HOGESCHOOL ROTTERDAM / CMI

Practicumhandleiding Connected Systems

TINCOS01

Cursusjaar: 2020–2021
Auteur: Thijs de Ruiter
Wouter Bergmann Tiest

Inhoudsopgave

Deze practicumhandleiding beschrijft stap voor stap hoe je een Connected System bouwt met behulp van Webots. De stappen werken toe naar de eindopdracht, die beschreven is in appendix A. Zorg dat je iedere week de bijbehorende stap afrond tijdens het practicum op locatie of op eigen gelegenheid thuis. De eindopdracht moet in de toetsweek (week 8) gedemonstreerd worden.

Practicum 1: Installatie en introductie Webots

Tijdens het vak Connected Systems gaan jullie een verbonden systeem bouwen. Omdat het in tijden van Corona niet mogelijk is om dat met hardware te doen zullen jullie deze gaan simuleren. Voor het simuleren gaan jullie gebruik maken van Webots. Webots is een open source multi platform programma waarmee je robots kunt simuleren. Webots ondersteunt de volgende programmeertalen: C, C++, Java, Python en ook Matlab. Jullie zijn vrij om zelf een van deze programmeertalen te kiezen voor het bouwen van de opdracht.

Download Webots op `https://cyberbotics.com/#download`

Wanneer je Webots hebt geïnstalleerd kun je de tutorials van Webots zelf volgen. Deze tutorials geven je een eerste introductie in Webots waardoor je de benodigde vaardigheden opdoet om met Webots aan de slag te gaan. Volg tutorial 1 t/m 7 via de volgende link: `https://cyberbotics.com/doc/guide/tutorials`

Practicum 2: Basisrobot & controller

In Webots kun je zeer complexe robots simuleren met allerlei sensoren en interactie met de omgeving. Voor Connected Systems houden we het simpel: we gaan uit van een raster met vakjes van 0,1 $\times$ 0,1 m, waar de robots zich over kunnen voortbewegen. De robot staat altijd midden op een vakje, zodat de positie van de robot met gehele getallen is aan te geven: het aantal vakjes in de x- en de z-richting. De robot hoeft ook geen wielen te hebben; het voortbewegen doen we door middel van een translatie in de x- en de z-richting. Er is dan ook geen `Physics`-node nodig.

In Webots kun je gebruik maken van de `RectangleArena`. Maak de vakjes 0,1 $\times$ 0,1 m door de `floorTileSize` 0,2 in beide richtingen te maken (één *floortile* bestaat uit 4 vakjes). Geef de arena ook een translatie zodat het midden van het vakje linksboven de coördinaten (0, 0) heeft.

Een robot kan bijv. een blok van 0,1 $\times$ 0,1 $\times$ 0,1 m zijn. Maak een `Robot` met als `children` een `Solid` met een `Shape` naar je eigen wensen, en gebruik dezelfde `Shape` als `boundingObject`. Gebruik hiervoor `DEF`/`USE`; zie Tutorial 2. Om de robot te kunnen transleren vanuit de controller is het belangrijk om `supervisor` op `TRUE` te zetten. Je zou nu iets als in figuur 1 moeten hebben.

Maak nu een controller voor de robot, die de robot naar een opgegeven doel laat bewegen door iedere seconde een stapje te zetten. Om de Supervisor-eigenschappen te kunnen gebruiken, moet je beginnen met:

```
from controller import Supervisor

# create the Robot instance
robot = Supervisor()
supervisorNode = robot.getSelf()
```

