Practicum Guide Connected Systems

TINCOS01

Course Year: 2020-2021
Author: Thijs de Ruiter
Wouter Bergmann Tiest

Table of contents

Practicum 1: Basic robot & controller 3

Practicum 2: Protocol 5

Practicum 3: Central server 5

Practicum 4: Indicators & sensors 7

Practicum 5: Physical hardware 9

Practicum 6: Path planning 10

Practicum 7: Dashboard 11

A Final assignment 13

This lab manual describes step by step how to build a Connected System using We- bots. The steps work toward the final assignment, which is described in Appendix A. Be sure to complete the corresponding step each week during practicum on site or on your own at home. The final assignment should be demonstrated in the test week (week 8).

Practicum 1: Basic robot & controller

During the Connected Systems course, you will build a connected system. Most of the system will take place in the simulation environment Webots. In doing so, we will build on what you learned in the Modeling & Simuation course. Webots supports the following programming languages: C, C++, Java, Python and also Matlab. You are free to choose any of these programming languages to build the assignment. If you don't already have Webots `on` your computer, download it first from `https://cyberbotics.com/#download`. If you have not already done so, follow tutorial 1 through 7 via the following link: https://cyberbotics.com/doc/guide/tutorials.

In Webots, you can simulate very complex robots with all kinds of sensors and interaction with the environment. For Connected Systems we keep it simple: we assume a grid with squares of 0.1 $\times$ 0.1 m, over which the robots can move. The robot is always in the middle of a square, so its position can be indicated by integers: the number of squares in the *x* and *y* directions. The robot also does not need to have wheels; we do the locomotion by means of a translation in the *x* - and *the y* -direction. Therefore, no `Physics` node is needed.

In Webots you can use the `RectangleArena`. Make the squares 0.1 $\times$ 0.1 m by making the `floorTileSize` 0.2 in both directions (`one` *floor* consists of 4 squares). Also translate the arena so that the center of the upper left square has the coo¨ rdinates (0, 0).

For example, a robot can be a 0.1 $\times$ 0.1 $\times$ 0.1 m block. Create a `Robot` with a `Solid` as `children` with a `Shape` of your own choosing, and use the same `Shape` as `boundingObject`. Use `DEF/USE` for this; see Tutorial 2. To translate the robot from the controller, it is important to set `supervisor` to `TRUE`. You should now have something like Figure 1.

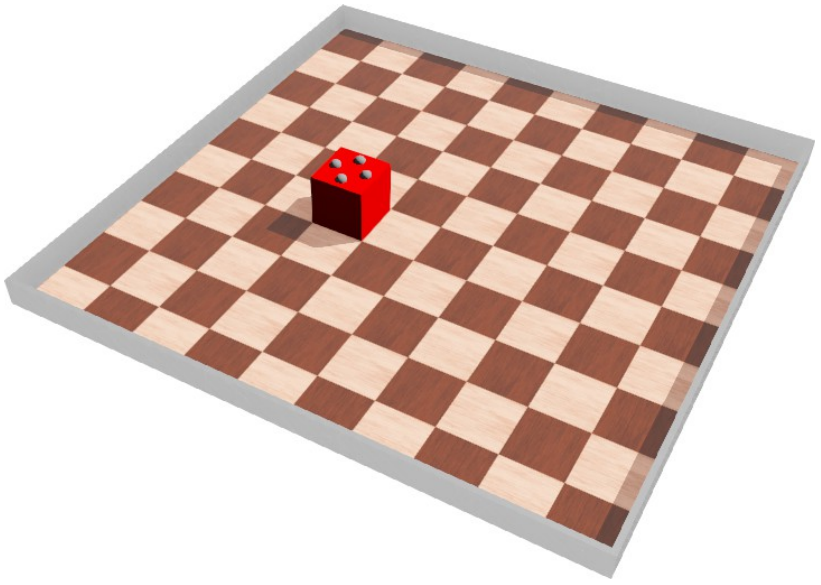


Figure 1: A simple robot on location (3, 5) in an arena

Now create a controller for the robot that lets the robot move to a specified goal by taking a step every second. To use the Supervisor properties, start with:

```python
from controller import Supervisor
```

```

# create the Robot instance
robot = Supervisor ()
supervisor Node = robot . get Self ()
```

