Robot software with OpenDLV

Autonomous robots, TME290

Ola Benderius

ola.benderius@chalmers.se

Applied artificial intelligence
Vehicle engineering and autonomous systems
Mechanics and maritime sciences
Chalmers

Credits

- Partly based on teaching material from Prof. Christian Berger, University of Gothenburg

Course orientation

Learning outcomes, mapping of this lecture

- Describe properties of common types of robotic hardware, including sensors, actuators, and computational nodes
- **Apply modern software development and deployment strategies connected with autonomous robots**
- Set up and use equations of motion of wheeled autonomous robots
- Apply basic sensor fusion
- Set up and use computer simulations of autonomous robots
- Apply global and local navigation of autonomous robots
- Apply the basics of behaviour-based robotics and evolutionary robotics
- Apply methods for decision making in autonomous robots
- Discuss the potential role of autonomous robots in society, including social, ethical, and legal aspects
- **Discuss technical challenges with autonomous robots in society**

Introduction

Robot software

- Distributed
  - Computational units
  - Logic
- Modular
- Standardized interfaces

Distributed software using Ethernet: UDP and TCP

- TCP, send and receive data through a formalised link
  - Server, opens a TCP port
  - Client, connects to an IP and a TCP port, then sends packets

Distributed software using Ethernet: UDP and TCP

- TCP, send and receive data through a formalised link
  - Server, opens a TCP port
  - Client, connects to an IP and a TCP port, then sends packets
- UDP, fire and forget
  - Server, *listens* to a UDP port
  - Client, sends datagrams to an IP and a UDP port (no connection)

Distributed software using Ethernet: UDP and TCP

- TCP, send and receive data through a formalised link
  - Server, opens a TCP port
  - Client, connects to an IP and a TCP port, then sends packets
- UDP, fire and forget
  - Server, *listens* to a UDP port
  - Client, sends datagrams to an IP and a UDP port (no connection)
- UDP multicast, fire and forget to many

Distributed software using Ethernet: UDP and TCP

- TCP, send and receive data through a formalised link
  - Server, opens a TCP port
  - Client, connects to an IP and a TCP port, then sends packets
- UDP, fire and forget
  - Server, *listens* to a UDP port
  - Client, sends datagrams to an IP and a UDP port (no connection)
- UDP multicast, fire and forget to many

Low-volume data in OpenDLV is typically transmitted using UDP multicast.

libcluon

libcluon, a framework for transmitting data

- A software that acts as a middleware (glue between software components)
- Supports many communication protocols

libcluon, a framework for transmitting data

- A software that acts as a middleware (glue between software components)
- Supports many communication protocols
- In OpenDLV, mainly
  - UDP multicast (small volume data)
  - Shared memory (high volume data)

Microservices

Microservice

- A small piece of a larger software suite
- Well-defined and well isolated functionality
- Well-defined input and output

Microservice benefits

- Prevents technical debt
- Works well with OTA and deployment

OpenDLV

What is OpenDLV?

- A moderated collection of libcluon-based microservices
- Implements features of autonomous systems
- A standardised set of inputs and outputs
- Wraps common AI and ML frameworks into microservies
- Wraps common video encoding and decoding into microservices
- Unified user interfaces using web technologies

What is OpenDLV?

- A moderated collection of libcluon-based microservices
- Implements features of autonomous systems
- A standardised set of inputs and outputs
- Wraps common AI and ML frameworks into microservies
- Wraps common video encoding and decoding into microservices
- Unified user interfaces using web technologies

Growing community

We are currently setting up a new web page for OpenDLV at `opendlv.org` and `git.opendlv.org` (before we used GitHub and an internal Chalmers GitLab environment).

The OpenDLV standard message set

- The standard libcluon messages used between OpenDLV microservices

Introducing libcluon, to make a distributed system

We will now start to send and receive data.

Download header-only `libcluon` library for communication and messaging:

```
wget https://chrberger.github.io/libcluon/headeronly/cluon-complete.hpp
```

We will now start to send and receive data.

Download header-only `libcluon` library for communication and messaging:

```
wget https://chrberger.github.io/libcluon/headeronly/cluon-complete.hpp
```

```
#include <iostream>
#include "cluon-complete.hpp"
#include "prime-checker.hpp"

int32_t main(int32_t, char **) {
  PrimeChecker pc;
  std::cout << "Hello world = " << pc.isPrime(43) << std::endl;
  cluon::UDPSender sender{"127.0.0.1", 1234};
  sender.send("Hello UDP world!");
  return 0;
}
```

We will now start to send and receive data.

Download header-only `libcluon` library for communication and messaging:

```
wget https://chrberger.github.io/libcluon/headeronly/cluon-complete.hpp
```

```
#include <iostream>
#include "cluon-complete.hpp"
#include "prime-checker.hpp"

int32_t main(int32_t, char **) {
  PrimeChecker pc;
  std::cout << "Hello world = " << pc.isPrime(43) << std::endl;
  cluon::UDPSender sender{"127.0.0.1", 1234};
  sender.send("Hello UDP world!");
  return 0;
}
```

Test communication (two terminals are needed):

```
    nc -l -u 1234
    ./helloworld
```

We will now start to send and receive data.

