KNOWLEDGE BASE ARTICLE

Accelerating DevSecOps with Connext

DDS CONTAINERIZATION AND BEST PRACTICES

AUTHORS

Dave Seltz
Regional Field Application Engineer Manager, RTI

Pablo Perez
Senior Application Engineer, RTI

INTRODUCTION

Over the past decade, containerization has captured the attention of the engineering community across all verticals due to the compelling benefits it offers to software development, security, and operations (DevSecOps) teams. Containerization enhances and accelerates DevSecOps by promoting software consistency, test automation, fault reproducibility and early detection of security vulnerabilities, ultimately leading to more robust and secure service deployments.

Some of the key benefits of containerization include:

- **Isolation and Consistency**: Containers encapsulate an application and its dependencies, ensuring consistent behavior across various environments. This helps eliminate "it works on my machine" issues and provides a consistent base for testing and validation.
- **Portability**: Containers allow you to package the application along with its dependencies, making it easier to manage different versions of the application, even when rolled out to heterogeneous deployment environments.
- **Scalability**: Containers can be quickly scaled up or down based on demand. This scalability enables efficient resource utilization and helps in managing performance concerns related to sudden spikes in traffic or loads.

Moreover, containers are an integral part of modern Continuous Integration/Continuous Delivery (CI/CD) pipelines. Containerized build environments, automated test platforms and artifact generation are some of the most common use cases. The output of a CI/CD pipeline is typically a tested and hardened container that is uploaded to a container repository. Containers are also well-suited for microservices architectures, in which large distributed systems are composed of many single-purpose, loosely-coupled services that can be scaled on demand. Containerization enables better segmentation of microservices, making it easier to isolate, design, and deploy individual components.

MOTIVATION: WHY CONTAINERIZE?

Containers are lightweight, standalone, executable software packages that include everything needed to run an application. A container framework like Docker is used for building, running, and managing containers. In a containerized deployment, separate containers share the machine's OS kernel and therefore don't require a separate OS per application. This offers the advantage of using less memory, while also reducing server and licensing costs.

Why are containers popular? Because encapsulating everything needed to run the applications means they can be easily moved from platform to platform and from development to deployment.

Also, as mentioned earlier, containers are relatively lightweight. It is not necessary to spin up a whole OS and all the services associated with it. One simply leverages the existing kernel on the host.

Another advantage is that containers offer configurable isolation: Users can selectively share specific OS resources with containers, such as IPC (shared memory) or the host's network stack, or isolate containers in their use of these resources to avoid things such as port collision. Users can separately configure resource limits for each container as well.

Lastly users can update each container application individually, including all its dependencies without affecting applications in different containers.

Containerization is a great fit for highly distributed systems that need to dynamically reconfigure, add, and remove applications on demand. Using an open standard such as the OMG Data Distribution Service (DDS™) is ideal for containers. DDS abstracts the communications infrastructure from applications, and provides a consistent environment for managing, migrating, updating, and removing distributed software components, enabling the efficient and robust delivery of the right information to the right place at the right time. Some key attributes of DDS:

- Highly modular
- Completely Distributed
- Supports multiple platforms, transports, and language bindings
- Offers data centricity
- Supports a Modular Open System Approach (MOSA) and DevSecOps development

RTI Connext®, a software connectivity framework based on the DDS standard, is used today by many Real-Time Innovations (RTI) customers in a containerized environment.

CONTAINER TECHNOLOGY AND CONTAINERIZATION TOOLS

The Open Container Initiative (OCI) is an open-source project that establishes a set of industry standards for container formats and container runtimes. The OCI was formed to ensure that container images and runtimes remain vendor-neutral, interoperable, and widely adopted. The two main components of the OCI are the "Image Specification" and the "Runtime Specification."

1. **Image Specification**: A container image is a lightweight, standalone, and executable software package that includes everything needed to run an application, including executable code, system and application libraries, and configuration settings. The OCI Image Specification defines how container images are structured, including various components such as layers, metadata, and configuration. This standardization ensures that container images are portable and can be used across different container runtimes.

2. **Runtime Specification**: A container runtime defines how containers are created, started, stopped, and managed. The OCI Runtime Specification aims to provide consistency and compatibility across different container runtimes, allowing container images to be executed without modification on various platforms that support the OCI standard.