You can make a controller that steps every second with:

```
# get the time step of the current world
timestep = int ( robot . get Basic Time Step ())

# calculate a multiple of timestep close to one second
duration = ( 1000 // timestep ) * timestep

# execute every second
while robot . step ( duration ) != -1:

    # insert controller code here
```

Now write code that makes the robot walk to a goal. For example, if the robot starts at (0, 0), it should walk to (9, 9). You can obtain the robot's current position with:

```
# get position
pos = supervisor Node . get Position ()
posX = round (10 * pos [0]) # times 10 because grid size is 0.1 x 0.1 m
posY = round (10 * pos [1])
```

Let the controller determine what the new position will be for the current step and move the robot there. You can move the robot with:

```
# get handle to translation field
trans = supervisor Node . get Field (" translation ").

# set position ; pos is a list with 3 elements : x, y and z coordinates
trans . set SFVec 3 f ( pos )
```

When the robot reaches its goal, it should remain stationary. Test your robot and turn it into a `PROTO` (see Tutorial 7). To a `PROTO` you can also add fields that you can set from the Webots environment, for example:

```
# VRML_SIM R 2020 a utf8
PROTO Unit [
  field SFString name " unit0 "
  field SFColor color 1 0 0
  field SFVec 3 f translation 0 0 0 0
  field SFVec 2 f target 5 5
]
{
  Robot {
    translation IS translation
    name IS name
    children [
      .
      .
    ]
  }
}
```

You link the values of the fields to the properties of the robot with `IS`. In this way you can set your own goal and color for each robot. In the controller, you can query the target again with:

```
# get target location
target = supervisor Node . get Field (" target ")
target Vec = target . get SFVec 2 f ()
tarX = int ( target Vec [0])
tarY = int ( target Vec [1])
```

Using the PROTO, put multiple copies of the robot in different colors in the arena, and give each one its own starting point and goal.

Practicum 2: Protocol

In order to share information with other robots, a set of agreements on how messages are built up is needed. Together, these make up the *protocol*. The idea is that you program your own robot, but together with your group agree on a common protocol that everyone in the group abides by.

For navigating the robots through the simulated space, it is helpful if the following information is shared with the other robots:

- The location of your robot.
- The locations of obstacles and walls in the world, as far as known.
- Possibly: the planned direction in which your robot intends to

move. In addition, the following aspects of messages may be of

interest:

- The sender: do I get my own message back or is it new information from another robot?
- The type of message: is it about the location of moving robots or about fixed obstacles? In the former case, I can wait a while if my path is blocked; in the latter, I have to choose another path.
- The date and time: is it a recent post or now obsolete?
- Possibly: a version number of the protocol: the version of the protocol determines how the information should be interpreted.

A protocol must be defined very precisely so that it cannot be interpreted in different ways. Consider what punctuation marks you use to separate fields, where spaces or newlines are allowed, how the end of the message is indicated, etc. This is written down in the *speci- fication*. For example:

```
message := sender : <space> content <newline>
sender := <string without spaces, colons or newlines>
content := <string without newlines>
```

Together with your group, design a protocol for data exchange between your robots. Write this down in a specification.

Exercise 3: Central server

For different robots to communicate with each other (remotely, in their own environment), they can contact a central server, sending each other messages about their position and direction. Thus, they can avoid each other and give each other information, such as the location of an obstacle.