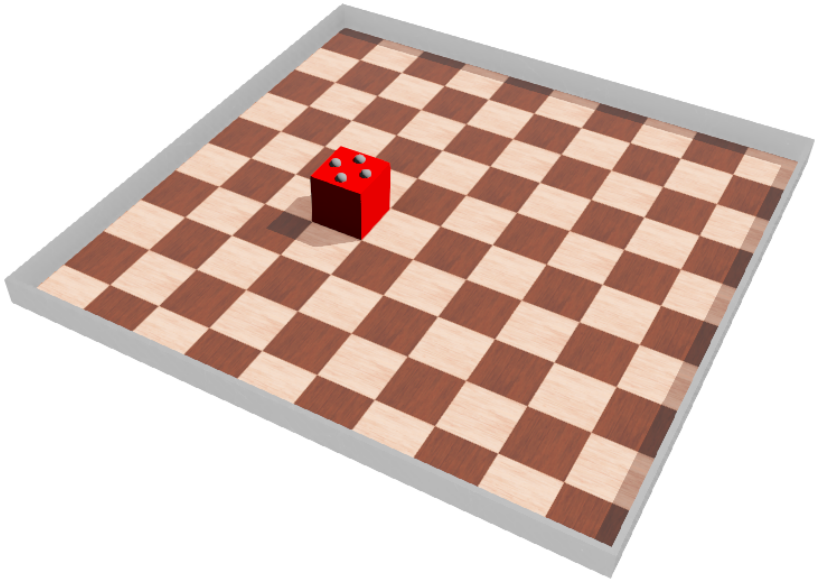
Je kunt een controller maken die elke seconde een stapje zet met:

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

# 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
```



Figuur 1: Een simpele robot op locatie (3, 5) in een arena

Schrijf nu code die de robot naar een doel laat lopen. Als de robot op (0, 0) begint, moet hij bijvoorbeeld naar (9, 9) lopen. De huidige positie van de robot kun je verkrijgen met:

```
# get position
pos = supervisorNode.getPosition()
posX = round(10 * pos[0]) # times 10 because grid size is 0.1 x 0.1 m
posY = round(10 * pos[2])
```

Laat de controller bepalen wat de nieuwe positie wordt voor de huidige stap en zet de robot daar naartoe. Je kunt de robot verplaatsen met:

```
# get handle to translation field
trans = supervisorNode.getField("translation")

# set position; pos is a list with 3 elements: x, y and z coordinates
trans.setSFVec3f(pos)
```

Als de robot zijn doel bereikt heeft, moet hij stil blijven staan. Test je robot en maak er een `PROTO` van (zie Tutorial 7). Aan een `PROTO` kun je ook velden toevoegen die je vanuit de Webots-omgeving kan instellen, bijvoorbeeld:

```
#VRML_SIM R2020a utf8
PROTO Unit [
  field SFString name "unit0"
  field SFColor color 1 0 0
  field SFVec3f translation 0 0 0
  field SFVec2f target 5 5
]
{
  Robot {
    translation IS translation
    name IS name
    children [
      ⋮
    ]
  }
}
```

De waarden van de velden koppel je aan de eigenschappen van de robot met `IS`. Zo kun je per robot een eigen doel en kleur instellen. In de controller kun je het doel weer opvragen met:

```
# get target location
target = supervisorNode.getField("target")
targetVec = target.getSFVec2f()
tarX = int(targetVec[0])
tarY = int(targetVec[1])
```

Zet met de `PROTO` meerdere exemplaren van de robot in verschillende kleuren in de arena, en geef ze ieder een eigen startpunt en doel.

Practicum 3: Protocol

Om informatie te kunnen delen met andere robots, is een set van afspraken nodig over hoe berichten opgebouwd worden. Deze vormen samen het *protocol*. Het idee is dat je zelf je eigen robot programmeert, maar samen met je groepje een gemeenschappelijk protocol afspreekt waar iedereen in de groep zich aan houdt.

Voor het navigeren van de robots door de gesimuleerde ruimte is het handig als de volgende informatie gedeeld wordt met de andere robots:

- De locatie van je robot.
- De locaties van obstakels en wanden in de wereld, voor zover bekend.
- Eventueel: de geplande richting waarin je robot wil gaan bewegen.

Verder kunnen de volgende aspecten van berichten van belang zijn:

- De afzender: krijg ik mijn eigen bericht weer terug of is het nieuwe informatie van een andere robot?
- Het soort bericht: gaat het over de locatie van bewegende robots of over vaste obstakels? In het eerste geval kan ik even wachten als mijn pad geblokkeerd is; in het tweede geval moet ik een ander pad kiezen.
- De datum en tijd: is het een recent bericht of inmiddels achterhaald?
- Eventueel: een versienummer van het protocol: de versie van het protocol bepaalt hoe de informatie geïnterpreteerd moet worden.

Een protocol moet heel precies omschreven worden, zodat het niet op verschillende manieren geïnterpreteerd kan worden. Denk hierbij aan welke leestekens je gebruikt om velden te scheiden, waar spaties of newlines mogen voorkomen, hoe het eind van het bericht aangegeven wordt, enz. Dit wordt opgeschreven in de *specificatie*. Bijvoorbeeld:

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

Ontwerp samen met je groepje een protocol voor de uitwisseling van gegevens tussen jullie robots. Noteer dit in een specificatie.

Practicum 4: Centrale server

Om verschillende robots met elkaar (op afstand, in hun eigen omgeving) te laten communiceren, kunnen ze contact maken met een centrale server, en zo elkaar berichten sturen over hun positie en richting. Zo kunnen ze elkaar ontwijken en elkaar informatie geven, bijvoorbeeld over de locatie van een obstakel.

Voor de server kunnen TCP/IP-sockets gebruikt worden. Deze werken via een zelf te kiezen poort. Poortnummers 1024 en hoger zijn vrij beschikbaar. Spreek met je groep een poortnummer af. In Python start je een server met:

```
import socket

HOST = ""
```

```
PORT = 1024

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

Vervolgens wacht de server op verbinding met een client met:

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

`conn` is een `socket`-object dat gebruikt kan worden om data te versturen en te ontvangen. `addr` bevat het adres en poortnummer van de client. Om data te ontvangen van de client gebruik je:

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

Data (als string) verstuur je met:

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

De client kan verbinding leggen met de server op deze manier:

```
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()
```

Vervolgens kan de client data versturen en ontvangen met `s.sendall()` en `s.recv()`.

Om met meerdere clients contact te houden, is het handig dat de server voor iedere verbinding een andere thread start. Hier is een voorbeeld van hoe dat zou kunnen:

```
import socket
import threading

HOST = ""
PORT = 1024

messages = {}
clientCount = 0

def echo(conn):
    global messages, clientCount
    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
    clientCount -= 1
    if clientCount == 0:
        print("All clients disconnected; resetting")
```

```
        messages = {}

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

Dit programma neemt input van clients in de vorm `afzender: bericht` en zet dit in een gemeenschappelijke *dictionary* `messages`. Als er een nieuw bericht komt met dezelfde afzender, wordt het vorige bericht van die afzender overschreven. Vervolgens worden alle berichten teruggestuurd naar de client als een string. Deze string is door de client weer om te zetten naar een dictionary met de functie `eval()`.

De bedoeling is dat verschillende robots via de server met elkaar communiceren, elk in hun eigen Webots-simulatie die draait op een aparte computer. In eerste instantie wil je dit waarschijnlijk testen met meerdere robots binnen één Webots-simulatie. Er zijn dan meerdere instanties van het controller-programma actief, voor elke robot één. Het is dan handig dat niet alle controllers precies op hetzelfde moment starten, zodat de robots één voor één hun stapjes zetten. Dit kun je bereiken door voordat de eigenlijke controller-lus begint, een willekeurig aantal simulatiestappen over te slaan:

```
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 de controller-lus moet een robot zijn eigen positie en richting aan de server doorgeven, en de posities van alle andere robots terugkrijgen. Hiervoor is het belangrijk dat iedere robot een unieke naam heeft. Die kun je in Webots instellen in het `name`-field van de robot. In de controller kun je die naam opvragen met `robot.getName()`.

Gebruik nu het protocol dat je vorige les ontworpen hebt om vanuit jouw robot-controller informatie over positie en richting naar de server te sturen, en informatie van de andere robots terug te krijgen. Op basis hiervan kan de controller de voorgenomen richting aanpassen of besluiten even te wachten, zodat botsingen vermeden worden. Test dit door verschillende robots tegelijk over het bord te laten lopen naar hun eigen doel.

