ISSN 0280–5316
ISRN LUTFD2/TFRT--5604--SE

Remote Control of a Mobile Robot Using Petri Nets

Anders Karlsson

Department of Automatic Control
Lund Institute of Technology
December 1998

Department of Automatic Control **Lund Institute of Technology** **Box 118** **S-221 00 Lund Sweden**	*Document name* MASTER THESIS *Date of issue* December 1998 *Document Number* ISRN LUTFD2/TFRT--5604--SE
Author(s) Anders Karlsson	*Supervisor* Anders Wallén, Pedro Lima (IST, Lisbon) *Sponsoring organisation*

Title and subtitle

Remote Control of a Mobile Robot Using Petri Nets
(Fjärrstyrning av rörlig robot med hjälp av Petrinät)

Abstract

In mobile robotics there is a need of new ways to do the control. Before it was often very hard to program the mobile robots and the programmed robots were also not very flexible. We need a system that is easy to apply, general and flexible.

Petri nets can be used to model all missions available in mobile robotics. Its way to handle parallel actions among other things is superior the more traditional state graph approach. There should be defined a set of primitive tasks for the robot that could be used to solve a mission. These tasks should each one be defined in a way that they can not be divided into smaller tasks. The robot should then offer one or more primitive actions for each primitive task that is defined for the robot. Each primitive action should solve the problems defined in its primitive task.

In this thesis a mobile robot called Centauro has been used. It is a differential drive mobile robot equipped with batteries, two electrical motors, a computer and a CCD camera.

A Petri net executor has been implemented, a set of primitive tasks have been defined for Centauro and one primitive action has been implemented for each primitive task. The system is then programmed by building up a Petri net, associating primitive tasks to the different places and setting up the conditions for the transitions. The result of this project is a mobile system using Petri nets, primitive tasks and primitive actions that is easy to use and program. It also lets the human operator control the mobile robot directly with a six degree of freedom input device. Digital image recognition is used to feed back the results of the commands given to the mobile robot.

Key words

Autonomous vehicles, mobile robots, Petri nets, supervisory control, primitive tasks, primitive actions

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280–5316		*ISBN*
Language English	*Number of pages* 40	*Recipient's notes*
Security classification		

Remote control of a mobile robot using Petri Nets

Anders Karlsson

Department of Automatic Control

Lund Institute of Technology

and

Institute for Systems and Robotics

Instituto Superior Técnico

October 98

1 Introduction

There is an increasing interest in using mobile robots in more and more applications. These ranges from automated vacuum cleaners to unmanned space missions, such as Path Finder[Golombek 1998].
In some of these applications we need an operator. The operator could inspect that the automation works well, do some operations that are complicated to program in advance or be called upon when things go wrong.
It is desirable that the operator is able to control the robot remotely from a terminal. Then the operator would be able to supervise more than one robot at a time. Other advantages are that the robot could be located in environments hazardous to humans, or that we could send away robots on missions without return.
The robot should contain a set of small primitive functions. These should all solve a small primitive task. With these functions it is possible to solve bigger problems by putting them together in a program. This program could be done with Petri nets, which gives an easy way of programming sequential algorithms.
We should later add redundant functions to the set of functions. We can then let the system learn which functions work best at which places with reinforcement learning. This part is not implemented in this project, but my design of Centauro is prepared for reinforcement learning and can be used as soon as it is implemented in the Petri net executor.

The methodology to use Petri nets and primitive tasks was developed in [Lima and Saridis, 1996] and [Wang and Saridis, 1993]. A short description will be given in Chapter 2. In this project, the methodology was applied to the mobile robot Centauro, see Figure 1.1. Centauro has a differential drive kinematic configuration with two independent actuated wheels and two free support wheels. The velocity difference between the two drive wheels makes Centauro turn. If they are both running at the same speed Centauro runs straight.
Centauro uses one CCD camera and a mirror. This gives the possibility of both a distant and an immediate foreground view of Centauro.