Download header-only `libcluon` library for communication and messaging:

```
wget https://chrberger.github.io/libcluon/headeronly/cluon-complete.hpp
```

```
#include <iostream>
#include "cluon-complete.hpp"
#include "prime-checker.hpp"

int32_t main(int32_t, char **) {
  PrimeChecker pc;
  std::cout << "Hello world = " << pc.isPrime(43) << std::endl;
  cluon::UDPSender sender{"127.0.0.1", 1234};
  sender.send("Hello UDP world!");
  return 0;
}
```

Test communication (two terminals are needed):

```
    nc -l -u 1234
    ./helloworld
```

Close the `nc` server by pressing `Ctrl+c`. This is the standard way to close terminal programs.

Use a `lambda` function as a callback to receive data:

```
1  #include <chrono>
2  #include <iostream>
3
4  #include "cluon-complete.hpp"
5  #include "prime-checker.hpp"
6
7  int32_t main(int32_t, char **) {
8    PrimeChecker pc;
9    std::cout << "Hello world = " << pc.isPrime(43) << std::endl;
10   cluon::UDPSender sender{"127.0.0.1", 1234};
11   sender.send("Hello UDP world!");
12
13   cluon::UDPReceiver receiver("0.0.0.0", 1235,
14     [](std::string &&data, std::string &&/*from*/,
15       std::chrono::system_clock::time_point &&/*timepoint*/) noexcept {
16         std::cout << "Received " << data.size() << " bytes." << std::endl;
17       });
18
19   while (receiver.isRunning()) {
20     std::this_thread::sleep_for(std::chrono::duration<double>(1.0));
21   }
22
23   return 0;
24 }
```

Use a `lambda` function as a callback to receive data:

```
#include <chrono>
#include <iostream>

#include "cluon-complete.hpp"
#include "prime-checker.hpp"

int32_t main(int32_t, char **) {
  PrimeChecker pc;
  std::cout << "Hello world = " << pc.isPrime(43) << std::endl;
  cluon::UDPSender sender{"127.0.0.1", 1234};
  sender.send("Hello UDP world!");

  cluon::UDPReceiver receiver("0.0.0.0", 1235,
    [](std::string &&data, std::string &&/*from*/,
      std::chrono::system_clock::time_point &&/*timepoint*/) noexcept {
        std::cout << "Received " << data.size() << " bytes." << std::endl;
      });

  while (receiver.isRunning()) {
    std::this_thread::sleep_for(std::chrono::duration<double>(1.0));
  }

  return 0;
}
```

Try to compile.

Now we got a linking error! We need to add support for threads via an external (standard) library in `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.2)
project(helloworld)
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wextra")

find_package(Threads REQUIRED)

add_executable(${PROJECT_NAME} ${CMAKE_CURRENT_SOURCE_DIR}/helloworld.cpp
    ${CMAKE_CURRENT_SOURCE_DIR}/prime-checker.cpp)
target_link_libraries(${PROJECT_NAME} Threads::Threads)

enable_testing()
add_executable(${PROJECT_NAME}-runner test-prime-checker.cpp
    ${CMAKE_CURRENT_SOURCE_DIR}/prime-checker.cpp)
add_test(NAME ${PROJECT_NAME}-runner COMMAND ${PROJECT_NAME}-runner)
```

Now we got a linking error! We need to add support for threads via an external (standard) library in `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.2)
project(helloworld)
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wextra")

find_package(Threads REQUIRED)

add_executable(${PROJECT_NAME} ${CMAKE_CURRENT_SOURCE_DIR}/helloworld.cpp
    ${CMAKE_CURRENT_SOURCE_DIR}/prime-checker.cpp)
target_link_libraries(${PROJECT_NAME} Threads::Threads)

enable_testing()
add_executable(${PROJECT_NAME}-runner test-prime-checker.cpp
    ${CMAKE_CURRENT_SOURCE_DIR}/prime-checker.cpp)
add_test(NAME ${PROJECT_NAME}-runner COMMAND ${PROJECT_NAME}-runner)
```

Test communication (two terminals):

```
    ./helloworld
    echo "Hi there!" | nc -u 127.0.0.1 1235
```

The part looks insane! Let us look a bit closer:

```
1  cluon::UDPReceiver receiver("0.0.0.0", 1235,
2    [](std::string &&data, std::string &&/*from*/,
3      std::chrono::system_clock::time_point &&/*timepoint*/) noexcept {
4        std::cout << "Received " << data.size() << " bytes." << std::endl;
5      });
```

The part looks insane! Let us look a bit closer:

```
1  cluon::UDPReceiver receiver("0.0.0.0", 1235,
2    [](std::string &&data, std::string &&/*from*/,
3      std::chrono::system_clock::time_point &&/*timepoint*/) noexcept {
4        std::cout << "Received " << data.size() << " bytes." << std::endl;
5      });
```

The most difficult argument is the lambda function, let's extract it:

The part looks insane! Let us look a bit closer:

```
1  cluon::UDPReceiver receiver("0.0.0.0", 1235,
2    [](std::string &&data, std::string &&/*from*/,
3      std::chrono::system_clock::time_point &&/*timepoint*/) noexcept {
4        std::cout << "Received " << data.size() << " bytes." << std::endl;
5      });
```

The most difficult argument is the lambda function, let's extract it:

```
1  auto udpReceiverLambda = [](std::string &&data, std::string &&,
2      std::chrono::system_clock::time_point &&) noexcept
3      {
4        std::cout << "Received " << data.size() << " bytes." << std::endl;
5      };
```