Als dit lokaal werkt, laat dan met je groepje ieders robot in zijn eigen Webots-omgeving rondlopen en contact maken met de gezamenlijke server. Om de remote-robots zichtbaar te maken, kun je lokaal een robot toevoegen die precies de bewegingen van een remote-robot kopieert. De controller van deze robot stuurt zelf geen data, maar ontvangt alleen maar. Om de server te triggeren kun je een dummy-bericht sturen: `"dummy: no content"`. Start de simulaties tegelijk en kijk of de robots elkaar correct ontwijken.

Practicum 5: Indicatoren & sensoren

Robots moeten vaak samenwerken met mensen, en dan is interfacing tussen de mens en de robot belangrijk. De mens moet de robot vertellen wat hij moet doen, en de robot moet aangeven wat hij gaat doen om gevaarlijke situaties te voorkomen.

In Webots kunnen we een robot bijvoorbeeld LEDjes geven waarmee hij kan aangeven in welke richting hij wil bewegen. Van een LED kun je een `PROTO` maken zodat je hem meerdere keren kunt gebruiken:

```
#VRML_SIM R2020a utf8
PROTO SimpleLED [
  field SFString name "led0"
  field MFColor color [
    1 0 0
  ]
  field SFVec3f translation 0 0 0
```

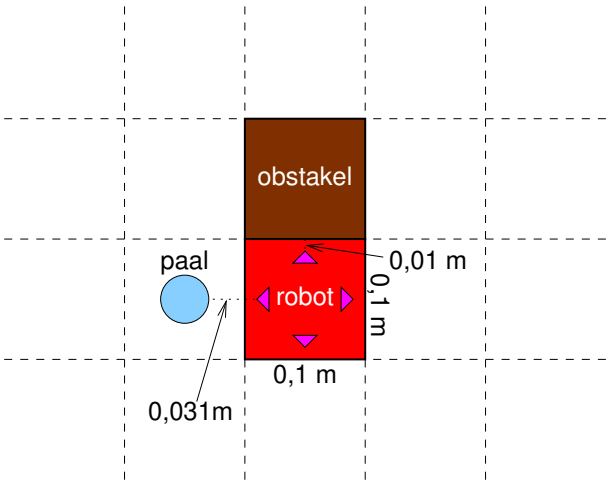
```
]
{
  LED {
    translation IS translation
    children [
      Shape {
        appearance Appearance {
          material Material {
            diffuseColor 0.5 0.5 0.5
          }
        }
        geometry Sphere {
          radius 0.01
        }
      }
    ]
    name IS name
    color IS color
  }
}
```

Zo kun je 4 LEDs maken die je allemaal een verschillende naam geeft. Geef ze een translatie zodat ze bovenop de robot staan, ieder in een eigen richting (zie ook figuur 1). In je controller kun je de LED dan bedienen met:

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

Zorg ervoor dat bij iedere stap die de robot doet, de juiste LED gaat branden.

Om obstakels te kunnen detecteren, kan je robot gebruik maken van afstandssensoren. Standaard heeft de in Webots ingebouwde `DistanceSensor` een range van 0–0,1 m (0–1000 eenheden). Als je aan iedere kant van je robot er één plaatst, kun je dus precies zien of zich in de vier velden om je robot heen een obstakel, een muur of een andere robot bevindt. Zet de sensoren niet precies op de buitenkant van de robot, want als deze precies tegen een obstakel staat, heb je kans dat de sensor het obstakel niet ziet door afrondingsfouten. Zet ze bijvoorbeeld 0,04 m uit het midden; zie ook figuur 2. Als de `DistanceSensor` dan een waarde kleiner dan 1000 (0,1 m) geeft, dan zit er een obstakel of muur naast de robot. Door naar de waarde te kijken, kun je achterhalen wat voor obstakel het is (bijv. een paal of een doos). Is de waarde precies 1000, dan ziet de sensor niets.