TCP/IP sockets can be used for the server. These operate through a self-selectable port. Port numbers 1024 and up are freely available. Agree a port number with your group. In Python, start a server with:

```python
import socket

HOST = ""
PORT = 1024

s = socket . create_server (( HOST , PORT ))
s. listen ()
```

Then the server waits to connect to a client with:

```python
conn , addr = s. accept ()
```

conn is a socket object that can be used to send and receive data. addr contains the address and port number of the client. To receive data from the client use:

```
data = conn . recv ( 1024 ) . decode () # 1024 is the buffer size
```

Send data (as a string) with:

```
conn . sendall ( data . encode (" ascii ")) # encode to convert a string object to a byte
    array
```

The client can connect to the server in this way:

```
import socket

HOST = " localhost " # change to IP address of server
PORT = 1024

s = socket . socket ()
try :
    s. connect (( HOST , PORT ))
except :
    print (" Connection refused ")
    exit ()
```

Then the client can send and receive data with `s.sendall()` and `s.recv()`.

To keep in touch with multiple clients, it is convenient for the server to start a different thread for each connection. Here is an example of how that could be done:

```
import socket
import threading

HOST = ""
PORT = 1024

messages = {}
client Count = 0

def echo ( conn ):
    global messages , client Count
    while True :
        data = conn . recv ( 1024 ) . decode ()
        if data == "":
            print (" Connection lost to", conn . getpeername ())
            break
        name , msg = data . split (": ", 1)
        messages [ name ] = msg
        try :
            conn . sendall ( str ( messages ). encode (" ascii "))
        except :
            print (" Connection lost to", conn . getpeername ())
            break
    client Count -= 1
    if client Count == 0:
        print (" All clients disconnected ; resetting ")
        messages = {}

s = socket . create_server (( HOST , PORT ))
s. listen ()
while True :
    conn , addr = s. accept ()
    client Count += 1
    print (" Connection accepted from ", addr )
    t = threading . Thread ( target = echo , args = ( conn ,))
    t. start ()
```

This program takes input from clients in the form `sender: message` and puts it into a common *dictionary* `messages`. When a new message arrives with the same sender, the previous message from that sender is overwritten. Then all messages are sent back to the client as a string. This string can be converted back to a dictionary by the client using the `eval()` function.

The idea is for several robots to communicate with each other via the server, each in their own Webots simulation running on a separate computer. Initially, you probably want to test this with several robots within one Webots simulation. Multiple instances of the controller program are then active, one for each robot. It is convenient that not all controllers start at exactly the same time, so that the robots take their steps one by one. You can achieve this by skipping any number of simulation steps before the actual controller loop starts:

```
import random

# sleep a random amount of time so as not to start all controllers at the same time
for i in range ( random . randint (0 , 1000 // timestep )):
    robot . step ( timestep )
```

In the controller loop, a robot must pass its own position and direction to the server, and get back the positions of all other robots. For this, it is important that each robot has a unique name. You can set that in Webots in the robot's `name-field`. In the controller, you can retrieve that name with `robot.getName()`.

Now use the protocol you designed last lesson to send position and direction information from your robot controller to the server, and get information back from the other robots. Based on this, the controller can adjust the intended direction or decide to wait for a while to avoid collisions. Test this by running several robots simultaneously across the board toward their own goal.

If this works locally, have your group have everyone's robot walk around in its own Webots environment and make con- tact with the shared server. To make the remote robots visible, you can locally add a robot that exactly copies the movements of a remote robot. The controller of this robot does not send data itself, but only receives it. To trigger the server, you can send a dummy message: `"dummy: no content"`. Start the simulations simultaneously and see if the robots avoid each other correctly.

Practicum 4: Indicators & sensors

Robots often have to cooperate with humans, and then interfacing between the human and the robot is important. The human must tell the robot what to do, and the robot must indicate what it will do to avoid dangerous situations.

In Webots, for example, we can give a robot LEDs to indicate which direction it wants to move. You can turn an LED into a `PROTO` so that you can use it multiple times:

```
# VRML_SIM R 2020 a utf8
PROTO Simple LED [
  field SFString name " led0 "
  field MFColor color [
    1 0 0
  ]
  field SFVec 3 f translation 0 0 0 0
]
{
  LED {
    translation IS translation
    children [
      Shape {
        appearance Appearance {
          material Material {
            diffuse color 0.5 0.5 0.5
          }
        }
        geometry Sphere {
          radius 0.01
        }
      }
    ]
    name IS name
    color IS color
  }
}
```

So you can make 4 LEDs that you give each one a different name. Give them a translation so that they are on top of the robot, each in its own direction (see also Figure 1). In your controller you can then operate the LED with:

```
led = robot . get Device (" led0 ")
led . set (1) # turn LED on
led . set (0) # turn LED off
```

Make sure that for each step the robot does, the appropriate LED lights up.

