Never waste a good crisis

How COVID-19 drove innovation in maritime education

Rapid advances in new technology are changing the way seafarers learn. Advanced simulation is known to be one of the most effective applied training tools. They have been used in the training and education of seafarers for many years but often limited due to relatively high acquisition and operation costs. Democratization of simulation training is now happening, which will allow many more students to get access to high-quality simulation tools at an affordable price. Cloud technology and the increasing internet availability across the globe enable this transformation. The ongoing COVID-19 pandemic further accelerates it. We can expect improved quality of education, but this also could prove to be an essential tool in the digitalization transformation that the maritime industry faces.

Authors Gullik Jensen (Product Director for Digital Services at Kongsberg), Roy Cornelissen (Consultant @ Xpirit, working with Kongsberg since 2017) and Sander Aernouts (Consultant @ Xpirit, working with Kongsberg since 2017)

For close to five decades, Kongsberg has been a provider of simulators. Anticipating the digital shift, Kongsberg embraced the advancing technology and, in collaboration with Xpirit, pioneered the first simulator service based on its acclaimed engine room simulator platform. This service was made publicly available in March 2020, months earlier than its planned release date, motivated by the closing of maritime academies in the wake of the spread of the COVID-19 virus. By the end of the year, we had delivered a staggering thirty thousand simulations sessions to students globally.

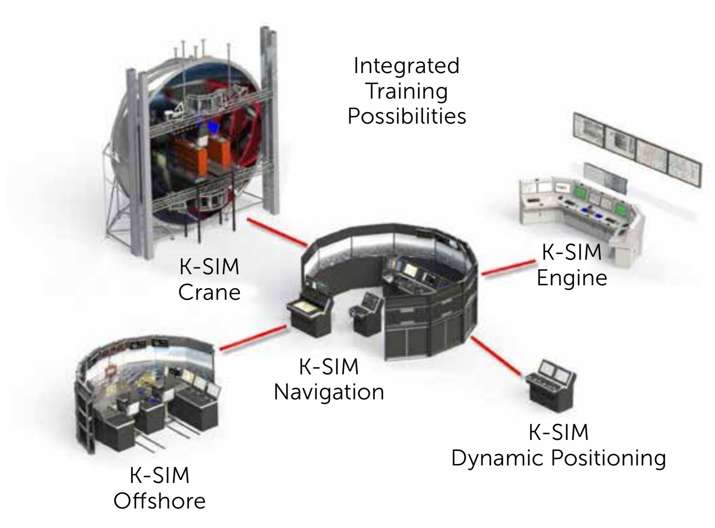
An ambitious plan

Motivated by this unconditional success, already in May 2020, Kongsberg accelerated its digitalization effort and started the cloudification of its navigation simulation platform, with the ambitious goal of offering the first public service, a RADAR simulation service, within the year. On the last day of November, we launched it. This story is about parts of the technology we created to deliver the first and probably the most advanced navigation simulator in the cloud.

We had learned a lot from bringing Kongsberg's Engine and Cargo simulator to the cloud, which we wrote about in our previous article in XPRT. Magazine #10. Could we also get their Navigation simulator to the cloud and have a working prototype in about eight weeks? Luckily, we could leverage all the work we had already done in the years before, but it wasn't a trivial task either!

First challenge: from (up to) 200 computers to 1 docker container

Kongsberg's simulator platform for navigation and offshore is called Spirit. It is a highly distributed system, with a simulator server at its core, simulating 'the world' and all hydrodynamics (motion of water and the forces acting on objects in the water). Spirit allows Kongsberg to build simulators ranging from a single desktop computer to full mission ship bridges consisting of hundreds of computers working together to drive instruments and provide real-time 3D visual imagery.