Figuur 2: Afstandssensoren in een robot

Je kunt de sensor gebruiken met:

```
ds = robot.getDevice("sensorName") # replace with name of sensor
ds.enable()

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

Rust je robot uit met 4 afstandssensoren en verwerk de data in de controller. Stuur berichten naar de server als je robot een obstakel ontdekt. Het is het handigst om de positie van nieuwe obstakels toe te voegen aan de reeds bekende (bijv. met `set.union()`), en de hele lijst steeds met de server te delen. Let er ook op dat als je robot andere robots detecteert, deze niet als obstakel gezien worden zodat er alleen stilstaande obstakels in de lijst staan. Dit kun je doen door de posities van andere robots uit de lijst te filteren.

Practicum 6: Path planning

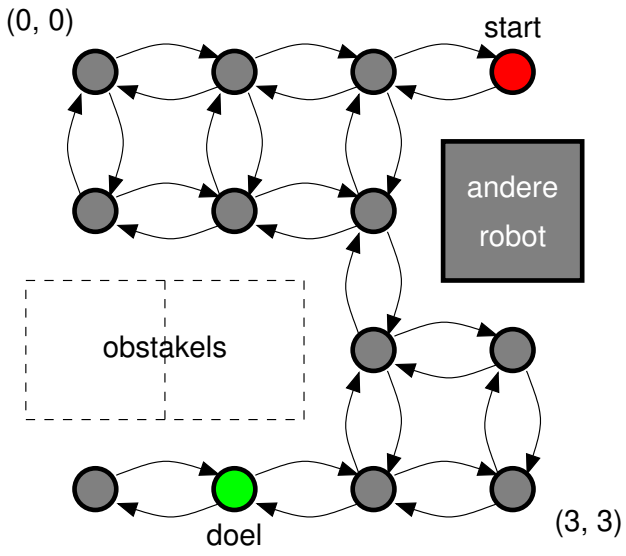
Tijdens deze les gaan wij aan de slag met het zo efficiënt mogelijk plannen van de route van je robot naar het opgegeven doel. Deze les worden er een aantal algoritmes besproken.

Om een route te kunnen bepalen moet er als eerste een kaart beschikbaar zijn waarmee de route berekend kan worden. Daarvoor is een datastructuur nodig. Een mogelijke datastructuur hiervoor zijn de grafen. Grafen kunnen worden gebruikt om de relaties tussen de verschillende objecten aan te geven. Deze objecten zullen in ons geval de verschillende coördinaten op het speelveld zijn.

Een graaf (Engels: graph) bestaat uit vertices en edges. Vertices zijn met elkaar verbonden door middel van de edges. Deze edges kunnen een richting hebben en/of een gewicht. Op een graaf kunnen algoritmes worden toegepast om de eigenschappen de graaf te berekenen of voorspellingen te doen. Zo kan onder andere een route bepaald worden door middel van algoritmes die worden toegepast op een graaf. De volgende algoritmes kun je gebruiken voor je robot:

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

Let op dat niet elk algoritme in elke situatie werkt; zo werken sommige algoritmes niet wanneer er een cirkel in de kaart zit. Bekijk alle algoritmes en zoek uit in welke gevallen ze niet werken en welke het meest geschikt is voor jouw robot.



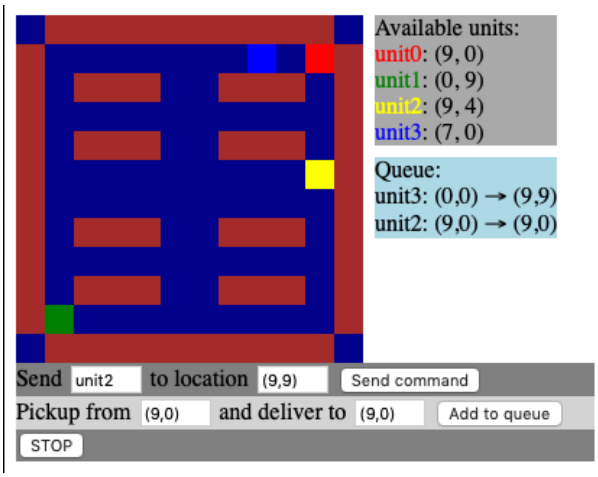
Figuur 3: Een graaf met 13 nodes en 30 links