To detect obstacles, your robot can use distance sensors. By default, the `DistanceSensor` built into Webots has a range of 0-0.1 m (0-1000 units). Thus, if you place one on each side of your robot, you can see exactly whether there is an obstacle, a wall or another robot in the four fields around your robot. Do not put the sensors exactly on the outside of the robot, because if it is exactly against an obstacle, there is a chance that the sensor will not see the obstacle due to rounding errors. For example, put them 0.04 m off center; see also Figure 2. If the `DistanceSensor` then gives a value less than 1000 (0.1 m), then there is an obstacle or wall next to the robot. By looking at the value, you can figure out what kind of obstacle it is (e.g., a pole or a box). If the value is exactly 1000, then the sensor sees nothing.

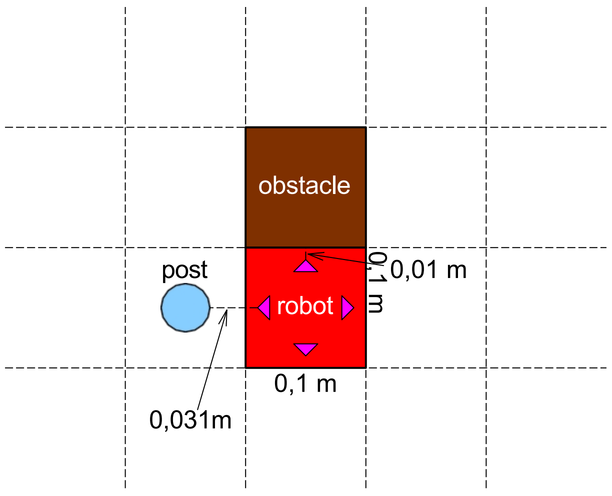


Figure 2: Distance sensors in a robot

You can use the sensor with:

```
ds = robot . get Device (" sensor Name ") # replace with name of sensor
Rev. enable ()

# in controller loop :
distance = ds. get Value ()
```

Equip your robot with 4 distance sensors and process the data in the controller. Send messages to the server when your robot detects an obstacle. It is most convenient to add the position of new obstacles to those already known (e.g. with `set.union()`), and always share the whole list with the server. Also make sure that when your robot detects other robots, they are not seen as obstacles so that there are only stationary obstacles in the list. You can do this by filtering the positions of other robots from the list.

Exercise 5: Physical hardware

Although working in a simulated environment has many advantages, it is also important to learn how to create a connected system with physical hardware. In this lab, we will implement the safety aspects, i.e. the emergency stop button and the LEDs that indicate the robot's direction, in physical hardware and connect them to the rest of the system. For this you need a microcontroller with network connection, for example an ESP32, ESP8266 or Raspberry Pi Pico W.

Connect 4 LEDs (don't forget the series resistor) and a push button (don't forget the pull-up resistor (internal or otherwise) to the microcontroller. Create a test program to verify that everything works. Then make sure the microcontroller can connect to the wireless network and make a socket connection to the server. If you are using the Arduino framework, you can use the ESP8266WiFi library, for example. To start, connect to the wireless network:

```
# include < ESP 8266 Wi Fi . h>

void setup () {
  Serial . begin ( 115200 ) ;
  Serial . print ("n Connecting ");
  WiFi . begin (" Tesla IoT ", " fs L 6 Hgj N ");
  while ( WiFi . status () != WL_CONNECTED ) {
    delay ( 500) ;
    Serial . print (".");
  }
  Serial . print ("n Connected , IP address : ");
  Serial . println ( WiFi . local IP ());
}
```