Source: https://www.kongsberg.com/digital/solutions/maritime-simulation/integrated-team-training/

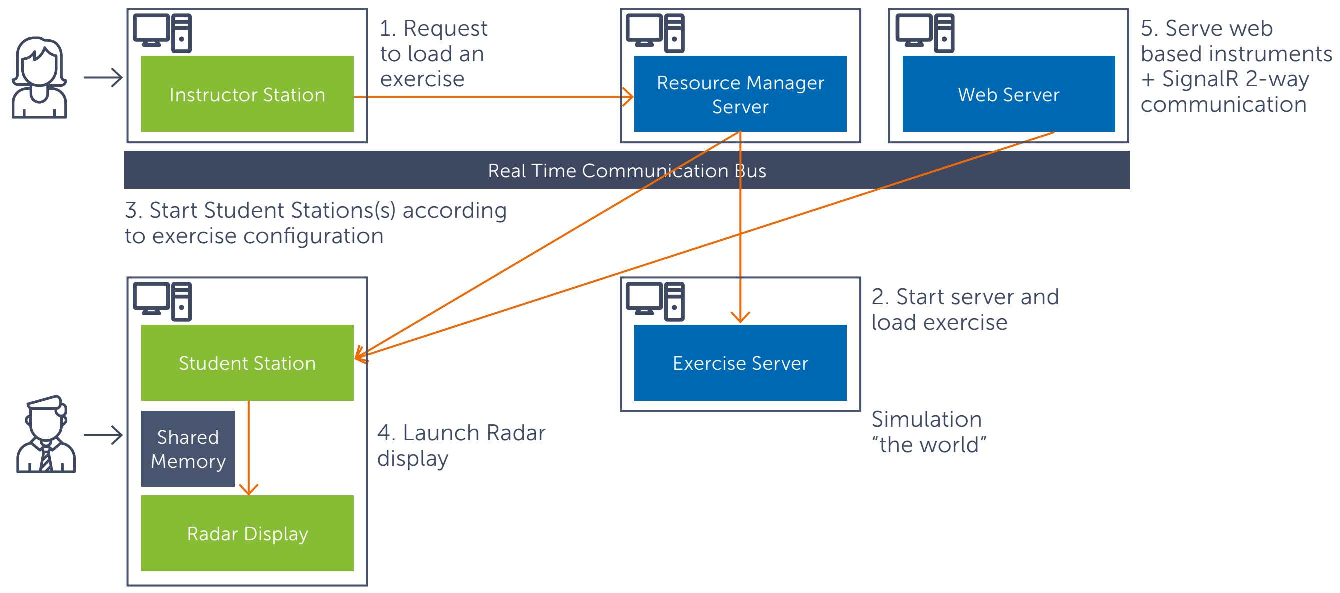
Spirit has years of investment in its platform. It is entirely Windows-based, and most of its components have some form of GUI, even the server components. However, in a cloud environment there is not much use for a GUI. A cloud-native system should run headless on a server. Sure, you can install them on a VM and serve the UI over Remote Desktop, but that is an old-fashioned solution and a costly one.

Many architects and developers would shout: "rewrite from scratch; this system is not cloud-native!". It would probably be the cheapest solution from an operational perspective in the long run, but it would have a very long time to market, tremendous development cost, not to mention disinvestment in an already successful and proven system. We had a very short time window to be successful, so while we were scaling up with our engine room simulators in production, we started working on bringing the Radar Navigation Trainer to the cloud.

Our existing platform had also proven that we could run Windows containers in the cloud just fine. Of course, Windows containers are big, and Windows nodes are more expensive to run, but it fully supports Windows-based software, including more "exotic" things like Win32 code and registry access. We knew we needed this for Spirit as well.

Our approach was somewhat trial-and-error at first because we needed to find out the obstacles we had to overcome. We took the Spirit installer and created a Docker file that installs it. Obstacle number one was that we needed to make the installer run headless. That was an easy fix in the InstallShield definition, by giving it a silent option.

Together with the Spirit architects, we looked at how we could make all the components involved in the simulation run headless.


Instructor Station
1. Request to load an exercise
Resource Manager Server
Web Server
5. Serve web based instruments + SignalR 2-way communication
Real Time Communication Bus
3. Start Student Stations(s) according to exercise configuration
Student Station
Shared Memory
Radar Display
4. Launch Radar display
Exercise Server
2. Start server and load exercise
Simulation "the world"

We already had an entire platform and ecosystem for running simulators as containers in a Kubernetes cluster named K-Sim Connect. It handles everything for scheduling simulations, managing exercises and students in a SaaS offering. Spirit also had to land in this environment. We knew we were in for a challenge to containerize a system that wasn't designed with containerization in mind.

There are roughly two approaches for this:

1. Rewrite from scratch as a headless system, using .NET Core/.NET 5 and Docker, and run it as Linux containers, or:
2. Adapt the existing system step by step and make it run in the cloud.