Zogauw jouw robot een opdracht krijgt, kun je een route gaan plannen naar het doel door een graaf te maken en één van de bovenstaande algoritmes toe te passen. Om een graaf van het speelveld te maken, gebruik je

de positie van alle robots en informatie over de lay-out van het speelbord, maar ook informatie die je eigen robot en de andere robots verzameld hebben over de positie van onverwachte obstakels. Je mag de afmetingen van het speelbord (bijv. 10×10) als vast gegeven beschouwen. Maak voor ieder veld een *node* (vertex) en zet deze in een lijst. Een node kan één of meer *links* (edges) naar andere nodes hebben. In principe kan een link natuurlijk twee kanten op (een link van node *A* naar node *B* is ook een link van node *B* naar node *A*), maar in de praktijk is het vaak handiger om met links te werken die maar één kant op gaan, en die dan dubbel uit te voeren. Maak voor iedere node links naar de naburige nodes, tenzij zich daar een obstakel of robot bevindt; zie figuur 3. In dit geval kan iedere link hetzelfde gewicht (lengte) hebben, bijv. 1. Door als start-node de positie van je eigen robot te gebruiken en als doel-node de positie van het doel, kun je de route bepalen en een stap in de goede richting zetten. Het is aan te raden om dit iedere stap opnieuw te doen, omdat er steeds meer informatie over de obstakels bekend wordt, en de andere robots ook van positie veranderen.

Practicum 7: Dashboard

Om de informatie over de robots te tonen en ze opdrachten te geven, is er een dashboard nodig. Dit kun je bouwen als een GUI-applicatie, maar je kunt het ook in een webbrowser laten draaien. Dat maakt het verspreiden erg makkelijk. Op het dashboard kan een kaart getekend worden met de posities van alle robots en obstakels. Je kunt laten zien welke opdrachten er in de *queue* staan en welke robot daar mee bezig is. Er moet een mogelijkheid zijn om een robot naar een lokatie te sturen, of opdrachten in de queue te plaatsen zoals “haal een item op van lokatie (3,2) en breng dit naar lokatie (7,4)”. Tenslotte moet er een knop voor een noodstop zijn. Een voorbeeld van een dashboard is te zien in figuur 4.



Figuur 4: Voorbeeld van een dashboard

Als je het dashboard in een webbrowser maakt, ligt het voor de hand om voor de verbinding met de server websockets te gebruiken. Hiervoor kun je je server uitbreiden met een 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()
```

