users. With them comes a need for more and better ways to learn the suite of tools that is MDSplus. The MDSplus team continually evaluates new technologies to improve our software and user experience. To this end, we investigated Docker to determine if and how it could help new users understand MDSplus, make MDSplus easier to install, and allow us to easily test old/new versions.

To achieve this, a set of Docker images and instructions have been developed. This paper will provide an overview of MDSplus, and detail the methods to create and use the Docker images.

Additionally, we will explore the limitations of such an approach, and the recommended applications.

The project where these Docker Images were built, along with the Demo is here:
https://github.com/WhoBrokeTheBuild/DockerizedMDSplus
https://hub.docker.com/r/whobrokethebuild/mdsplus

The now official Docker Images are available here:
https://github.com/MDSplus/Docker
https://hub.docker.com/r/mdsplus/mdsplus

Authors
S. Lane-Walsh, J. Stillerman, F. Santoro, T. Fredian

Keywords
Docker; Containers; MDSplus; Data acquisition;

1. Motivations

Common feedback that we receive about MDSplus, is that the initial installation and configuration is too difficult. As a response, MDSplus developers continuously explore the methods and interfaces that could be used to run MDSplus. Following this mission, we have containerized MDSplus in Docker to provide a streamlined installation alternative. This paper will cover the decisions and implementation details involved.

2. What is MDSplus

MDSplus is a set of tools consisting of an expression evaluator, a hierarchical data store, a networked communication protocol, and a data acquisition / analysis workflow engine. MDSplus allows users to capture, analyse, and manage data in a performant and extensible manner. Proven by decades of fusion research, MDSplus continues to explore opportunities to improve. One such way is expanding the methods and tools available to run MDSplus, which lead us to use Docker to provide our containerized approach.

MDSplus usage can be roughly separated into two categories, Client and Infrastructure. Those working with MDSplus as a Client generally use the command line or graphical tools, along with the various programming interfaces. This allows them to configure data acquisition and automated analysis, process the recorded data, perform analysis on it, and display the results. Those concerned with MDSplus Infrastructure generally support the Clients, providing data storage and access for analysis. MDSplus infrastructure involves several services for acquiring data, sending events, and doing analysis.

2.1. Connection Modes

One detail of MDSplus necessary to understand this paper is the various ways clients connect to MDSplus over the network. The four connection modes are Thin, Thick, Local and Distributed. Each comes with benefits and trade-offs, and can be distinguished by their tree path or the functions used.

Thin clients connect to MDSplus servers using `mdsconnect()`, and evaluate expressions using `mdsvalue()`. The result is that all computation and data access happens on the server, hence it is a thin client. This mode is primarily used with the programming interface, or with programs specifically intended for thin connections, e.g. `dwscope_remote`.

Thick, Local, and Distributed modes all rely on a tree path to locate the MDSplus trees. All computation is done on the client machine. This is an environment variable either in the form of `{tree-name}_path`, where `{tree-name}` is the name of the MDSplus tree or a `default_tree_path` environment variable that will be searched if the tree-specific environment

variable is absent. These paths operate in the same way as the Windows PATH variable, and contain a semicolon-separated list of paths to search.

Thick clients are configured by setting the tree path to the IP of the server followed by two colons. For example, `default_tree_path=my-server::`. This causes all requests for data to be proxied to the server, relying on the server to know where all of the trees are located. Once the data is retrieved, all processing of expressions happens locally.

Local and Distributed both work by specifying the location of the tree files as a file path. The primary difference is that Local searches the computer's filesystem, and Distributed searches a given server's filesystem. For example, local looks like `default_tree_path=/path/to/my/trees` and distributed looks like `default_tree_path=my-server::/path/to/my/trees`. In both cases, all processing of expressions happens locally.

One of the potential downsides of Thick, Local, and Distributed is that the processing of expressions happens on the client computer. Depending on the hardware of the local computer, the speed of the network, or the size of the dataset this could severely hinder analysis. One solution is to use Thin client, but this cannot be used with our Object-Oriented programming APIs. Another solution is to use Remote Python Objects[3], a method of running the majority of computation on a remote computer, while still accessing the data on the client computer.

2.2. User Mapping

Another necessary detail of MDSplus is the user mapping and access scheme. This feature, configured by the `/etc/mdsip.hosts` file, allows control over remote access to MDSplus data.