The OCI standards are widely adopted in the industry, with key container runtimes and tools such as Docker, Podman, containerd, and RKT, among others, adhering to these standards. This adoption ensures that containerization remains an open and standardized practice, benefiting both developers and operators in building and managing containerized applications.

CONTAINER NETWORKING AND DDS

Containers are just isolated and restricted Linux processes. The network resources available to a container depend on its configuration. There are several ways you can configure your container network:

- **bridge (default)**: an isolated network that local containers can use to communicate with each other
- **host**: enables the direct use of the host's network by the container
- **macvlan**: gives containers their own MAC address, making them appear as a physical device on the host's network
- **overlay**: connect multiple Docker daemons together and use swarm services to enable communication in between containers
- **<custom plugin>**: third-party network plugins are available for Docker
- **none**: disables networking in the container

MESSAGE EXCHANGE OVER THE BRIDGE NETWORK

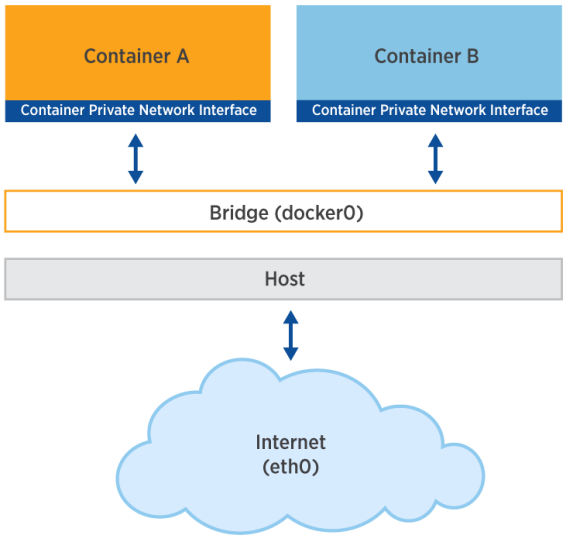

Figure 1. Detail of message exchange over the bridge network

A bridge network is the default Container network. It is a private, internal network that allows containers to communicate with each other within the same host. Containers attached to the same bridge can communicate using IP addresses. This network is often used when developers want to isolate a group of containers from the host's network. It also allows communication between the host and containers running locally.

Note: in Docker, a default bridge network is created automatically, and newly-started containers connect to it unless otherwise specified.

When using a bridge network, the container runtime installs a NAT (Network Address Translator) between the host and containers. This NAT prevents direct communication between the local containers and remote hosts and containers using the default transports in Connext.

There are two options to deal with this issue:

- Use a Connext transport that supports communication over Wide Area Networks (WAN), such as the TCP or the Real-Time WAN Transport. These transports support NAT traversal and therefore enable communication between local containers and remote hosts and containers.
- Use RTI Routing Service running on the local host (trivial configuration in Linux hosts), to bridge data between the local containers and the remote host or containers:
    - One side of the Routing Service routes talks to the bridge network.
    - The other side talks to remote hosts.
    - Remote hosts with bridge network containers similarly have their own Routing Service.
    - This approach reduces meta-traffic between hosts, but potentially increases end-to-end latency for off-host traffic.

Click here for more information.

BENEFITS AND DISADVANTAGES OF BRIDGE NETWORK COMMUNICATIONS

There are pros and cons to using DDS with a bridged network:

Pros

- Easy to configure if on-host communication is all that is needed
    - Multicast and unicast work as expected
- Provides good network isolation for containers
- Easy to map DDS ports per container
- Available on all Docker platforms

Cons

- More difficult to communicate off-host
    - RTI Routing Service can help here (especially in Linux hosts)
    - In general, a WAN transport is required to traverse the internal NAT
- Moreover, the default bridge has technical limitations
    - User-created bridges are preferable
    - See Docker documentation for additional details