This diagram depicts the critical components that participate in a Radar simulation. All server components (light blue boxes) had a GUI that displays their states and provides manual controls like stopping, starting and pausing. The first thing the Spirit team did for us was changing these components to run without any GUI in a Docker container.

The green components are full-blown GUI applications (mostly WPF). They all play an essential role in the system.

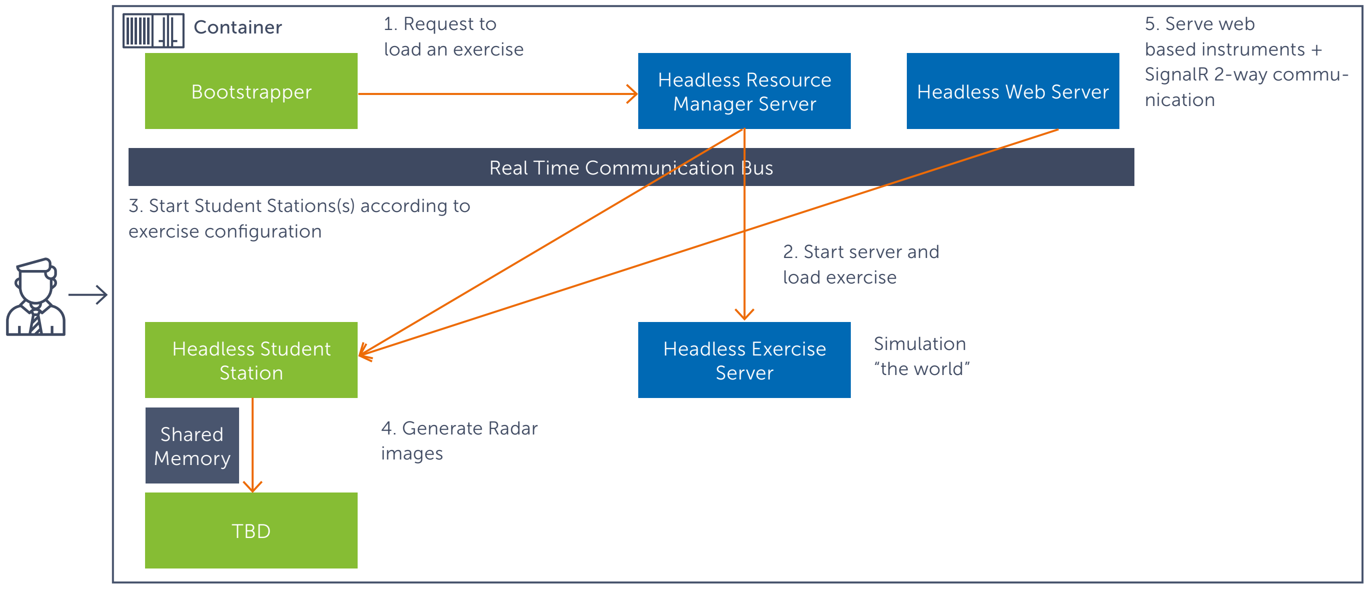
The Instructor Station is used to create exercises, add vessels, set up conditions like weather and the sailing area, assign students and start the exercise. Based on the configuration in the exercise and the simulator, the Resource Manager starts several other components.

We needed to automate the process of loading an exercise and starting the simulation without user interaction. Since we don't need all of the Instructor Station features in the cloud, only the ability to load an exercise, the Spirit team delivered a console application that did precisely that. This way, we could bootstrap the system via the command line.

The Student Station was trickier. It has the vital task of running the instruments that the students interact with. Instruments have a GUI but also hold logic to interact with the server components. The Radar instrument, in particular, has a part that generates sweeps based on the input data. A radar sweep is a full 360 degree turn of the radar beam, generating one picture. On each step in this turn, the radar generates a scan by shooting the beam in that direction.

This meant that we could reuse the existing components that generate radar sweeps. We just needed a new way to host them since they ran in the Student Station GUI application. Specifically, for the Docker container, we created a Headless Student Station. This is a .NET Console Application that loads the Spirit framework components that run the instrument logic but skips the presentation layer. One tricky part here was that the Student Station is a Windows application driven by the Windows message pump. Some components in the Student Station rely on having this message pump available. Also, the Radar's COM components require an STA thread (Single-Threaded Apartment) to run. We created a class that sets up an invisible Window that drives the message pump and sets up a Dispatcher that guarantees the Single-Threaded Apartment. It was a quick trick to make things work, but this is typically something you'd want to revisit later to make the application more container-friendly. However, it requires a more significant change in the architecture.