Here is an example `mdsip.hosts` file.

```
*@192.168.1.*    | MAP_TO_LOCAL
*@*.example.com | MAP_TO_LOCAL
10.1.2.3         | bob
*                | nobody
```

When an MDSplus server accepts a connection, it compares the source address against the left column in its `mdsip.hosts` file. If a match is found, it then uses the user specified in the right column. The special value of `MAP_TO_LOCAL` can be used to map the remote user to a local user of the same name. This mapped user is then used to access the tree files, allowing normal file permissions to limit access to tree files.

3. What is Docker

Docker is a tool used to run software in isolated contexts, leveraging many of the advancements made in the last decade of computer science. While not implemented as such, Docker can be thought of as a lightweight virtual machine. Instead of virtualizing hardware for the guest operating system to use, Docker merely isolates processes while running on the same Linux kernel. Docker consists of several top-level constructs, including; Images, Containers, Networks, Volumes.

3.1. Docker Images

Docker Images are analogous to executables, in that they store everything needed for the application to run. Docker Images can be managed with the `docker image` command, and built using the `docker build` command. Recipes called Dockerfiles[4] store the instructions needed to build an Image. Each instruction in the Dockerfile generates a new layer, which is a compressed snapshot of the changes made to the filesystem as a result of the instruction. This often takes the form of a package installation, or a file being copied into the image. In order to reduce the space these Images take up, layers can be shared between different Images. For example, if two images are both based on the Ubuntu 18.04 Image, then only one copy of the base image is needed. In addition, the metadata added to the image stores which command to execute when the container is run, and labels containing relevant information about the image.

Once Images are built, they are referenced using Tags. Tags are shown after the Image owner and repository, separated by a colon, e.g. `mdsplus/mdsplus:alpha`. These allow the mapping of human readable names to specific Image IDs. These names often contain the function of the image, as well as the version of the software in the Image. If no Tag is specified, the `latest` Tag is used. Using Images with Tags that contain version information can be a powerful tool for testing new versions of software. Simply run a new container with the new version tag, confirm the application functions as expected, and if it doesn't, roll back. This removes the need to search for old installers, or experience downtime during version upgrades.

3.2. Docker Containers

Docker Containers are instances of Images, containing all of their processes, logs, and changes to the original filesystem. Docker Containers can be created and managed with the `docker run` command and an Image Tag or ID. When a Docker Container is run, it will run the predefined entrypoint from the image, unless an override is given. Containers can optionally specify exposed network ports, which can then be mapped to real network ports on the host system and used for accepting traffic from the host system or its network.

In order to separate processes inside a Docker Container from those on the host system, they are placed into Control Groups (CGroups). CGroups are a Linux-native way to isolate processes from each other. Logs for a given Container are stored in the Docker Daemon, the server coordinating all docker resources, and can be retrieved using the `docker logs` command. Since Docker Images are shared resources, a running Container cannot change the original Image without altering any other running Containers using that Image. To prevent this, any changes to the filesystem are stored in one final layer, containing the differences from the Image.

3.3. Docker Networks

Docker Networks are virtual implementations of computer networks. Docker Networks can be created and managed with the `docker network` command. These provide the ability to network Containers together with each other and the host machine. Containers receive IP addresses from the Docker Daemon, on the subnet associated with their Network. Multiple Networks can be associated with a Container, allowing for complex setups mirroring real computer networks. For example, a Web Server Container could be configured to accept incoming traffic, and talk to a Database Container, while the Database Container could be configured to only talk to the Web Server Container.

3.4. Docker Volumes

Docker Volumes are virtual storage drives, generally implemented as shared folders on the host computer. Docker Volumes can be created and managed with the `docker volume` command. These can be mounted inside one or more Containers, allowing persistent storage independent of the state or lifespan of the Containers they are attached to. Docker Volumes can also take the form of Host Filesystem Mounts, which allow direct access from a Container to a folder on the host computer.

3.5. Dockerfiles and Building Images

Dockerfiles provide a simple syntax for building Docker Images one layer at a time. Given this layer based approach, it's easy to build an image off of an existing one. Dockerfiles are built into Images using the `docker build` command. This optionally allows tagging the images so that they can be referenced in the future.

For example, this `Dockerfile` installs Python and runs a script.

```
FROM alpine:3.6

RUN apk add --update python3
COPY script.py /script.py
```

```
ENTRYPOINT ["python3", "/script.py"]
```

Then from the directory where this `Dockerfile` is stored, run the following.

```
$ docker build . -t myimage
$ docker run --rm myimage
```

3.6. Docker Compose

In an effort to coordinate all of the above components, an ancillary tool named `docker-compose`[5] was created. This allows the defining of details and relationships between Images, Containers, Networks, and Volumes. This information is stored in a `docker-compose.yml` file, which then allows building, creating, updating, or destroying all of the resources at once. This is especially useful when defining complex applications, such as MDSplus infrastructure.