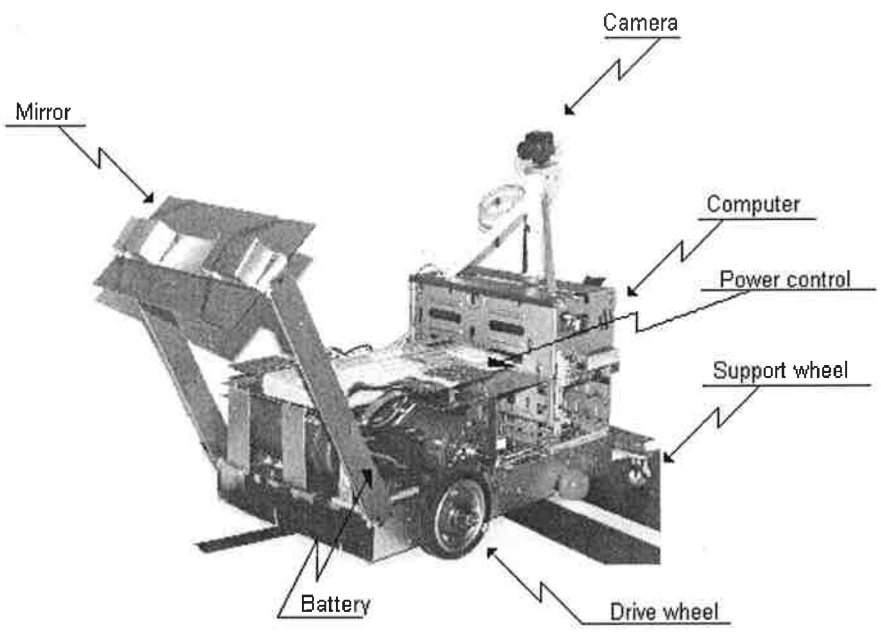


Figure 1.1 Picture of Centauro.

1.1 Goal

The goal is to add remote control to the already existing mobile robot Centauro. The remote control should be done with Petri nets built up with primitive actions. It should also be possible for the operator to control the robot directly with a six degrees of freedom joystick. The Petri nets should be executed in a stationary computer, which should send requests to execute primitive tasks in the mobile robot.

2 Modelling robotic primitive tasks with Petri Nets

In all automation there is a need of a tool to sequence the operations in some way. Often it is also important to be able to make decisions during the on-line operation, for example to discriminate parts that doesn't fulfil the quality requirements. To co-ordinate different primitive actions at the same time we have to do things in parallel.
There are a lot of different ways to make this happen. One of these is to use Petri nets.

2.1 Petri-Nets

Petri nets are basically built up by places and transitions. The places are drawn like circles, each one having a primitive task associated with it. The place is connected to a transition, which is connected to another place and so on.

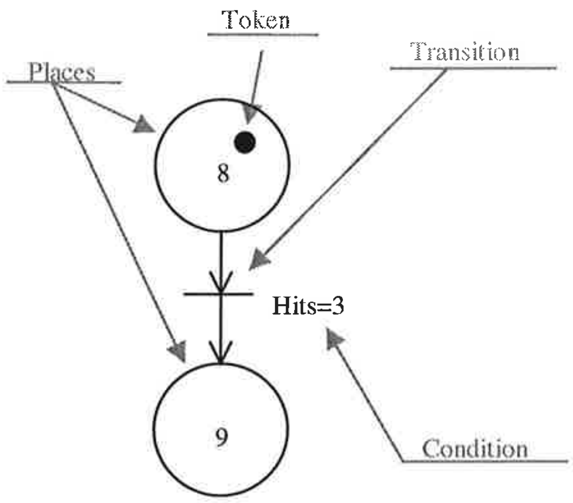


Figure 2.1 Petri net with two places and one transition.

In the Petri net there should be one or more tokens running around. They stay in a place until a transition connected to an outgoing arc from the place becomes true. In the Figure 2.1 we have the token in place 8. The token will move from place 8 to place 9 when `Hits` becomes equal to 3.
By using weighted transitions, we are able to create and destroy tokens in an orderly fashion. In Figure 2.2, we must have 3 tokens in place 8 before we are able to fire transition `Ball=blue`. When firing `Ball=blue`, place 8 will lose three tokens and place 10 will gain one token because of the weights.
In place 10, one token is enough to fire the transition `Ball=gone`, and then one token disappears in place 10 and two tokens will appear in place 11.
After we have gone through Figure 2.2 we have put in 3 tokens in place 8 and we will have 2 tokens in place 11.
To summarise, we must put two additional tokens in place 8 to start the execution of the net in Figure 2.2. After we have gone through the net, we will have two tokens in place 11.