MESSAGE EXCHANGE OVER THE HOST NETWORK

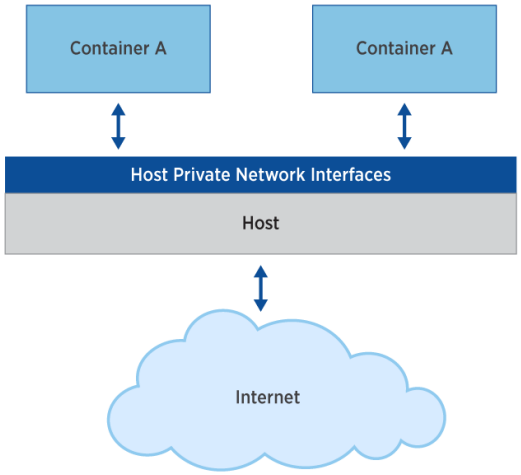

Figure 2. Detail of message exchange over the host network

In the host networking mode, a container shares the network namespace with the host machine. In other words, the container directly uses the host's network stack and is effectively connected to the same network interfaces as the host itself. This means that the container can access network services and resources just like the host. A Container does not get its own IP-address, but shares the host's networking namespace and port space. NAT is not used in the host networking mode.

SHARED MEMORY OVER THE HOST NETWORK

When using the Host network mode, DDS will interpret that both applications are in the same machine and will try to communicate using shared memory. This is an optimization performed by RTI's Shared Memory transport. To specify whether to use shared memory or not, there are three possible solutions:

1. Disabling the Shared Memory transport, as explained in the TRANSPORT_BUILTIN QosPolicy section of the User's Manual. The User's Manuals for the different Connext releases can be found on the RTI documentation page here.
2. Configuring your Connext applications and Docker containers to enable communication using shared memory, as is done for a bridged network and is explained in this Knowledge Base article.
3. Changing how the GUID is set, as explained in Controlling How the GUID is Set (rtps_auto_id_kind) in the RTI Connext DDS Core Libraries User's Manual (the User's Manuals for the different Connext releases can be found on the RTI documentation page here).

Click here for more information.

BENEFITS AND DISADVANTAGES OF THE HOST NETWORK MODE

The advantages of using a host network with Connext are:

- Easy to configure for inter-host communication
    - Multicast and unicast both work as expected
- Mostly easy configuration for on-host communication
    - SHMEM config can prevent communication between containers in some versions
- Getting a packet capture is straightforward (in Linux hosts)

The disadvantages of using a host network include:

- Shared port space limits containerized DDS apps for a domain
- Weaker isolation than a bridge network
- Works with Linux hosts only

SHARED MEMORY COMMUNICATION

By default, the IPC namespace of a Docker container is isolated from the host machine and from other containers. Fortunately, Docker provides a method to share the IPC of a Docker container with other containers or with the host. To enable message exchange using shared memory, one has to ensure that containers are configured to share memory on the host:

- Option 1:
    - Start one container with option --ipc="shareable". (This container can be either a DDS publisher or subscriber.)
    - Start other containers with option --ipc="container:<name/id>"

- Option 2:
    - For SHMEM communication with the host start containers with option --ipc="host"

Click here for more information.

CREATE A DOCKER IMAGE WITH RTI CONNEXT

The first step in creating a DDS-enabled application container is creating a "helper" image. This will be a container image that includes a DDS host/target installation. Using RTI Connext, one would gather host and target files in a directory and create a Dockerfile:

```
FROM ubuntu:20.04
WORKDIR /rti
ENV NDDSHOME /rti/rti_connext_dds-6.1.0
RUN --mount=type=bind,target=/rti/_tmp/ \
    /rti/_tmp/rti_connext_dds-6.1.0-pro-host-x64Linux.run \
    --mode unattended \
    --unattendedmodeui none \
    --prefix /rti \
    --disable_copy_examples true && \
    $NDDSHOME/bin/rtipkginstall \
    -unattended \
   /rti/_tmp/rti_connext_dds-6.1.0-pro-target-x64Linux4gcc7.3.0.rtipkg
RUN rm -rf /rti/_tmp
```

*Note this image doesn't work on platforms that don't have a host installer (e.g., Arm).

```
$> docker build -t myuser/rti-connext-dds:6.1.0 .
```

The second step in creating a Connext application container is to use the Connext helper image we just created to provide Connext build requirements (include headers and libraries).