In the container, we launch this Headless Student Station instead of the regular one.


Container
1. Request to load an exercise
5. Serve web based instruments + SignalR 2-way commu-nication
Bootstrapper
Headless Resource Manager Server
Headless Web Server
Real Time Communication Bus
3. Start Student Stations(s) according to exercise configuration
2. Start server and load exercise
Headless Student Station
Headless Exercise Server
Simulation "the world"
Shared Memory
4. Generate Radar images
TBD

A separate executable called the Radar Display reads scans from shared memory and draws them on the screen. So, there were several things to address here:

- remove the GUI of the instruments while still running the logic;
- run Shared Memory in a docker container (could we do that?);
- replace the Radar Display application with something that could generate images without a GUI.

We had already learned that you could run quite a bit of "old" Windows mechanics in a Windows container (provided that you run a Windows Server Core image). COM, registry access, Win32, all of that works. We quickly verified that Shared Memory, which also is an old construct, worked as well.

You can run multiple processes in one container. This is what we do: all of the Spirit components run inside this single container, one container per student.

The Bootstrapper component that replaces the Instructor Station plays an important role here. It is the root process that determines the lifetime of the container. Furthermore, it communicates with the K-Sim Connect platform to track the status and progress of the simulation session. Recently, we added an automatic assessment of the student based on data from the simulator, which our web portal displays in real-time.

Second challenge: from WPF to a web-native UI

This brings us to the next elephant in the room: How to deal with the Student UI? Our first-generation Engine Room simulators still have a local GUI application. It works by virtue of a relatively simple client installation and a pure Client/

Server topology. We could bring the product to market fast, even though it's somewhat of a compromise to require a local client.

The Spirit platform is more complex, with its real-time communication bus. We knew that installing the Student Station on a client PC was not an option because of its large footprint, and we wanted to push the platform to be web-native anyway.

Over the past years, Kongsberg had invested in an extension framework for Spirit. An important driver for this was the ability to innovate on top of the platform without changing, testing, and releasing the entire platform itself every time. This multi-speed architecture of the extension framework significantly accelerated our efforts as well.

One of the tenets in the extension framework was that new instruments based on this framework would be served and rendered in a web UI. This is where the Spirit Web Server comes into play. It is an integral part of our solution. Normally these web components are hosted by the Student Station GUI application as individual panels with an embedded browser. The radar was going to be a web-based instrument as well, based on the extension framework. The Spirit Web Server would serve it, which we exposed in the Docker container, via a Kubernetes ingress. Each student gets his own (temporary) environment with a unique URL:

```
<session id>.<cluster-region>.elearning.ksimconnect.com
```

Kubernetes Ingress rules take care of the magic of routing traffic to the correct container.

The final piece of the puzzle was the replacement of the Student Station's "chrome", which handles the display and arrangement of the instrument panels. This application, named PanoramaWeb, was written specifically for our move to the web as a pure native web app, using Vue.js as its basis. Albert Brand's article in this magazine provides a more detailed background of the technology behind the web app. We will continue to extend PanoramaWeb and, as it matures, it will be the future Student Station.