The part looks insane! Let us look a bit closer:

```
1  cluon::UDPReceiver receiver("0.0.0.0", 1235,
2    [](std::string &&data, std::string &&/*from*/,
3      std::chrono::system_clock::time_point &&/*timepoint*/) noexcept {
4        std::cout << "Received " << data.size() << " bytes." << std::endl;
5      });
```

The most difficult argument is the lambda function, let's extract it:

```
1  auto udpReceiverLambda = [](std::string &&data, std::string &&,
2      std::chrono::system_clock::time_point &&) noexcept
3      {
4        std::cout << "Received " << data.size() << " bytes." << std::endl;
5      };
```

The lambda is a variable that is at the same time a function. Can at any time be executed as:

```
1  udpReceiverLambda("Hello!", "", std::chrono::system_clock::now());
```

Turning to UDP multicast

We will now start to send data via *multicast* (1-to-many).

Install missing Ubuntu package, only for testing:

```
1 sudo apt-get install socat
```

```
#include <chrono>
#include <iostream>

#include "cluon-complete.hpp"
#include "prime-checker.hpp"

int32_t main(int32_t, char **) {
  PrimeChecker pc;
  std::cout << "Hello world = " << pc.isPrime(43) << std::endl;

  cluon::UDPSender sender{"225.0.0.111", 1236};
  sender.send("Hello UDP world!");

  std::this_thread::sleep_for(std::chrono::duration<double>(5.0));

  cluon::UDPReceiver receiver("225.0.0.111", 1236,
    [](std::string &&data, std::string &&/*from*/,
      std::chrono::system_clock::time_point &&/*timepoint*/) noexcept {
        std::cout << "Received " << data.size() << " bytes." << std::endl;
      });

  while (receiver.isRunning()) {
    std::this_thread::sleep_for(std::chrono::duration<double>(1.0));
  }

  return 0;
}
```

Test communication (received from our program):

```
socat UDP4-RECVFROM:1236,ip-add-membership=225.0.0.111:0.0.0.0,fork -
./helloworld
```

Test communication (received from our program):

```
socat UDP4-RECVFROM:1236,ip-add-membership=225.0.0.111:0.0.0.0,fork -
./helloworld
```

Test communication (sending to our program):

```
./helloworld

  Wait 5s

echo "Hi there" | nc -u 225.0.0.111 1236
```

Introducing libcluon messages

Now it's time to add some structure to the communication.

Now it's time to add some structure to the communication.

Install libcluon to Ubuntu to get some needed tools:

```
sudo add-apt-repository -y ppa:chrberger/libcluon
sudo apt-get update
sudo apt-get install libcluon
```

Now it's time to add some structure to the communication.

Install libcluon to Ubuntu to get some needed tools:

```
sudo add-apt-repository -y ppa:chrberger/libcluon
sudo apt-get update
sudo apt-get install libcluon
```

Add a message specification (`gedit messages.odvd`):

```
message MyTestMessage1 [id = 2001] {
  uint16 myValue [id = 1];
}
```

Now it's time to add some structure to the communication.

Install libcluon to Ubuntu to get some needed tools:

```
sudo add-apt-repository -y ppa:chrberger/libcluon
sudo apt-get update
sudo apt-get install libcluon
```

Add a message specification (`gedit messages.odvd`):

```
message MyTestMessage1 [id = 2001] {
  uint16 myValue [id = 1];
}
```

Create C++ bindings (manual example):

```
cluon-msc --cpp --out=messages.hpp messages.odvd
```

```
cmake_minimum_required(VERSION 3.2)
project(helloworld)
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wextra")

find_package(Threads REQUIRED)

add_custom_command(OUTPUT ${CMAKE_BINARY_DIR}/messages.hpp
  WORKING_DIRECTORY ${CMAKE_BINARY_DIR}
  COMMAND cluon-msc --cpp --out=${CMAKE_BINARY_DIR}/messages.hpp
      ${CMAKE_CURRENT_SOURCE_DIR}/messages.odvd
  DEPENDS ${CMAKE_CURRENT_SOURCE_DIR}/messages.odvd)
include_directories(SYSTEM ${CMAKE_BINARY_DIR})

add_executable(${PROJECT_NAME} ${CMAKE_CURRENT_SOURCE_DIR}/helloworld.cpp
    ${CMAKE_CURRENT_SOURCE_DIR}/prime-checker.cpp ${CMAKE_BINARY_DIR}/messages.hpp)
target_link_libraries(${PROJECT_NAME} Threads::Threads)

enable_testing()
add_executable(${PROJECT_NAME}-runner test-prime-checker.cpp
    ${CMAKE_CURRENT_SOURCE_DIR}/prime-checker.cpp ${CMAKE_BINARY_DIR}/messages.hpp)
add_test(NAME ${PROJECT_NAME}-runner COMMAND ${PROJECT_NAME}-runner)
```

```cpp
#include <chrono>
#include <iostream>

#include "cluon-complete.hpp"
#include "prime-checker.hpp"
#include "messages.hpp"

int32_t main(int32_t, char **) {
  PrimeChecker pc;
  std::cout << "Hello world = " << pc.isPrime(43) << std::endl;

  cluon::UDPSender sender{"225.0.0.111", 1238};

  uint16_t value;
  std::cout << "Enter a number to check: ";
  std::cin >> value;
  MyTestMessage1 msg;
  msg.myValue(value);
  cluon::ToProtoVisitor encoder;
  msg.accept(encoder);
  std::string data{encoder.encodedData()};
  sender.send(std::move(data));

  std::this_thread::sleep_for(std::chrono::duration<double>(5.0));

  cluon::UDPReceiver receiver("225.0.0.111", 1238,
    [](std::string &&data, std::string &&/*from*/,
      std::chrono::system_clock::time_point &&/*timepoint*/) noexcept {
        std::stringstream sstr{data};
        cluon::FromProtoVisitor decoder;
        decoder.decodeFrom(sstr);
        MyTestMessage1 receivedMsg;
        receivedMsg.accept(decoder);
        PrimeChecker pc;
        std::cout << receivedMsg.myValue() << " is "
          << (pc.isPrime(receivedMsg.myValue()) ? "" : "not") << " a prime." << std::endl;
      });

  while (receiver.isRunning()) {
    std::this_thread::sleep_for(std::chrono::duration<double>(1.0));
  }

  return 0;
}
```