For example, this `docker-compose.yml` creates a MySQL database and accompanying WordPress website.

```
version: '3'
services:
  database:
    image: mysql:5.7
    volumes:
    - database_data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: supersecret
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wpuser
      MYSQL_PASSWORD: secret
  wordpress:
    image: wordpress:latest
    depends_on:
    - database
    ports:
    - "8000:80"
    environment:
      WORDPRESS_DB_HOST: database:3306
      WORDPRESS_DB_USER: wpuser
      WORDPRESS_DB_PASSWORD: secret
volumes:
  database_data:
```

3.7. Limitations

Docker has several limitations that apply to its use with MDSplus.

In section 2.2, it was discussed that MDSplus requires the ability to enumerate users on a system. In order to allow Docker to use that information, you would have to either install your sites authentication scheme inside of the Docker Container, or map in the user listing from the host. When using local user accounts the `/etc/passwd` file can be mapped into the Docker Container in order to provide this. However, when using a centralized user management system such as Active Directory, you would need to configure the Docker Container to connect to this system like one of your hosts. Doing so would require building a custom Docker Image.

As will be discussed in section 4.2, MDSplus has several graphical client tools such as `traverser` and `dwscope`. These tools require a connection to an X server in order to run. Linux natively has an X server, but by default Docker has no connection to it. This can be remedied with environment variables and volume mounts, for more information see section 4.2. Mac OSX and Windows do not natively have X servers, however Mac OSX does ship with XQuartz which properly emulates one.

Most Docker images are built for Linux, including the ones referenced in this paper. While native containers for Windows have come a long way, the MDSplus team does not currently support running MDSplus this way. Containers that are built for Linux require the Linux Kernel to run. Neither Mac OSX nor Windows have Linux Kernels, even if Mac OSX is considered similar to Linux. Therefore, in order for Docker to run on these systems, a Linux Virtual Machine has to be created, and all Docker processes have to run in that.

4. MDSplus Applications

Just as MDSplus itself supports a large set of use cases, so does the Docker implementation. Several potential applications are outlined below, including any pre-existing setup required. We also provide a live demo MDSplus endpoint at `demo.mdsplus.org`. Throughout the following explanations are working examples using this endpoint.

4.1. Client Tools

If the site provides an existing MDSplus infrastructure, the Dockerized client tools such as `mdstcl` and `tditest` will work in place of their installed counterparts. This provides an easy installation-free way to access the basic tools of MDSplus. These commands often look similar to the following command.

```
$ docker run --rm -it mdsplus/mdsplus mdstcl
TCL> show version
```

```
MDSplus version: 7.76.3
----------------------
  Release:  alpha_release-7-76-3
  Browse:   https://github.com/MDSplus/...
  Download: https://github.com/MDSplus/...
  Build date: Thu May 30 14:00:35 UTC 2019
```

This long command invocation would be onerous to type every time, so it is recommended to wrap it in a shell function, or create an alias like so.

```
$ alias mdstcld='docker run --rm -it mdsplus/mdsplus mdstcl'

$ mdstcld
TCL>
```

The command `docker run` is used to create a running Container from an image. The `--rm` flag tells Docker to remove the container once it exits. By default they are kept, allowing viewing of the logs and exit codes. The `-i` and `-t` flags specify that the Container should run interactively, and with a TTY. The `mdsplus/mdsplus` argument specifies the user and repository of the Image, both of which are `mdsplus. Finally, the last argument is specifying the command to run inside the Container when it starts.

Using MDSplus client tools without access to data would not be very useful, however. All of the connection modes described in the Connection Modes section can be used to solve this. Here are some examples.

```
# Thick Client
$ docker run --rm -it \
    -e "default_tree_path=my-server::" \
    mdsplus/mdsplus mdstcl

# Distributed Client
$ docker run --rm -it \
    -e "default_tree_path=my-server::/path/to/trees" \
    mdsplus/mdsplus mdstcl

# Local Client
$ docker run --rm -it \
    -v "/path/to/trees:/trees" \
    mdsplus/mdsplus mdstcl
```

```
# Thin Client
$ docker run --rm -it mdsplus/mdsplus tditest
mdsconnect("my-server")
```

The `-e` flag is used to set an environment variable in the container, and can be used to set specific paths for trees or the global `default_tree_path`. The `-v` flag is used to mount volumes from the host filesystem into the container's filesystem. See the Docker Volumes section. For convenience, a `default_tree_path` has been set to `/trees/~t/` in all of the Docker Images. This means that the tree files can be mounted into `/trees/` and used with no additional configuration.