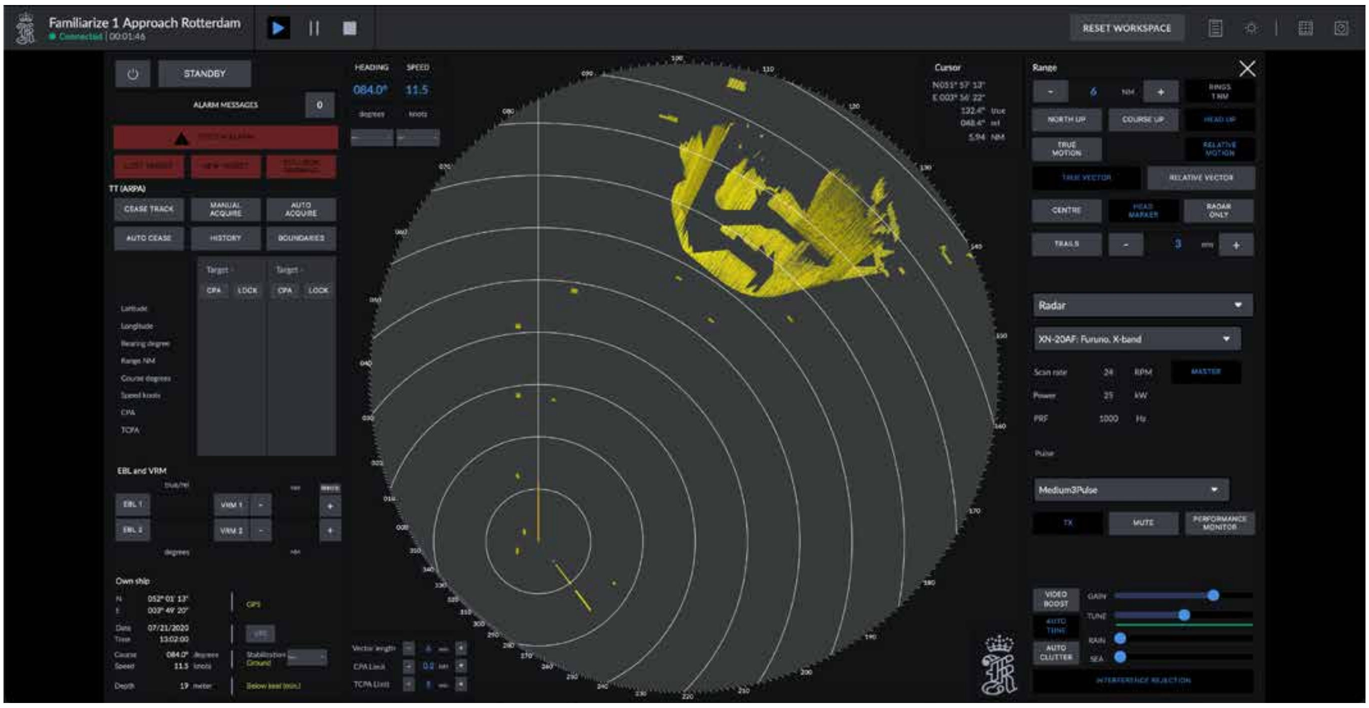
Now that we had a way to display instruments over the web, we could build the foundation of the Radar instrument. Buttons, status indicators and other user interaction like drawing range markers or bearing lines are all handled on the client side. The extension framework includes a SignalR connection with the server, which allows us to communicate state and updates between the browser and the container.

Replacing the radar display

An essential part of a radar instrument is the radar video, the well-known, often circular, view that displays the radar sweeps.

As the first diagram illustrates, the existing Radar Display is also a GUI application. It handles the drawing of the radar video, as well as all the user input. We had already dealt with the user input via the web panel. What was left was the radar video.

The data feed for the sweeps was already available in the shared memory block. The Radar Generator component constantly writes new values for each scan, much like a real radar would. We extracted the logic from the existing Radar Display GUI and created a new headless component to house that logic. It's called ScanConverter. Apart from PanoramaWeb, this is one of the few parts we rewrote for our cloud scenario. ScanConverter takes the data from Shared Memory and produces an image. We do this roughly 25 times per second, which is an acceptable frame rate.

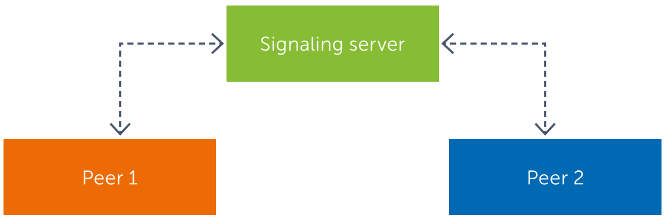


Third challenge: near-real-time communication on the web
Next, we needed a way to send these radar video frames to the browser.