Details about message field types (only for reference, **do not add**):

```
message MyTestMessage1 [id = 2001] {
  ...
}

message MyTestMessage2 [id = 2002] {
  bool myValue1 [d = 1];
  uint8 myValue2 [id = 2];
  int8 myValue3 [id = 3];
  uint16 myValue4 [id = 4];
  int16 myValue5 [id = 5];
  uint32 myValue6 [id = 6];
  int32 myValue7 [id = 7];
  uint64 myValue8 [id = 8];
  int64 myValue9 [id = 9];
  float myValue10 [id = 10];
  double myValue11 [id = 11];
  string myValue12 [id = 12];
  MyTestMessage1 myValue13 [id = 13];
}
```

Introducing the libcluon OD4 session

Introducing the OD4 session, a way to encapsulate the above and to support multiple messages.

```cpp
#include <chrono>
#include <iostream>

#include "cluon-complete.hpp"
#include "prime-checker.hpp"
#include "messages.hpp"

int32_t main(int32_t, char **) {
  PrimeChecker pc;
  std::cout << "Hello world = " << pc.isPrime(43) << std::endl;

  cluon::OD4Session od4(111,
    [](cluon::data::Envelope &&envelope) noexcept {
      if (envelope.dataType() == 2001) {
        MyTestMessage1 receivedMsg = cluon::extractMessage<MyTestMessage1>(std::move(envelope));

        PrimeChecker pc;
        std::cout << receivedMsg.myValue() << " is "
          << (pc.isPrime(receivedMsg.myValue()) ? "" : "not") << " a prime." << std::endl;
      }
    });

  uint16_t value;
  std::cout << "Enter a number to check: ";
  std::cin >> value;
  MyTestMessage1 msg;
  msg.myValue(value);

  od4.send(msg);

  return 0;
}
```

Starting our Continuous integration (CI) environment

Creating a repository

- In this course we use Chalmers GitLab environment
- NOTE: This is not the same as the public GitLab, but Chalmers' own instance
  - Due to GDPR we should use Chalmers resources
  - Easier for us to find repositories
- We will create GitLab groups for each project group
- It is highly recommended to use it, mandatory for grade 5

Some words about Git

- Git is a software to allow for joint code development
- Originally developed by Linus Torvalds for Linux development
- Decentralised, no server is needed
    - Users can push changes between themselves locally
- Can very well be used without a GUI, and many do
- GitHub and GitLab are famous GUIs
- Another famous alternative to Git is Mercurial (better in some ways)

Logging in to Chalmers GitLab

- Go to `https://git.chalmers.se`
- Use your CID to login to the web interface
- Go to the profile settings and create a password for pushing code changes

Create a new repository for the code

- Create a *New project*
- Name it **opendlv-logic-primechecker**
- Create it under your username
- Make it private
- Initialise with a README.md
- Click *Create project*
- Bring down the *Clone* link, and copy the HTTPS clone link

Now we want to start using this. Open a terminal in Ubuntu and run:

```
sudo apt-get install git
git config --global user.email "your.email@example.com"
git config --global user.name "Your␣Name"
```

NOTE: Use your own email and name here.

```
cd ~
git clone https://git.chalmers.se/ola.benderius/opendlv-logic-primechecker.git
cd opendlv-logic-primechecker
```

- The first command is to go back to the root of your home folder
- NOTE: Of course change the URL into your own username.

We can now start pushing changes:

```
gedit README.md
```

Make changes to the file and push:

```
git status
git add README.md
git status
git commit -m "Improved␣documentation"
git push
```

You can now see the changes online, or get the latest version by:

```
1 git pull
```

The next step is to copy in the code for the prime checker

```
cp mytest/CMakeLists.txt opendlv-logic-primechecker
cp mytest/helloworld.cpp opendlv-logic-primechecker
cp mytest/prime-checker.hpp opendlv-logic-primechecker
cp mytest/prime-checker.cpp opendlv-logic-primechecker
cp mytest/catch.hpp opendlv-logic-primechecker
cp mytest/test-prime-checker.cpp opendlv-logic-primechecker
cp mytest/cluon-complete.hpp opendlv-logic-primechecker
cp mytest/messages.odvd opendlv-logic-primechecker
git status
git add -A
git status
git commit -m "Initial␣version"
git push
```

Note: Use the TAB key to make the terminal auto complete your writing, double tap to get options

The `git add -A` might be problematic. Add a `.gitignore` (note the dot before the name) file to prevent some files to be accidentally pushed:

```
# files and folders to ignore
build
```

Now, build as usually:

```
mkdir build
cd build
cmake ..
make
```

Go back and see that the ignore worked:

```
cd ..
git status
git add -A
git status
git commit -m "Added␣a␣.gitignore"
git push
```

Note: The build folder was not added or pushed, due to the ignore.