In order to test this against `demo.mdsplus.org`, do the following. It is recommended to use thick client.

```
$ docker run --rm -it \
    -e "test_path=demo.mdsplus.org::" \
    mdsplus/mdsplus python3
>>> import MDSplus
>>> t = MDSplus.Tree('test', -1, 'READONLY')
>>> t.dir()
DEMO_TIME       STRUCTURE
SCRATCH         STRUCTURE
SUB             SUBTREE
...
>>> n = t.getNode('TXT')
>>> print(n.record)
```

Of course this only discusses how to run the tools packaged with MDSplus. To run custom tools that rely on MDSplus, they must either be mounted in, or a new image must be built. See the Dockerfiles and Building Images section. It is recommended to build new images off of the `mdsplus/mdsplus:latest` image.

Unfortunately, there is a drawback to using Docker Containers in this way. As mentioned in the User Mapping section above, MDSplus will attempt to map remote users to local ones. Docker Containers operate as if they are full Linux installations, and therefore don't use the list of users from the host computer. This is limiting, there are several potential solutions depending on the infrastructure configuration.

Given the general nature of the Images, we implemented the most portable solution we could find. Therefore, the Images support passing in the `UID` and `GID` environment variables. These are then used to create a fake user and group inside the Container, that will operate on the data using the provided `UID` and `GID`. Inside the docker the user and group are named

`hostuser` and `hostgroup`, respectively. Then, the default `mdsip.hosts` file in the Image has the following.

```
*     | hostuser
```

This maps all incoming connections to the docker user `hostuser`, allowing the Containers to work for one user. For implementations using multiple users, this will need to be customized by building a new image and configuring it to use the site's user list.

4.2. Graphical Client Tools

Docker was primarily designed for command line and server software, so using it for graphical applications requires some additional configuration.

On Linux, one can use X-forwarding to allow the container to access the desktop's X server. This can be done by passing in the `DISPLAY` environment variable, and mounting the `/tmp/.X11-unix` folder inside the Container. Otherwise, the graphical client tools operate exactly like the command-line client tools. Other operating systems require different methods to expose graphical programs from Docker.

```
$ xhost +local:root
$ docker run -d --rm -it \
    -v "/tmp/.X11-unix:/tmp/.X11-unix:rw" \
    -e DISPLAY \
    mdsplus:mdsplus dwscope
```

In order to use the graphical tools against `demo.mdsplus.org` on Linux, do the following.

```
$ xhost +local:root
$ docker run -d --rm -it \
    -v "/tmp/.X11-unix:/tmp/.X11-unix:rw" \
    -e "test_path=demo.mdsplus.org::" \
    -e DISPLAY \
    mdsplus/mdsplus traverser -tree test -shot -1
```

4.3. Application Programming Environment

In order to do comprehensive analysis on data, often custom applications are needed. Providing Application Programming Interfaces (APIs) in many languages has long since been a point of pride of the MDSplus team. There are two ways to use the Docker Images to run the custom software. The first is volume mounting it in when the Container is run, which is only viable if the software has little to no requirements other than MDSplus.

For example, a python script that uses MDSplus and numpy could be volume mounted in and run like so.

```
$ docker run --rm -it \
    -v "$PWD/script.py:/script.py"\
    mdsplus:mdsplus python3 /script.py
```

For applications that require other dependencies or linking against the MDSplus libraries, building an image is the better solution. See the Dockerfiles and Building Images section. It is recommended to build new images off of the `mdsplus/mdsplus:latest` image.

4.4. Data Server

Providing access to data is the quintessential role of MDSplus infrastructure. An MDSplus data server, or Tree Server as they're often called, is tasked with responding to as many connections as possible and providing them access to tree data. Given the special nature of this service, a Docker Image named `tree-server` was made that uses `inetd` to serve all incoming MDSplus connections on port 8000. The goal of using `inetd`, or an equivalent super-server daemon, is to give each connection it's own private MDSplus process. This allows all work to happen independent of other connections to the server.

Running a Tree Server requires a slightly different command than the client tools.

```
$ docker run -d \
    --restart unless-stopped \
    -v "/path/to/trees:/trees"\
    mdsplus/mdsplus:tree-server
```

First, we're now using `-d` to daemonize our application. This means that it will continue running in the background. Next, we add `--restart unless-stopped` which will restart the container in the event of an unexpected crash. Absent from the previous examples is `--rm`, as we do not want this container to remove itself when it stops. This allows us to look at any logs or exit codes left behind when it crashes. Finally, we mount in the trees we want to serve up with `-v "/path/to/trees:/trees"`, relying on the `default_tree_path` mentioned earlier.