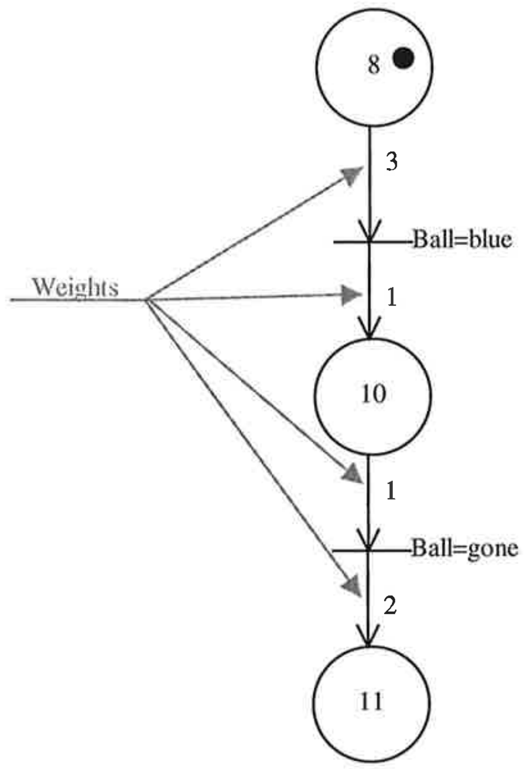


Figure 2.2 Petri net with weighted transitions

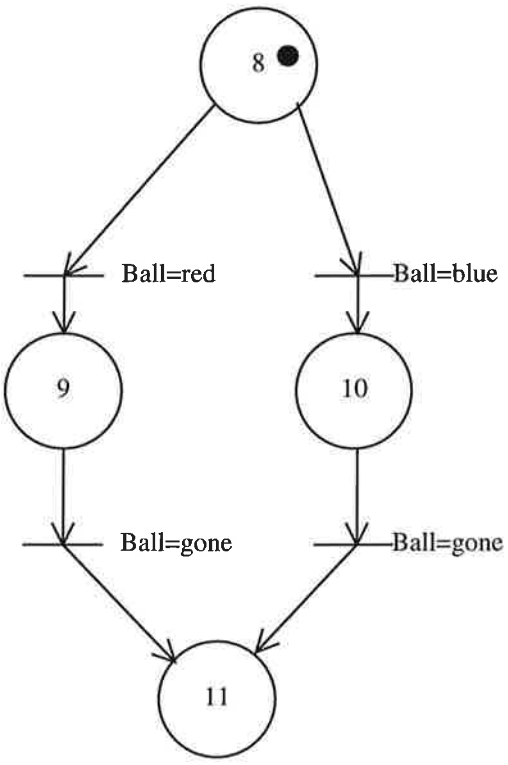


Figure 2.3 Petri net with alternative paths

In Figure 2.3 the token is placed in place 8. The token moves to place 9 if the variable `Ball` becomes red and it moves to place 10 if `Ball` becomes blue. The token continues to place 11 when the variable equals gone again. In the robot application, alternative paths may for example be used when Centauro should decide which track to follow.

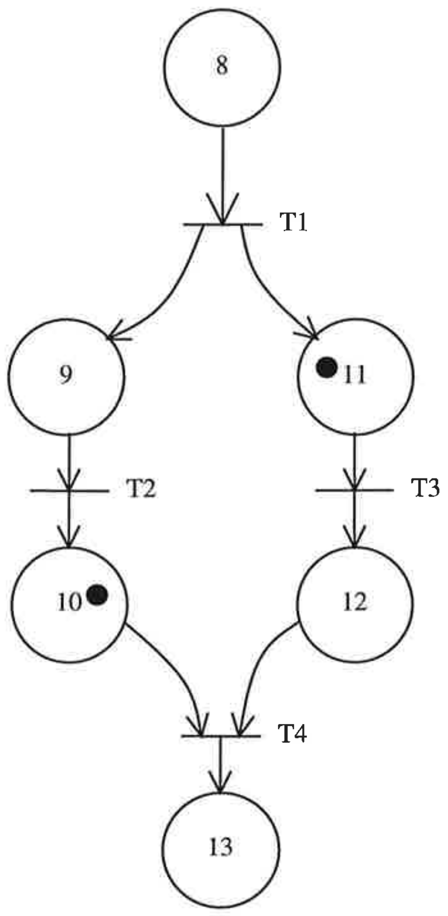


Figure 2.4 Petri net with parallel paths.