Team work

If you are many developers working at the same time

- If working in the same repo, use the Git branch concept: Git branches
- Possibly not needed when working with microservices, since they are split into different repos

Introducing Docker, the OS-level virtualisation

Now we want to automate the building, testing, and deployment. In this course we use Docker for that.

Installing Docker:

```
sudo apt-get install curl
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add
sudo add-apt-repository \
  "deb␣[arch=amd64]␣https://download.docker.com/linux/ubuntu␣$(lsb_release␣-cs)␣stable"
sudo apt-get update
sudo apt-get install docker-ce
sudo usermod -aG docker $USER

 Logout from Ubuntu, and login again (to make the group change stick)

docker run hello-world
```

Manually building with Docker (make sure to stand in your `opendlv-logic-primechecker` folder):

```
docker run --rm -ti -v $PWD:/opt/sources ubuntu:16.04 /bin/bash

apt-get update
apt-get install build-essential cmake software-properties-common

add-apt-repository -y ppa:chrberger/libcluon
apt-get update
apt-get install libcluon

mkdir /tmp/build && cd /tmp/build
cmake /opt/sources
make && make test
./helloworld
```

We need to let the network through. Adjust and redo everything:

```
docker run --rm -ti --net=host -v $PWD:/opt/sources ubuntu:16.04 /bin/bash

apt-get update
apt-get install build-essential cmake software-properties-common

add-apt-repository -y ppa:chrberger/libcluon
apt-get update
apt-get install libcluon

mkdir /tmp/build && cd /tmp/build
cmake /opt/sources
make && make test
./helloworld
```

Making changes persistent and automated, create file named `Dockerfile`:

```
1  FROM alpine:3.7 as builder
2  RUN apk update && \
3      apk --no-cache add \
4          ca-certificates \
5          cmake \
6          g++ \
7          make \
8          linux-headers
9  RUN apk add libcluon --no-cache --repository \
10       https://chrberger.github.io/libcluon/alpine/v3.7 --allow-untrusted
11 ADD . /opt/sources
12 WORKDIR /opt/sources
13 RUN mkdir /tmp/build && \
14     cd /tmp/build && \
15     cmake -D CMAKE_BUILD_TYPE=Release /opt/sources && \
16     make && make test && cp helloworld /tmp
17
18 # Deploy.
19 FROM alpine:3.7
20 RUN apk update && \
21     apk --no-cache add \
22         libstdc++
23 RUN mkdir /opt
24 WORKDIR /opt
25 COPY --from=builder /tmp/helloworld .
26 CMD ["/opt/helloworld"]
```

Running and testing the build:

```
1 docker build -t myrepository/mydockerimage .
2 docker run --rm -ti --net=host myrepository/mydockerimage
```

Running and testing the build:

```
1 docker build -t myrepository/mydockerimage .
2 docker run --rm -ti --net=host myrepository/mydockerimage
```

Save the Docker images:

```
1 docker save myrepository/mydockerimage > myImage.tar
```

Running and testing the build:

```
1 docker build -t myrepository/mydockerimage .
2 docker run --rm -ti --net=host myrepository/mydockerimage
```

Save the Docker images:

```
1 docker save myrepository/mydockerimage > myImage.tar
```

Load the Docker images (after transferring to some other computer):

```
1 cat myImage.tar | docker load
```

Running and testing the build:

```
1 docker build -t myrepository/mydockerimage .
2 docker run --rm -ti --net=host myrepository/mydockerimage
```

Save the Docker images:

```
1 docker save myrepository/mydockerimage > myImage.tar
```

Load the Docker images (after transferring to some other computer):

```
1 cat myImage.tar | docker load
```

Can we make this more convenient?

Turning to an online Docker image registry

Use the GitLab Container Registry

Instead of doing this manually, we should use the Internet (approaching OTA):

- In GitLab it is called the Container Registry
  - A poor choice of name, should rather be Image Registry
- There are other service, like Docker Hub, but integration with the CI is what we want!

Start by logging in to the registry, use your username and password:

```
1 docker login registry.git.chalmers.se
```

Now rebuild your docker image with a name that will be accepted by the registry, and then push your image (binary):

```
1 docker build -t registry.git.chalmers.se/ola.benderius/opendlv-logic-primechecker .
2 docker push registry.git.chalmers.se/ola.benderius/opendlv-logic-primechecker
```

Note: Of course change into your own username.

On any computer in the world, having Docker installed, you can now run (remember to change the username):

```
docker login registry.git.chalmers.se
docker pull registry.git.chalmers.se/ola.benderius/opendlv-logic-primechecker
docker run --rm -ti --net=host ola.benderius/opendlv-logic-primechecker
```

On any computer in the world, having Docker installed, you can now run (remember to change the username):

```
1 docker login registry.git.chalmers.se
2 docker pull registry.git.chalmers.se/ola.benderius/opendlv-logic-primechecker
3 docker run --rm -ti --net=host ola.benderius/opendlv-logic-primechecker
```

This computer could also be running inside a robot out in the wild, we now have a basic OTA!