Once you have made network connection, you can connect to your central server from Practicum 3:. For this- you need the IP address of the computer the server is running on and the port number. The server must be connected to the same WiFi network (or you must set up IP forwarding).

```
Wi Fi Client Client ;

void loop () {
  client . set Timeout ( 1000 ) ; // set timeout for establishing connection
  if (! client . connect (" 192 . 168 . 1 . 1 ", 1024) ) { // fill in IP address and port number
    Serial . println (" connection failed ; wait 5 sec ... ");
    delay ( 5000 ) ;
    return ;
  }
  Serial . println (" connected to server ");
      .
      .
}
```

If you are connected to the server, you can wait for messages:

```
  client . set Timeout (1) ; // set timeout for end - of - message
  while ( client . connected ()) {
    if ( client . available ()) {
      String line = client . read String ();
      Serial . print (" message from server : ");
      Serial . println ( line );
    }
    yield ();
  }
  Serial . println (" connection lost ");
```

Within this do-loop, of course, you then have to do something with the incoming message, such as turn on the LEDs. The time-out here is set to 1 ms instead of the default 1000 ms, so that you can react almost immediately to incoming messages. Note: set the timeout to 1 ms only *after* you *have* connected to the server; otherwise it will not succeed in connecting.

To read the emergency stop button, you can use a hardware interrupt. On the ESP8266, you can use any GPIO pin for hardware interrupts except D0 (GPIO16). For example, if you use pin D1 (GPIO5), add to `setup()`:

```
  pin Mode ( D1 , INPUT_PULLUP );
  attach Interrupt ( D1 , sendStop , FALLING );
```

where `sendStop` is the *interrupt service routine* (ISR). You must declare this function with the keyword `IRAM_ATTR` to ensure it is placed in RAM and not flash memory:

```
IRAM_ATTR void send Stop () {
  if ( client . connected ()) {
    // send message . Do not use println ( introduces an empty line )
    client . print (" STOP ");
    client . flush ();
  }
}
```

Here, `print()` is used to send the message without an end-of-line at the end. If you were to use `println()`, you are actually sending two messages with the second being an empty message. Note: in the ISR, you cannot use `delay()` or functions that call `yield()`. If you do need `yield()`, use `esp yield()`.

Now program your physical unit and modify the server so that it responds to the STOP message by sending a message to all robot controllers in the simulation to stop moving. Adjust the robot controller accordingly and also make sure that it tells the physical unit in which direction the robot wants to move. Test the whole simulation, server and physical unit.

Exercise 6: Path planning

During this lesson, we will work on planning your robot's route to the specified goal as efficiently as possible. This lesson will discuss a number of algorithms.

To determine a route, the first thing that must be available is a map that can be used to calculate the route. This requires a data structure. One possible data structure for this is graphs. Graphs can be used to indicate the relationships between different objects. In our case, these objects will be the different coo¨ rdinates on the playing field.

A graph consists of vertices and edges. Vertices are connected by means of edges. These edges can have a direction and/or a weight. Algorithms can be applied to a graph to calculate its properties or to make predictions. For example, a route can be determined using algorithms applied to an edge. You can use the following algorithms for your robot:

- Dijkstra's Algorithm: `https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm`
- Topological Search: `https://en.wikipedia.org/wiki/Topological_sorting`
- Bellman-Ford Algorithm: `https://en.wikipedia.org/wiki/Bellman-Ford_algorithm`

Note that not every algorithm works in every situation; for example, some algorithms do not work when there is a circle in the map. Look at all the algorithms and find out in which cases they do not work and which one is most suitable for your robot.

As soon as your robot receives a task, you can start planning a route to the goal by making a count and applying one of the above algorithms. To make a count of the board, you use the position of all robots and information about the layout of the board, as well as information that your own robot and the other robots have collected about the position of unexpected obstacles. You may take the dimensions of the game board (e.g. 10×10) as fixed. For each field, create a *node* (vertex) and list it. A node can have one or more *links* (edges) to other nodes. In principle, a link can go two ways (a link from node *A* to node *B* is also a link from node *B* to node *A*), but in practice it is often more convenient to work with links that only go one way, and then double it.

