Build your own container runtime

Write a C/C++ program which will a spawn a command shell in a container-like isolated environment. The following are the specifications of this container shell:

- The shell must run in separate network, mount, PID and UTS namespaces. You can achieve this in one of two ways: you can either create these namespaces beforehand and pass suitable arguments to your program to join the newly created namespaces, or you can create the namespaces directly from within your program. You can use some subset of the `clone`, `setns`, `unshare` system calls or their commandline wrappers. You can use the `ip netns` shell commands to create network namespaces. Supporting user and IPC namespaces is optional.
- You must pass a root filesystem as an argument to your program, and your shell must begin execution in the top-level root directory of this root filesystem. That is, running `ls` in the shell must show the files in the root of your new root filesystem.
- You must pass a new hostname as an argument to your program, and the shell must display this hostname upon running the `hostname` command.
- Running the `ps` command in the shell of your container must display a view of processes in the new PID namespace. You can achieve this by mounting a new proc filesystem in the container.
- You must be able to demonstrate network connectivity between your container's namespace and the default/parent namespace by running a client server application across namespaces. You can choose any client-server application of your choice. The server must be started from the shell of your container, and a client running in the parent namespace should be able to successfully connect to this server and exchange information. The two namespaces must run with different IP addresses, and connectivity between these namespaces must be suitably configured via a veth device pair. You must also ensure that the server code/executable is part of your root filesystem, so that it can be executed from your container's shell.
- You must configure limits on any one resource (e.g., CPU, memory) used by this container via Linux cgroups. You must also run a suitable program from the shell to demonstrate that the limits are being enforced. For example, if you set a limit on the maximum memory that can be used by your container, you must show that a memory-hungry application run from the shell cannot consume more memory beyond the configured limit.
- You must be able to run multiple of such container shells, and show that they are isolated from one another. For example, the two separate containers must be able to start servers listening on the same port numbers (something you cannot do with regular Linux shell). Further, you must show that the processes in the two containers are not aware of each other using the output of the ps command.
- Your program must not invoke an existing container runtime like LXC, but must accomplish the above by directly invoking the cgroups and namespaces functionality of the Linux kernel.

You must design a suitable demo for your shell, where you will run tests to demonstrate all of the above requirements, using commands of your choice. For example, you should start processes and demonstrate the ps command. You must create resource-hungry applications to demonstrate resource limits, and so on. You are responsible for writing code for all of these testcases as well.