The final steps to automate the whole CI chain

Automating the building of binaries using CI

We still have some manual steps when creating the images

- Since everything is containerized, we can as well let the server (using Docker) run the build step
- GitLab integrates a CI environment with builtin support for Docker, called GitLab CI/CD
- There are other services for this like Jenkins or Travis, but GitLab also integrates with the source code
- We want to have everything build and pushed to the GitLab Container registry when pushing a new change to the source code
- This can be controlled via a `.gitlab-ci.yml` (note the dot) file in the source folder
- We also need to activate this feature for each repository

Activating the automated build

For the GitLab repository:

- Go to Settings $\rightarrow$ CI/CD
- Click *Enable shared Runners*

Now it is time to add the `.gitlab-ci.yml` file to the repository:

```
image: docker:19.03.3
variables:
  DOCKER_HOST: tcp://docker:2375
  DOCKER_TLS_CERTDIR: ""
  DOCKER_CLI_EXPERIMENTAL: enabled
  PLATFORMS: "linux/amd64,linux/arm64,linux/arm/v7"
services:
  - name: docker:19.03.3-dind
    command: ["--experimental"]
stages:
  - build
  - deploy
before_script:
  - docker info
  - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
build-amd64:
  tags:
    - docker-build
  stage: build
  script:
    - docker build .
  only:
    - master
release:
  tags:
    - docker-build
  stage: deploy
  script:
    - docker run --privileged linuxkit/binfmt:v0.7
    - apk update && apk add curl
    - >
      curl -L "https://github.com/docker/buildx/releases/download/v0.3.1/buildx-v0.3.1.linux-amd64"
      --output "/tmp/docker-buildx" && chmod 755 /tmp/docker-buildx
    - /tmp/docker-buildx create --name multiplatformbuilder
    - /tmp/docker-buildx use multiplatformbuilder
    - /tmp/docker-buildx build --platform "$PLATFORMS" -t "$CI_REGISTRY_IMAGE":"$CI_COMMIT_TAG" . &&
      /tmp/docker-buildx build --platform "$PLATFORMS" -t "$CI_REGISTRY_IMAGE":"$CI_COMMIT_TAG" --push .
  only:
    - tags
    - /^v[0-9.]+$/
  when: on_success
```

Do not forget to push it:

```
git add .gitlab-ci.yml
git commit -m "Enabled␣CI"
git push
```

The exact workings of the CI file is outside the scope of this course, but essentially server starts the Dockerized build as soon as a new Git tag is pushed to the repository. For example if we tag the latest version of the code with the tag `v1.0`.

```
1 git tag v1.0 master
2 git push origin v1.0
```

The exact workings of the CI file is outside the scope of this course, but essentially server starts the Dockerized build as soon as a new Git tag is pushed to the repository. For example if we tag the latest version of the code with the tag `v1.0`.

```
git tag v1.0 master
git push origin v1.0
```

The progress of the new build job can be seen under the CI/CD → Jobs item at GitLab.

When done, run

```
docker login registry.git.chalmers.se
docker pull registry.git.chalmers.se/ola.benderius/opendlv-logic-primechecker:v1.0
docker run --rm -ti --net=host ola.benderius/opendlv-logic-primechecker:v1.0
```

Note: The login step is only needed once every time you log in to Ubuntu.

Further automation connected to deployment

There is one further simplification we can do, especially important since using the microservice concept as we want to start several binaries at once.

If we want to start many docker containers on a computer we could do:

There is one further simplification we can do, especially important since using the microservice concept as we want to start several binaries at once.

If we want to start many docker containers on a computer we could do:

```
docker run chalmersrevere/opendlv-device-kiwi-prugw-armhf:v0.0.2 \
     /usr/bin/opendlv-device-kiwi-prugw --cid=111
docker run chalmersrevere/opendlv-device-ultrasonic-srf08-armhf:v0.0.2 \
     /usr/bin/opendlv-device-ultrasonic-srf08 --cid=111 --freq=10 --id=1
docker run chalmersrevere/opendlv-device-ultrasonic-srf08-armhf:v0.0.2 \
     /usr/bin/opendlv-device-ultrasonic-srf08 --cid=111 --freq=10 --id=2
docker run chalmersrevere/opendlv-device-adc-bbblue-armhf:v0.0.3 \
     /usr/bin/opendlv-device-adc-bbblue --cid=111 --freq=10 --id=1
docker run chalmersrevere/opendlv-device-adc-bbblue-armhf:v0.0.3 \
     /usr/bin/opendlv-device-adc-bbblue --cid=111 --freq=10 --id=2
```

There is one further simplification we can do, especially important since using the microservice concept as we want to start several binaries at once.