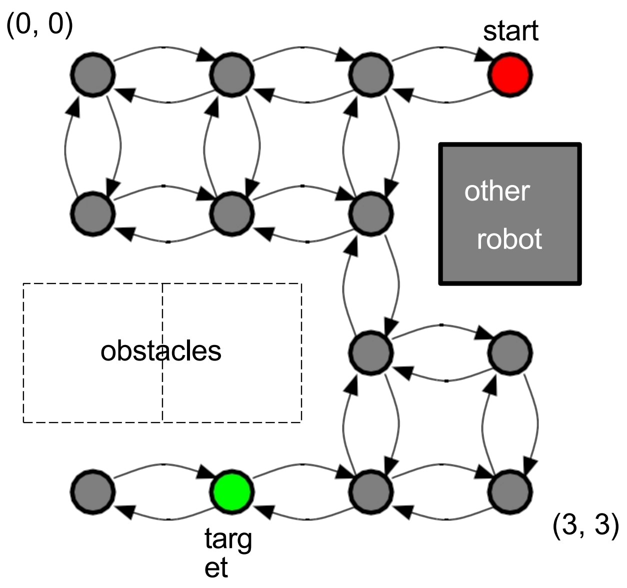
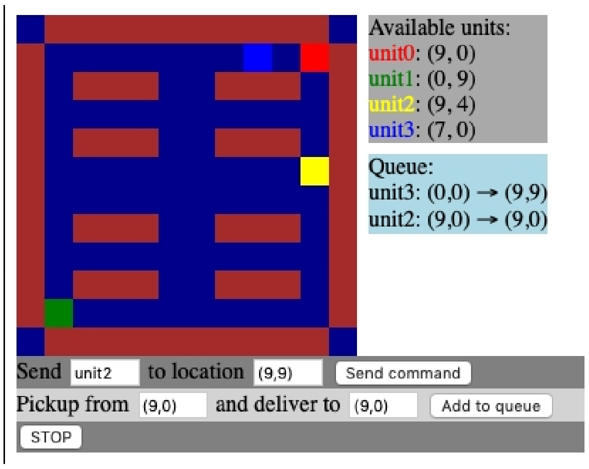


Figure 3: A count with 13 nodes and 30 links

perform. For each node make links to the neighboring nodes, unless there is an obstacle or robot there; see Figure 3. In this case each link can have the same weight (length), e.g. 1. By using as start-node the position of your own robot and as target-node the position of the target, you can determine the route and take a step in the right direction. It is recommended to do this every step, as more and more information about the obstacles becomes known, and the other robots also change position.

Exercise 7: Dashboard

To display information about the robots and give them commands, you need a dashboard. You can build this as a GUI application, or you can run it in a Web browser. That makes distribution very easy. On the dashboard you can draw a map with the positions of all the robots and obstacles. You can show what jobs *are* in the *queue* and which robot is working on them. There should be a possibility to send a robot to a location, or place commands in the queue such as "pick up an item from location (3,2) and bring it to location (7,4)". Finally, there should be a button for an emergency stop. An example of a dashboard is shown in Figure 4.


Available units:
unit0: (9, 0)
unit1: (0, 9)
unit2: (9, 4)
unit3: (7, 0)

Queue:
unit3: (0,0) → (9,9)
unit2: (9,0) → (9,0)

Send unit2 to location (9,9) Send command
Pickup from (9,0) and deliver to (9,0) Add to queue
STOP

Figure 4: Example of a dashboard

If you create the dashboard in a web browser, the obvious solution is to use websockets for the connection to the server. For this, you can extend your server with a websocket server:

```
import websockets
import asyncio
import json

HOST = ""
PORT = 8000

messages = {}

async def handle_ws ( websocket , uri ):
    print ( f" Connection accepted from { uri }")
    while True :
        try :
            data = await websocket . recv ()
        except :
            print ( f" Connection lost to { uri }")
            return
        if data == " request_messages ":
            await websocket . send ( json . dumps ( messages ))

ws = websockets . serve ( handle_ws , HOST , PORT )
asyncio . get_event_loop (). run_until_complete ( ws)
asyncio . get_event_loop (). run_forever ().
```