Als een client verbinding maakt met deze server en de tekst `request_messages` stuurt, wordt de dictionary `messages` omgezet naar JSON-formaat en teruggestuurd. In de browser kun je met JavaScript een websocket-client maken:

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

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

ws.onmessage = function(message) {
    let data = JSON.parse(message.data);
    // doe iets met de data
}

setInterval(get_data, 1000);
</script>
```

Deze code stuurt iedere seconde de tekst `request_messages` naar de server. Als er een bericht terugkomt, wordt de functie `ws.onmessage()` aangeroepen. Hier wordt de data van JSON-formaat omgezet naar een JavaScript-object, waar je vervolgens iets mee kan doen.

Bijvoorbeeld voor het tekenen van een kaart kun je een HTML canvas gebruiken:

```
<canvas id="theCanvas" width="240" height="240" style="background-color: darkblue">
</canvas>
<script>
CanvasRenderingContext2D.prototype.clear = function() {
    this.clearRect(0, 0, this.canvas.width, this.canvas.height);
};
CanvasRenderingContext2D.prototype.drawBlock = function(x, y) {
    this.fillRect(20 * x, 20 * y, 20, 20);
}

let context = document.getElementById("theCanvas").getContext("2d");

function draw() {
    context.clear();
    context.fillStyle = "red";
    context.drawBlock(3, 5); // Vervang door waardes afkomstig van de server
}
</script>
```

Met bovenstaande code worden twee extra functies toegevoegd aan de context van het canvas: `clear()` en `drawBlock(x, y)`. `context.clear()` wist het canvas zodat je een nieuwe tekening kan maken. `context.drawBlock()` tekent een vierkantje van 20 $\times$ 20 pixels op de aangegeven coördinaten (in dit voorbeeld tussen de 0 en 11). De kleur stel je in met `context.fillStyle`. Met deze functies kun je de kaart tekenen in de functie `draw()`, die je aanroept als een nieuw bericht binnenkomt.

Je kunt op het dashboard ook tekst weergeven. De makkelijkste manier om dynamisch op een webpagina tekst weer te geven is om een `<div>` te maken en hiervan de `innerHTML`-property aan te passen:

```
<div id="myDiv" style="background-color: darkgray"></div>
```

```
<script>
document.getElementById("myDiv").innerHTML = "Deze tekst verschijnt op de plaats van de &lt;div&gt;.";
</script>
```

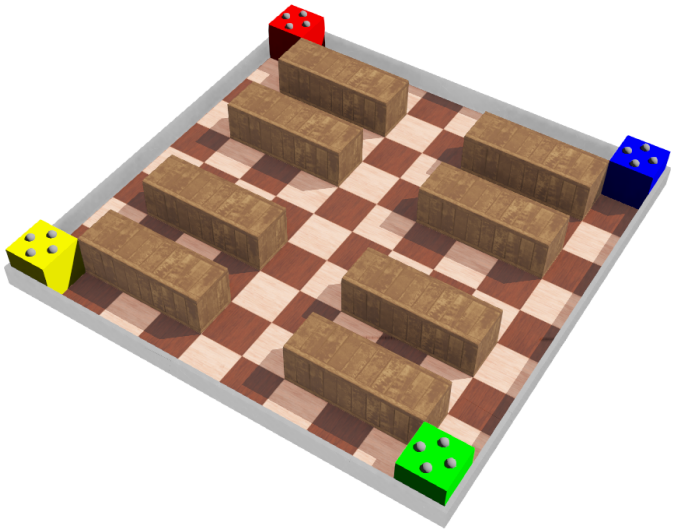
Om input-velden en knoppen te gebruiken kun je onderstaande code gebruiken:

```
<input type="text" id="myInput" size="6">
<button onclick="do_something()">Knop</button>
<script>
function do_something() {
    let value = document.getElementById("myInput").value;
    ws.send("The value was: " + value);
}
</script>
```

Bouw een dashboard dat verbinding maakt met de server en positiegegevens van de robots en obstakels ontvangt en deze weergeeft in een kaart en/of een lijst. Zorg dat je de robots vanaf het dashboard instructies kan geven. Maak ook een mogelijkheid om opdrachten in een queue te plaatsen, en laat de server of de robots zelf bedenken hoe ze deze opdrachten het efficiënts kunnen uitvoeren.

A Eindopdracht

Voor de eindopdracht van Connected Systems laat jij je robot samen met die van je teamgenoten opdrachten uitvoeren op een door de docenten aangeleverd speelveld. Dit speelveld bestaat uit een raster van velden van 0,1 $\times$ 0,1 m, waarop obstakels geplaatst zijn, omringd door muren. Voor een voorbeeld, zie figuur 5. Het speelveld wordt tijdens de cursus geplaatst op Microsoft Teams. Een opdracht kan zijn dat je robot iets ophaalt op een lokatie en dit vervolgens aflevert op een andere lokatie. Daarbij moeten vaste en bewegende obstakels vermeden worden. Deze opdrachten worden via een dashboard verstrekt. Je bepaalt zelf met je groep of het verdelen van de opdrachten over de units in de server of de units moet gebeuren. Hetzelfde geldt voor de path planning.



Figuur 5: Voorbeeld van een speelveld met 4 robots

De eindopdracht wordt gemaakt in een team van 4 personen. Iedereen bouwt en programmeert zelf een unit (zie hieronder voor beschrijving) en samen wordt er een protocol ontworpen, een server geschreven. en een dashboard gemaakt. De robot controller, de server en het dashboard mogen naar keuze in C, C++, Java, JavaScript of Python geschreven worden.

Requirements De *unit* heeft de volgende functionaliteiten:

- Een unit kan communiceren met de server via een zelf opgesteld protocol.
- Een unit kan visueel aangeven welke richting deze opgaat.
- Een unit kan over het speelveld bewegen per vak.
- Een unit kan obstakels, muren en andere robots detecteren.
- Een unit kan een botsing met obstakels, muren of andere robots voorkomen.
- Een unit kan zijn eigen positie en richting, en de positie van gedetecteerde obstakels doorgeven aan de server.
- Een unit kan de posities van andere robots, obstakels en een doel doorkrijgen van de server.
- Een unit kan deze informatie gebruiken om een route te plannen naar een doel; of een unit kan een geplande route doorkrijgen van de server.
- De units bepalen onderling een verdeling van de uit te voeren opdrachten, of laten dit over aan de server.
- De units houden bij het verdelen van de opdrachten rekening met specifieke beperkingen van iedere unit. Bijvoorbeeld, als een opdracht is om een item met een bepaalde massa van een lokatie op te halen, dan wordt deze opdracht alleen gegeven aan een unit die hiervoor voldoende capaciteit heeft.

De *server* heeft de volgende functionaliteiten:

- De server wacht op binnenkomende verbindingen via een vooraf bepaalde IP-poort.
- De server kan met de units communiceren via een zelf opgesteld protocol.
- De server houdt de laatste versies van alle berichten bij totdat alle clients de verbinding verbroken hebben.
- De server stuurt de bijgehouden berichten naar alle clients.
- De server plant een route voor de units en geeft deze door; of laat de path planning over aan de units.
- De server bepaalt de verdeling van de binnenkomende opdrachten over de units; of laat dit over aan de units.

Er mag ook gekozen worden voor een systeem waar alle units onderling kunnen communiceren via een zelf opgesteld protocol zonder server. In dat geval zullen de unit wel de functionaliteiten van de server moeten overnemen.

Het *dashboard* heeft de volgende functionaliteiten:

- Het dashboard kan communiceren met de server via een zelf opgesteld protocol.
- Het dashboard toont een kaart van het speelveld met de posities van de units en obstakels.
- Het dashboard toont de huidige opdrachtenqueue.
- Via het dashboard is het doel van een bepaalde unit in te stellen.
- Via het dashboard is een opdracht in de queue te plaatsen en weer te verwijderen.
- Via het dashboard kan een noodstop gegeven worden.

Opleverset De opleverset moet worden ingeleverd op Microsoft Teams. De opleverset moet bestaan uit de volgende onderdelen:

- Alle code voor de controller, de server en het dashboard
- Een beschrijving met uitleg van het ontworpen protocol
- Alle Webots protobestanden
- Eventuele andere bestanden.

Beoordeling Voor elk van de volgende criteria waaraan volledig is voldaan kan er 1 punt worden verdiend voor het eindcijfer. Alle punten bij elkaar opgeteld vormen het eindcijfer voor het vak Connected Systems.

1. De unit beweegt zich in de gegeven gesimuleerde ruimte.
2. De unit ontwijkt andere aanwezige units en obstakels.
3. De unit wisselt eigen positiegegevens en posities van obstakels uit met de centrale server.
4. De unit bereikt het opgegeven doel.
5. Een display op de unit geeft de gewenste richting aan voor iedere stap.
6. De routes van de units worden bepaald door middel van een algoritme.
7. Op het dashboard kunnen taken voor de units ingegeven worden en kan een noodstop gegeven worden.
8. Informatie over de positie, de uitgevoerde en de nog openstaande taken van de units wordt afgebeeld op een dashboard.
9. De units verdelen onderling taken afgestemd op de specifieke beperkingen van iedere unit.
10. De units hebben een extra feature (voorafgaand aan de toetsing afgestemd met de docent).