If we want to start many docker containers on a computer we could do:

```
docker run chalmersrevere/opendlv-device-kiwi-prugw-armhf:v0.0.2 \
     /usr/bin/opendlv-device-kiwi-prugw --cid=111
docker run chalmersrevere/opendlv-device-ultrasonic-srf08-armhf:v0.0.2 \
     /usr/bin/opendlv-device-ultrasonic-srf08 --cid=111 --freq=10 --id=1
docker run chalmersrevere/opendlv-device-ultrasonic-srf08-armhf:v0.0.2 \
     /usr/bin/opendlv-device-ultrasonic-srf08 --cid=111 --freq=10 --id=2
docker run chalmersrevere/opendlv-device-adc-bbblue-armhf:v0.0.3 \
     /usr/bin/opendlv-device-adc-bbblue --cid=111 --freq=10 --id=1
docker run chalmersrevere/opendlv-device-adc-bbblue-armhf:v0.0.3 \
     /usr/bin/opendlv-device-adc-bbblue --cid=111 --freq=10 --id=2
```

Is there a better way?

Install `docker-compose`:

```
sudo curl -L \
  https://github.com/docker/compose/releases/download/1.20.1/docker-compose-`uname -s`-`uname -m` \
  -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
docker-compose --version
```

Put this in a file called `docker-compose.yml`:

```
version: '2'

services:
    device-kiwi-prugw:
        image: chalmersrevere/opendlv-device-kiwi-prugw-armhf:v0.0.2
        network_mode: "host"
        command: "/usr/bin/opendlv-device-kiwi-prugw␣--cid=111"

    device-ultrasonic-srf08-front:
        image: chalmersrevere/opendlv-device-ultrasonic-srf08-armhf:v0.0.2
        network_mode: "host"
        command: "/usr/bin/opendlv-device-ultrasonic-srf08␣--cid=111␣--freq=10␣--id=1"

    device-ultrasonic-srf08-rear:
        image: chalmersrevere/opendlv-device-ultrasonic-srf08-armhf:v0.0.2
        network_mode: "host"
        command: "/usr/bin/opendlv-device-ultrasonic-srf08␣--cid=111␣--freq=1␣--id=2"

    device-adc-bbblue-left:
        image: chalmersrevere/opendlv-device-adc-bbblue-armhf:v0.0.3
        network_mode: "host"
        command: "/usr/bin/opendlv-device-adc-bbblue␣--cid=111␣--freq=1␣--id=1"

    device-adc-bbblue-right:
        image: chalmersrevere/opendlv-device-adc-bbblue-armhf:v0.0.3
        network_mode: "host"
        command: "/usr/bin/opendlv-device-adc-bbblue␣--cid=111␣--freq=1␣--id=2"
```

Put this in a file called `docker-compose.yml`:

```
version: '2'

services:
    device-kiwi-prugw:
        image: chalmersrevere/opendlv-device-kiwi-prugw-armhf:v0.0.2
        network_mode: "host"
        command: "/usr/bin/opendlv-device-kiwi-prugw␣--cid=111"

    device-ultrasonic-srf08-front:
        image: chalmersrevere/opendlv-device-ultrasonic-srf08-armhf:v0.0.2
        network_mode: "host"
        command: "/usr/bin/opendlv-device-ultrasonic-srf08␣--cid=111␣--freq=10␣--id=1"

    device-ultrasonic-srf08-rear:
        image: chalmersrevere/opendlv-device-ultrasonic-srf08-armhf:v0.0.2
        network_mode: "host"
        command: "/usr/bin/opendlv-device-ultrasonic-srf08␣--cid=111␣--freq=1␣--id=2"

    device-adc-bbblue-left:
        image: chalmersrevere/opendlv-device-adc-bbblue-armhf:v0.0.3
        network_mode: "host"
        command: "/usr/bin/opendlv-device-adc-bbblue␣--cid=111␣--freq=1␣--id=1"

    device-adc-bbblue-right:
        image: chalmersrevere/opendlv-device-adc-bbblue-armhf:v0.0.3
        network_mode: "host"
        command: "/usr/bin/opendlv-device-adc-bbblue␣--cid=111␣--freq=1␣--id=2"
```

Start everything with a single command:

```
docker-compose up
```

How to extend our building to cross compilation

- Robots typically use computers of different kinds
- Different CPUs work with different machine code
- The specific compiler turns source code into a specific machine code

- Robots typically use computers of different kinds
- Different CPUs work with different machine code
- The specific compiler turns source code into a specific machine code

When working between a PC/Mac (amd64) and the Kiwi's Raspberry or BeagleBone computers (armhf) we need to cross-compile.

From before, in `Dockerfile`:

```
1  FROM alpine:3.7 as builder
2  RUN apk update && \
3      apk --no-cache add \
4          ca-certificates \
5          cmake \
6          g++ \
7          make \
8          linux-headers
9  RUN apk add libcluon --no-cache --repository \
10       https://chrberger.github.io/libcluon/alpine/v3.7 --allow-untrusted
11 ADD . /opt/sources
12 WORKDIR /opt/sources
13 RUN mkdir /tmp/build && \
14     cd /tmp/build && \
15     cmake -D CMAKE_BUILD_TYPE=Release /opt/sources && \
16     make && make test && cp helloworld /tmp
17
18 # Deploy.
19 FROM alpine:3.7
20 RUN mkdir /opt
21 WORKDIR /opt
22 COPY --from=builder /tmp/helloworld .
23 CMD ["/opt/helloworld"]
```

Using Docker for cross compilation

- Thankfully someone wrapped such cross-compilation tools within already prepared Docker images