When a client connects to this server and sends the text `request messages`, the dictionary `messages` are converted to JSON format and sent back. In the browser, you can use JavaScript to create a websocket client:

```
<script >
let ws = new Web Socket (" ws :// localhost : 8000 ");

function get_data () {
    ws. send (" request_messages ");
}

ws. onmessage = function ( message ) {
    let data = JSON . parse ( message . data );
    // do something with the data
}

set Interval ( get_data , 1000) ;
</ script >
```

This code sends text `request messages` to the server every second. When a message is returned, the function `ws.onmessage()` is called. Here the data is converted from JSON format to a JavaScript object, which you can then do something with.

For example, for drawing a map, you can use an HTML canvas:

```
< canvas id=" the Canvas " width =" 240 " height =" 240 " style =" background - color : darkblue
" >
</ canvas >
<script >
Canvas Rendering Context 2 D . prototype . clear = function () {
    this . clear Rect (0 , 0 , this . canvas . width , this . canvas . height );
};
Canvas Rendering Context 2 D . prototype . draw Block = function ( x, y) {
    this . fill Rect (20 * x, 20 * y, 20 , 20) ;
}

let context = document . get Element By Id (" the Canvas "). get Context (" 2 d");

function draw () {
    context . clear ();
    context . fill Style = " red ";
    context . draw Block (3 , 5); // Replace with values coming from the server
}
</ script >
```

The code above adds two additional functions to the canvas context: `clear()` and `drawBlock(x, y)`. `context.clear()` clears the canvas so you can create a new drawing. `context.drawBlock()` draws a 20 × 20 pixel square at the specified coordinates (between 0 and 11 in this example). You set the color with `context.fillStyle`. You can use these functions to draw the map in the `draw()` function, which you call when a new message arrives.

You can also display text on the dashboard. The easiest way to dynamically display text on a web page is to create a `<div>` and modify its `innerHTML`-property:

```
<div id=" my Div " style =" background - color : darkgray"></div >
<script >
document . get Element By Id (" my Div "). inner HTML = " This text appears in the
    location of the & lt; div & gt ;.";
</ script >
```

To use input fields and buttons you can use the code below:

```
<input type =" text " id=" my Input " size =" 6 " >
< button onclick =" do_something ()" >Button </ button
<script >
function do_something () {
    let value = document . get Element By Id (" my Input "). value ;
    ws. send (" The value was : " + value );
}
</ script >
```

Build a dashboard that connects to the server and receives position data from the robots and obstacles and displays it in a map and/or a list. Make sure you can give instructions to the robots from the dashboard. Also create an option to place commands in a queue, and let the server or the robots figure out for themselves how to execute these commands most efficiently.

A Final assignment

For the final assignment of Connected Systems, you will have your robot and those of your teammates perform tasks on a playing field provided by the teachers. This playfield consists of a grid of fields of 0.1 × 0.1 m, on which obstacles are placed, surrounded by walls. For an example, see Figure 5. The playfield will be posted on Microsoft Teams during the course. A task could be for your robot to pick something up at one location and then deliver it to another location. This involves avoiding fixed and moving obstacles. These assignments are provided via a dashboard. You and your group decide whether assignments should be distributed to units in the server or units. The same goes for path planning.

The final assignment is made in a team of 4 people. Everyone builds and programs their own simulated and physical unit (see below for description) and together they design a protocol, write a server. and create a dashboard. The robot controller, the server and the dashboard may be written in C, C++, Java, JavaScript or Python as desired.

Requirements The *simulated unit* has the following functionalities:

- The unit can communicate with the server via a self-established protocol.
- The unit can visually indicate its direction.
- The unit can move across the playing field by compartment.
- The unit can detect obstacles, walls and other robots.