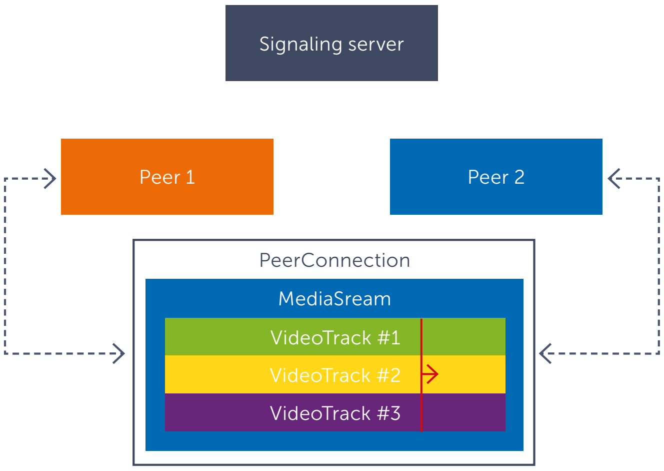
We started by looking at how streaming services such as YouTube or Netflix solved streaming video to clients at an incredible scale. But there is an important difference between streaming content such as videos and streaming a live radar video feed. When dealing with videos, users need to see them from start to finish without skipping parts of the video. Even when live streaming on YouTube, for example, the view does not have to be near real-time. For us it is more important that the user sees what is happening *right now* on the radar than that the user views the video from start to finish.

We looked at streaming technologies such as Dynamic Adaptive Streaming over HTTP (DASH or MPEG-DASH) or HTTP Live Streaming (HLS). Still, each of those prioritizes delivering a smooth (live) stream to a large number of users over providing a video stream as close to real-time as possible to a single user. We then looked at a different type of video streaming, focused on near real-time video conferencing: WebRTC. WebRTC is a protocol for real-time voice and video communication on the web. It focuses on peer-to-peer communication, and an important feature is that it is natively supported by browsers these days.

When using WebRTC, we need a so-called *signaling server*. The clients use this central server to discover each other when initially setting up the WebRTC connection. After the initial bootstrapping, the signaling server is no longer needed, and the WebRTC clients communicate directly with each other peer-to-peer. WebRTC has several mechanisms to enable such direct communication across different networks separated by the internet. When this fails, clients can use a TURN (Traversal Using Relays around NAT) relay as a fallback. With a TURN relay, clients no longer communicate peer-to-peer, but they use this central relay to communicate. The TURN relay is an essential component for us because we often need it in restricted environments such as corporate or school networks. These types of networks typically don't allow any of the mechanism that WebRTC uses to set up a peer-to-peer connection.



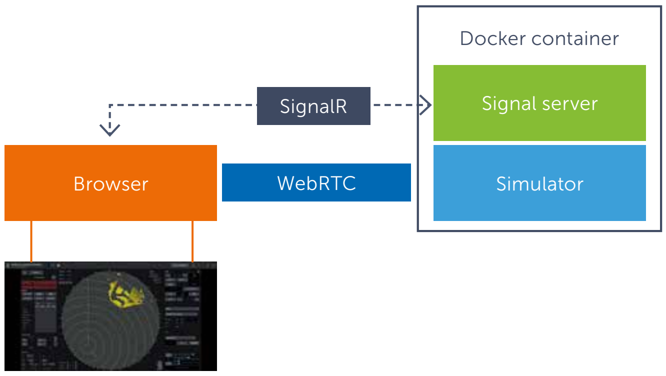
The peer connection in a WebRTC session can contain multiple video and audio streams that are synchronized. This is important in video conferencing because when you see people talk, you want to hear the sound that matches the movement of that person's mouth. In a simulation, we also want to synchronize multiple video streams such as a radar video, a 3D view and audio streams to make sure what the user sees and hears matches the current state of the simulated world.



WebRTC seemed to fit our needs perfectly, but our main challenge was to set up a WebRTC session between a Docker container and a web browser. While WebRTC is natively supported in browsers, using it on the side of the server in a C# .NET application was more complicated. We had to implement support for WebRTC into our C# application, just like the browser vendors did for their browsers. The source code for the WebRTC libraries is made public by Google, but its native C and C++ need to be integrated into your own application. After some digging, we found that Microsoft already had a project on GitHub that was aimed at supporting WebRTC in the HoloLens applications, and the library produced by this project allowed us to integrate WebRTC into our C# application with relative ease.