Having a service that needs to be configured once and kept running is a good use case for Docker Compose. See the Docker Compose section. Here is the `docker-compose.yml` that powers `demo.mdsplus.org`.

```
version: "3.3"
```

```
services:
  tree_server:
    image: "mdsplus/mdsplus:tree-server"
    restart: unless-stopped
    volumes:
      - ./trees:/trees
    ports:
      - "8000:8000"
```

This can be run with the following command.

```
docker-compose up -d
```

4.5. Experiment Infrastructure

The infrastructure required to use MDSplus to collect and serve data during an experiment is one of the most complex aspects of it. This is where Docker can provide the most benefit to understanding the MDSplus infrastructure configuration. Unlike all previous applications, it is not recommended to run this using `docker run` at all. Instead create a `docker-compose.yml` describing the experiment MDSplus infrastructure. This becomes both code and documentation of the infrastructure. Of course, this should be committed to source control.

In addition to the tree server mentioned above, this section requires the use of a generic MDSip server. The only difference between this and the tree server, is that all connections share a global context. This means all connections will share the same variables and dispatch tables.

Here is an example infrastructure.

```
version: "3.3"
services:
  tree_server:
    image: "mdsplus/mdsplus:tree-server"
    volumes:
      - ./trees:/trees
      - ./pydevices:/pydevices
      - ./scripts:/scripts
    ports:
      - "8000:8000"
  dispatch_server:
    image: "mdsplus/mdsplus:mdsip-server"
    environment:
      - "MDSIP_PORT=8101"
      - "default_tree_path=tree_server::"
```

```
    volumes:
      - ./pydevices:/pydevices
      - ./scripts:/scripts
    ports:
      - "8101:8101"
  daq_server:
    image: "mdsplus/mdsplus:mdsip-server"
    environment:
      - "MDSIP_PORT=8102"
      - "default_tree_path=tree_server::"
    volumes:
      - ./pydevices:/pydevices
      - ./scripts:/scripts
    ports:
      - "8102:8102"
  analysis_server:
    image: "mdsplus/mdsplus:mdsip-server"
    environment:
      - "MDSIP_PORT=8103"
      - "default_tree_path=tree_server::"
    volumes:
      - ./pydevices:/pydevices
      - ./scripts:/scripts
    ports:
      - "8103:8103"
```

In this example, we describe three generic MDSip services. These are only separated by convention, and are configured the same save for the port they listen on. The Dispatch service is tasked with building dispatch tables and executing them. This often takes the form of calling functions on the DAQ service. The DAQ service is for running functions that connect to, and acquire data from the data acquisition devices. Finally, the Analysis service is for running analysis programs in this remote context.

To see a working example, check out https://github.com/WhoBrokeTheBuild/DockerizedMDSplus#demo.

5. Conclusions

Using MDSplus with Docker can help us leverage some of the benefits of containerized software. Version upgrades can be tested and reverted by changing which Docker Image you are running. The MDSplus team uses Docker to determine which versions of MDSplus are affected when we discover a software defect. Docker can be used to test MDSplus without committing to a full installation. The Docker Compose files outlined in this paper can be used to explain and demonstrate the pieces of an experiment infrastructure. The example scripts

provided in this paper and the Demo form a reasonable starting point for sites that want to explore using MDSplus. However, Docker currently struggles to provide cross-platform support for graphical applications. Additionally, the `UID` and `GID` mapping issues discussed above are limiting. These are both issues that are well known in the Docker community, and may receive fixes in the future. In addition, no site (not even the MDSplus team) currently uses these Docker Images in production. Given these results, the MDSplus team will continue to develop Docker as an alternative method for using MDSplus, but do not recommend it for general use at existing sites.

References

1. Stillerman, Joshua, and Thomas W. Fredian. “The MDSplus Data Acquisition System, Current Status and Future Directions.” *Fusion Engineering and Design* 43, no. 3 (January 1, 1999): 301–8. https://doi.org/10.1016/S0920-3796(98)00398-6.
2. “Enterprise Container Platform.” Docker, n.d. https://www.docker.com/.
3. “Pyro - Python Remote Objects - 4.60 — Pyro 4.60 Documentation,” n.d. https://pythonhosted.org/Pyro4/.
4. “Best Practices for Writing Dockerfiles.” Docker Documentation, June 11, 2019. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.
5. “Overview of Docker Compose.” Docker Documentation, July 31, 2019. https://docs.docker.com/compose/.

Acknowledgements

This work was funded under DOE cooperative agreement DE-SC0012470.