Cross compilation, in `Dockerfile.armhf` (example, do not add):

```
FROM pipill/armhf-alpine:edge as builder
RUN [ "cross-build-start" ]
RUN cat /etc/apk/repositories && \
    echo http://dl-4.alpinelinux.org/alpine/v3.7/main > /etc/apk/repositories && \
    echo http://dl-4.alpinelinux.org/alpine/v3.7/community >> /etc/apk/repositories
RUN apk update && \
    apk --no-cache add \
        ca-certificates \
        cmake \
        g++ \
        make \
        linux-headers
RUN apk add libcluon --no-cache --repository \
      https://chrberger.github.io/libcluon/alpine/v3.7 --allow-untrusted
ADD . /opt/sources
WORKDIR /opt/sources
RUN mkdir /tmp/build && \
    cd /tmp/build && \
    cmake -D CMAKE_BUILD_TYPE=Release /opt/sources && \
    make && make test && cp helloworld /tmp
RUN [ "cross-build-end" ]

# Deploy.
FROM pipill/armhf-alpine:edge
RUN mkdir /opt
WORKDIR /opt
COPY --from=builder /tmp/helloworld .
CMD ["/opt/helloworld"]
```

Can be used as:

```
docker build -f Dockerfile.armhf -t myrepository/mydockerimage .
```

Still, this is a bit cumbersome and requires some specialised work for each platform, and we get separate images for each. Can it be simplified?

Yes, we actually already have it in our CI script:

```
  [...]
  PLATFORMS: "linux/amd64,linux/arm64,linux/arm/v7"
  [...]
    - >
      curl -L "https://github.com/docker/buildx/releases/download/v0.3.1/buildx-v0.3.1.linux-amd64"
      --output "/tmp/docker-buildx" && chmod 755 /tmp/docker-buildx
    - /tmp/docker-buildx create --name multiplatformbuilder
    - /tmp/docker-buildx use multiplatformbuilder
    - /tmp/docker-buildx build --platform "$PLATFORMS" -t "$CI_REGISTRY_IMAGE":"$CI_COMMIT_TAG" . &&
      /tmp/docker-buildx build --platform "$PLATFORMS" -t "$CI_REGISTRY_IMAGE":"$CI_COMMIT_TAG" --push .
  [...]
```

The `docker-buildx` is a way to formalise cross compilation using docker, and will also wrap each file system (for each platform) side-by-side in the resulting image. The correct sub-image will be selected on run-time `docker run` based on the host platform.

Yes, we actually already have it in our CI script:

```
  [...]
  PLATFORMS: "linux/amd64,linux/arm64,linux/arm/v7"
  [...]
    - >
      curl -L "https://github.com/docker/buildx/releases/download/v0.3.1/buildx-v0.3.1.linux-amd64"
      --output "/tmp/docker-buildx" && chmod 755 /tmp/docker-buildx
    - /tmp/docker-buildx create --name multiplatformbuilder
    - /tmp/docker-buildx use multiplatformbuilder
    - /tmp/docker-buildx build --platform "$PLATFORMS" -t "$CI_REGISTRY_IMAGE":"$CI_COMMIT_TAG" . &&
      /tmp/docker-buildx build --platform "$PLATFORMS" -t "$CI_REGISTRY_IMAGE":"$CI_COMMIT_TAG" --push .
  [...]
```

The `docker-buildx` is a way to formalise cross compilation using docker, and will also wrap each file system (for each platform) side-by-side in the resulting image. The correct sub-image will be selected on run-time `docker run` based on the host platform.

- `linux/amd64`: AMD and Intel 64bit CPUs (most laptops and desktops)
- `linux/arm64`: ARM 64bit CPUs (such as the ARM M1)
- `linux/arm/v7`: Some ARM 32bit CPUs (such as Raspberry Pi)

Putting it all together

- Using the `.gitlab-ci.yml` we can now output a single Docker image for a variety of different platforms. When a developer pushes a new tag to GitLab:

Putting it all together

- Using the `.gitlab-ci.yml` we can now output a single Docker image for a variety of different platforms. When a developer pushes a new tag to GitLab:
  - The building of all the sub-images is started

Putting it all together

- Using the `.gitlab-ci.yml` we can now output a single Docker image for a variety of different platforms. When a developer pushes a new tag to GitLab:
  - The building of all the sub-images is started
  - The unit tests are run for all sub-images

Putting it all together

- Using the `.gitlab-ci.yml` we can now output a single Docker image for a variety of different platforms. When a developer pushes a new tag to GitLab:
  - The building of all the sub-images is started
  - The unit tests are run for all sub-images
  - If not successful, the developers get an email

Putting it all together

- Using the `.gitlab-ci.yml` we can now output a single Docker image for a variety of different platforms. When a developer pushes a new tag to GitLab:
  - The building of all the sub-images is started
  - The unit tests are run for all sub-images
  - If not successful, the developers get an email
  - If successful, the GitLab Registry adds the new images that contains all side-by-side (per platform) images

Putting it all together

- Using the `.gitlab-ci.yml` we can now output a single Docker image for a variety of different platforms. When a developer pushes a new tag to GitLab:
  - The building of all the sub-images is started
  - The unit tests are run for all sub-images
  - If not successful, the developers get an email
  - If successful, the GitLab Registry adds the new images that contains all side-by-side (per platform) images
  - The image is ready to be automatically downloaded and used by the robot, the right sub-image will be used depending on the platform type found at run-time

Questions

Please post all questions on the Canvas discussion pages, in that way we can all benefit from the answers, and I can highlight important outcomes.