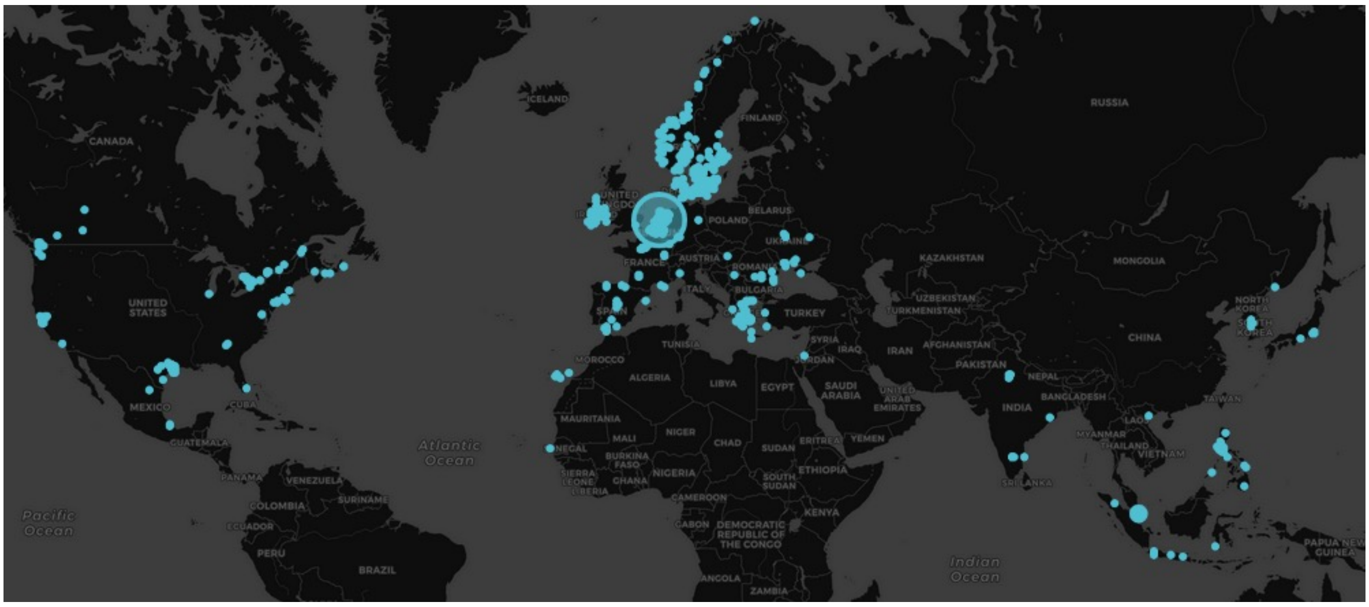
Since each simulator container is a self-contained application with its unique endpoint, we opted to put both the signaling server and the server-side peer in the same process in the Docker container: the Spirit Web Server.

The browser connects to a SignalR hub (the signaling server) that is exposed on the Docker container and exchanges the required messages to set up a WebRTC connection with the simulator, running in the same container. Once the connection is established, the simulator starts streaming the radar video, generated by the ScanConverter component, over WebRTC to the browser.



The world after COVID-19

2020 started quiet for us, and we expected to steadily grow our customer base and start working on bringing the next simulator to the cloud. COVID-19 fast-tracked our plans and ambitions. Our product owner asked us "whether we were up for a challenge," and the team boldly accepted. And now, one year later, we have clusters running in multiple regions, with users across the globe using our simulators in the cloud.



But we are far from done, we merely started to unlock the navigation simulator's potential in the cloud, and we are already looking ahead to bring more and more features besides radar to the cloud.

As with the Engine Room simulator, this project shows that you don't need a complete rewrite of your system to capitalize on it in the cloud. We were able to bring it to market fast with targeted changes, but we realize we still have work to do. But by just doing it, we have learned a lot more about where and how to focus our efforts to optimize the Spirit platform for the cloud than starting with a complete redesign. Behind the scenes, teams within Kongsberg are now working hard on making the simulator leaner and more container-friendly. New features are already being developed "cloud-first".

The launch of the RADAR service is one more important step in the democratization of maritime simulation. One hurdle at the time, we are shaping the future of maritime simulation and doing it to the benefit of the user and for the benefit of a safer and greener world. And on the way there we create some epic shit technology. </>

Roy Cornelissen
Distributed architecture, mobile development, creative

xpirit.com/roy

Sander Aernouts
Microsoft application lifecycle management (ALM)

xpirit.com/sander

Gullik Anthon Jensen
Lead digital transformation Maritime Simulation, Kongsberg Digital