The Petri net in Figure 2.4 is a simple example of how to model parallel activities. When T1 has fired the branch with the places 9 and 10 will execute in parallel with the branch with the places 11 and 12. T4 may fire when there are tokens both in place 10 and in place 12. The parallel branching can be very useful in many automation procedures. One benefit is the clear representation compared to a totally sequential modelling. You may also gain in performance, for example if the times to perform the primitive actions in steps 9 through 12 are unknown or if they are varying. One possible application could be a large assembly machine where some pieces may be assembled independently. In the Centauro system, it could be used in the future to synchronise the robot with events taking place in its environment.

All of the classical real-time problems may be modelled using Petri nets, for example

- mutual exclusion in critical regions,
- synchronisation of concurrent processes (rendez-vous and so on),
- readers-writers problem,

- producers-consumers problem.

These problems can be solved by using parallel paths and multiple tokens, see [David and Alla, 1992] for details.

2.2 Primitive actions and primitive tasks

The previous section introduced Petri nets and how they are constructed. This section will discuss the primitive actions that are associated with the Petri net places.

A primitive task is a small sub-problem that can't be divided into smaller problems, from a functional standpoint. A primitive action is an implementation of such a primitive task, i.e. a specific way of solving the primitive task.

The most natural way to divide a complete mission into primitive tasks is to find clear divisions in the functional behaviour. One way to find the primitive tasks is to look at a mission, and try to identify what repetitious procedures that shall be done. You may then divide the mission into smaller parts where the robot should change focus.

The primitive task to follow a track could be an example of one such primitive task. This primitive task could be implemented as different primitive actions. For example, the robot could follow a track using feedback from a camera or just by following a predefined trajectory in open loop.

Every primitive task always has one or more primitive actions associated with it, and every primitive action is always associated with one primitive task. The different primitive actions associated with a primitive task are different ways to solve the primitive task.

This architecture can be used for reinforcement learning. A primitive action can give three different results

- ERROR: it did nothing, or with very poor result (e.g., because it was trying to follow a path without object detection and bumped into a large object).
- FAILURE: it did not very well (e.g., specifications not met completely), but fair enough to allow the continuation of primitive task execution.
- SUCCESS: it met the specifications of the corresponding primitive task.

More about how this is supposed to be used can be found in [Lima and Saridis, 1996].

2.3 Data handling

We have decided not to allow any parameters to the primitive actions. The knowledge is supposed to be located in Centauro or in a central database instead. The information is supposed to be down in the bottom of the architecture and not sent down from the Petri net program to the primitive actions. For example the primitive action `rotate_x_degrees` is not allowed. Instead you should have a

function `rotate_to_desired_direction`, which looks at the variable `desired_direction`.
Every separate part of the system has to take care of its own data. It may send information to other parts by sending updates on variables. It may also ask other parts of the system what the values are on different variables. The protocol of this data transfer is described in chapter 3.3.

3 Centauro

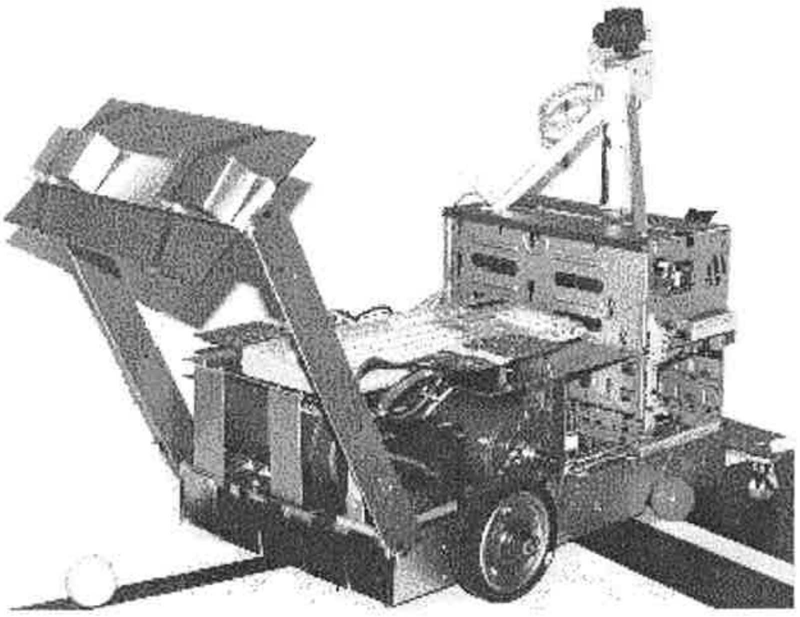


Figure 3.1 A picture of Centauro.