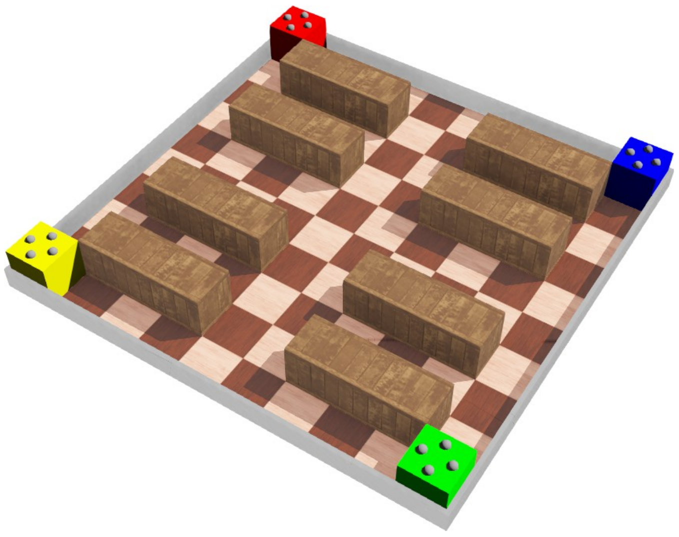


Figure 5: Example of a playfield with 4 robots

- The unit can avoid collision with obstacles, walls or other robots.
- The unit can transmit its own position and direction, and the position of detected obstacles to the server.
- The unit can get the positions of other robots, obstacles and a target through from the server.
- The unit can use this information to plan a route to a target; or the unit can pass on a planned route from the server.
- The units determine among themselves a distribution of tasks to be performed, or leave this to the server.
- Units take into account specific constraints of each unit when distributing assignments. For example, if an assignment is to retrieve an item of a certain mass from a location, this assignment is given only to a unit that has sufficient capacity to do so.

The *physical unit* has the following functionalities:

- The unit can communicate with the server via a self-established protocol.
- The unit can visually indicate which direction the simulated unit is heading.
- The unit is equipped with an emergency stop button that shuts down

the entire system. The *server* has the following functionalities:

- The server waits for incoming connections through a predetermined IP port.
- The server can communicate with the units via a self-established protocol.
- The server keeps the latest versions of all messages until all clients have disconnected.
- The server sends the tracked messages to all clients.
- The server plans a route for the units and relays it; or leaves path planning to the units.
- The server determines the distribution of incoming jobs among the units; or leaves it up to the units.

A system may also be chosen where all units can communicate among themselves via a self-established protocol without a server. In that case, however, the unit will have to take over the functionalities of the server.

The *dashboard* has the following functionalities:

- The dashboard can communicate with the server via a self-defined protocol.
- The dashboard shows a map of the playing field with the positions of units and obstacles.
- The dashboard shows the current command queue.
- Through the dashboard, the purpose of a particular unit can be set.
- Through the dashboard, a job can be queued and deleted.
- An emergency stop can be given via the dashboard.

Delivery Set The delivery set must be turned in to Microsoft Teams. The revival set must consist of the following components:

- All code for the controller, physical unit, server and dashboard
- A description with explanation of the designed protocol
- All Webots proto files
- Any other files.

Assessment For each of the following criteria fully met, 1 point can be earned toward the final grade. All points added together will constitute the final grade for the Connected Systems course.

1. The unit moves in the given simulated space.
2. The unit dodges other units present and obstacles.
3. The unit exchanges its own position data and positions of obstacles with the central server.
4. The unit achieves the specified goal.
5. A display on the unit indicates the desired direction for each step.
6. The desired direction is also indicated on physical hardware, and there is a working emergency stop button there as well.
7. Unit routes are determined by an algorithm.
8. On the dashboard, tasks can be entered for the units and an emergency stop can be given.
9. Information about the units' position, performed and outstanding tasks is displayed on a dashboard.
10. The units divide tasks among themselves according to the specific constraints of each unit.

An extra point may be awarded if the units or dashboard have an additional feature (coordinated with the teacher prior to testing), but only if at least 6 of the above criteria are met.