A few points to note:

- For this step you would use a multistage build to allow multiple FROM statements – the foundation of your build comes from different bases (eg., your DDS "helper," gcc image, etc.).
- As with any container, it is important to keep the Connext app container reasonably small
    1. Standard Linux distros (Ubuntu, UBI, etc.) images are 70+ MB, so if possible, use something like Alpine Linux is ~5MB (but is missing a lot of potential dependencies).
    2. gcr.io/distroless/cc works well for our C++ test application.
- Use a Docker volume mount to store QoS (Quality of Service).

Here is a Dockerfile for an example of an application container using Connext:

```
FROM myuser/rti-connext-dds:6.1.0 AS dds
FROM gcc:8 AS builder
ARG RTI_ARCH=x64Linux4gcc7.3.0
WORKDIR /build
ENV NDDSHOME /build/nddshome
COPY src .
RUN --mount=type=bind,from=dds,source=/rti/rti_connext_dds-6.1.0,tar-
get=/build/nddshome \
           cd src && \
           make -f makefile_test_${RTI_ARCH} && \
           mkdir /build/bin && \
           mv objs/${RTI_ARCH}/test_publisher /build/bin && \
           mv objs/${RTI_ARCH}/test_subscriber /build/bin
FROM gcr.io/distroless/cc AS run-base
WORKDIR /app
VOLUME /qos
WORKDIR /qos
FROM run-base AS pub
COPY --from=builder /build/bin/test_publisher .
ENTRYPOINT ["/app/test_publisher"]
FROM run-base AS sub
COPY --from=builder /build/bin/test_subscriber .
ENTRYPOINT ["/app/test_subscriber"]
```

Build commands to use:

```
$> docker build \
      -t myuser/rti-test-app-basic-pub:6.1.0 \
      --target pub .

$> docker build \
      -t myuser/rti-test-app-basic-sub:6.1.0 \
      --target sub .
```

Commands to run the container image:

```
$> docker run \
      --mount type=bind,source=/mnt/share/qos,target=/qos \
      myuser/rti-test-app-basic-pub:6.1.0


$> docker run \
      --mount type=bind,source=/mnt/share/qos,target=/qos \
      myuser/rti-test-app-basic-sub:6.1.0
```

RUNNING RTI TOOLS AND SERVICES IN CONTAINERS

Running RTI Services applications in containers is straightforward, but please check with your account team on potential restrictions if you have a license-managed installation. Running RTI Tools in containers is less straightforward. Tools generally require a license to run, and the use of Tools is intended for licensed developers only. Please do not put an RTI license file in a shared image. Instead use a mount for the license, pointing to the license when you run the container.

Here is an example Dockerfile for building a container running the RTI Routing Service:

```
FROM myuser/rti-connext-dds:6.1.0 AS rti
FROM gcr.io/distroless/cc
WORKDIR /rti
ENV NDDSHOME /rti/rti_connext_dds-6.1.0
COPY --from=rti \
    ${NDDSHOME}/resource/app/bin/x64Linux4gcc7.3.0/rtiroutingservice \
       /rti/bin/rtiroutingservice
COPY --from=rti ${NDDSHOME}/lib/x64Linux4gcc7.3.0 /rti/lib
VOLUME /plugins
VOLUME /xml
WORKDIR /xml
ENV LD_LIBRARY_PATH /rti/lib:/plugins
ENTRYPOINT ["/rti/bin/rtiroutingservice"]
```

DEBUGGING AND TROUBLESHOOTING CONTAINERS WITH CONNEXT

So how can developers debug Connext applications that are running in containers? Well, it can be done just like any other DDS applications, but there are some issues that are specific to containers.

- If your DDS apps aren't discovering/communicating, please make sure to configure the networking correctly (as described earlier in this paper). Apps using bridge networking need special configuration to talk to remote containers and hosts – Use WAN transport or RTI Routing service.
- If you restarted your container and the rest of the network can't see it, check for a GUID collision issue:
  - Make sure to use --pid="host"
  - Make sure to use rtps_auto_id_kind == RTPS_AUTO_ID_FROM_UUID

Packet Capture of Docker Bridge Networks

To use Wireshark to capture packets on a Docker container using a bridge network, find the host interface for the bridge network:

- Run docker network ls
- Find the NETWORK ID of the Docker network to capture
- Find the network interface for this ID
  - ip a | grep <NETWORK ID>
  - ifconfig | grep <NETWORK ID>
- Run packet capture on this interface to see traffic between the containers on the network bridge

Another great tool for debugging containers is the docker inspect command. It provides an easy way of retrieving the configuration of a Docker container, including such information as:

- Defined environment variables ($NDDSHOME, $LD_LIBRARY_PATH, $NDDS_QOS_PROFILES, etc)
- Network configuration
- IPC namespace configuration

Example output from the docker inspect command for a Connext application container is shown at right:

```
$ docker inspect container1
[
    {
        "State": {
            "Status": "running",
        },
        "Name": "/container1",
        "NetworkMode": "bridge",
        "IpcMode": "shareable",
        "Image": "connext_611:1.0",
        ],
        "Config": {
            "Env": [
"PATH=/usr/local/sbin:/usr/local/bin:
/usr/sbin:/usr/bin:/sbin:/bin:/opt/rti
_connext_dds-6.1.1/bin:/opt/cmake/bin",
          "NDDSHOME=/opt/rti_connext_dds-6.1.1"
          ],
      },
      "NetworkSettings": {
          "IPAddress": "172.17.0.2",
          "Networks": {
              "bridge": {
                  (...)
              }
        (...)
```

BENEFITS AND IMPACT TO DEVSECOPS TEAMS

Containerization can reinforce the benefits of Connext by providing a flexible, scalable and consistent development and deployment environment for DDS-based applications.

Containerization complements and enhances the benefits of DDS and Connext in the following ways:

- **Isolation and Consistency**: Containers encapsulate the DDS application and its dependencies, ensuring a consistent environment across different stages of the development and deployment lifecycle.
- **Portability**: Containerized DDS applications are portable across different environments and platforms. This is particularly valuable in DDS systems that need to operate in various contexts, such as edge devices, cloud environments, and data centers. Containerization simplifies the deployment process and reduces compatibility issues.

- **Scalability**: Containers can be easily scaled horizontally or vertically to accommodate varying workloads in DDS systems. As the demand for real-time data communication grows, container orchestration platforms such as Kubernetes can automatically scale the number of containers based on traffic, ensuring optimal performance.
- **Resource Efficiency**: DDS applications typically require real-time capabilities and efficient resource utilization. Containers share the host's operating system kernel, leading to efficient use of resources, while maintaining the low-latency requirements of DDS.

By combining the benefits of Connext with the advantages of containerization, DevSecOps teams across all verticals can build more agile, scalable, and reliable real-time communication systems that can adapt to changing requirements and heterogenous deployment environments.

Additional Resources

1. Create a docker image with Connext
2. Communicate across containers
3. Using "host" driver to communicate DPs
4. Communicate DPs through SHMEM
5. RTI Xcelerator: Containers and Connext (filter on "Learn")
6. 2023 Webinar: Data Communications Issues and Strategies in a Containerized Environment

ABOUT RTI

Real-Time Innovations (RTI) is the largest software framework company for autonomous systems. RTI Connext® is the world's leading architecture for developing intelligent distributed systems. Uniquely, Connext shares data directly, connecting AI algorithms to real-time networks of devices to build autonomous systems.

RTI is the best in the world at ensuring our customers' success in deploying production systems. With over 2,000 designs, RTI software runs over 250 autonomous vehicle programs, controls the largest power plants in North America, coordinates combat management on U.S. Navy ships, drives a new generation of medical robotics, enables flying cars, and provides 24/7 intelligence for hospital and emergency medicine. RTI runs a smarter world.

RTI is the leading vendor of products compliant with the Object Management Group® (OMG®) Data Distribution Service (DDS™) standard. RTI is privately held and headquartered in Sunnyvale, California with regional offices in Colorado, Spain and Singapore.

Download a free 30-day trial of the latest, fully-functional Connext software today: www.rti.com/downloads.

Your systems. Working as one.

CORPORATE HEADQUARTERS
232 E. Java Drive, Sunnyvale, CA 94089
Telephone: +1 (408) 990-7400
Fax: +1 (408) 990-7402
info@rti.com

rti.com
company/rti
rti_software
connextpodcast
rtisoftware
rti_software