3.1 Original purpose

Centauro was built to compete in a robot competition held in 1997. The main goal in this competition was to follow a track drawn on a gigantic chessboard as fast as possible. In addition every robot should collect balls of a specific colour and discriminate ball of another colour placed on the track. More information is found in [Lima *et al.,* 1997].

3.2 Hardware

The hardware is built by the students from Instituto Superior Técnico (IST) in Lisbon and is very much an experimental vehicle. Basically, it uses a CCD camera, two large electrical motors and a computer. I have later extended the vehicle with one Ethernet card connected to a radio modem in order to make it possible to perform remote control. This chapter describes the hardware configuration briefly. A detailed description is found in [Lima *et al.,* 1997].

Camera

The camera is an EDC-1000M. It takes 5 frames per second with the resolution of 324 x 242 with 256 grey scales. You may reduce the number of lines in order to make it work faster.

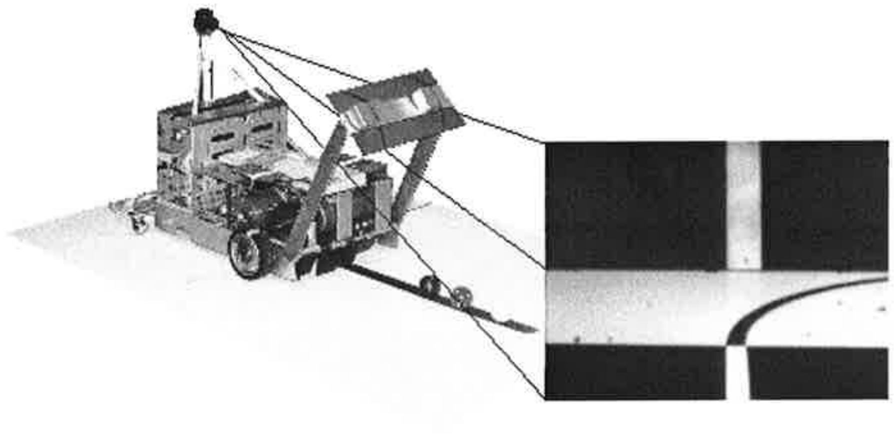


Figure 3.2 Illustration of how the camera looks through the mirror

Mirror

Centauro is equipped with a mirror in order to both get a view straight ahead, and also one of the track underneath the robot.. The mirror makes the upper half of the video image show what Centauro has right in front of itself, and the lower half of the image shows what Centauro can se further away, see Figure 3.2 and 3.3. This concept makes it possible to have a very good view of the immediate surroundings but also to be prepared of what will emerge in the near future. The image in the mirror ranges from 36 cm to 50 cm ahead of the driving wheels. The width of the image is 53 cm.

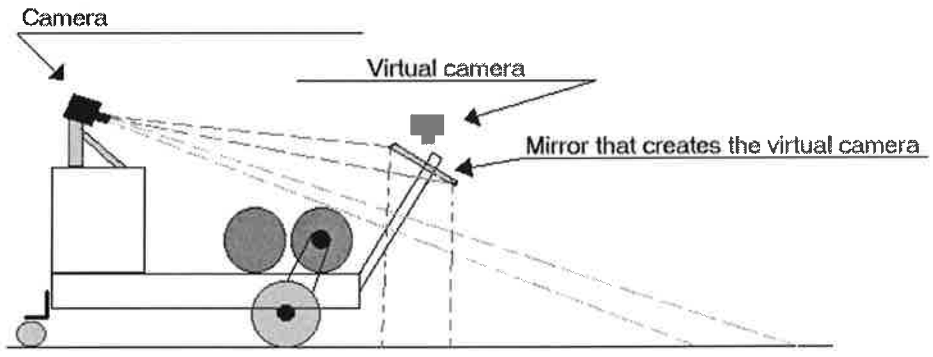


Figure 3.3 Vision system of Centauro

Motors

To cope with the relatively high speed requirements that Centauro originally had (above 1 m/s), and the considerable weight of the vehicle (50 Kg), it is equipped with two 12 V, 1750 rpm, 2 Nm Pacific-Scientific DC motors. They are further described in [Lima, Silva, Santos and, Cardeira 1997].

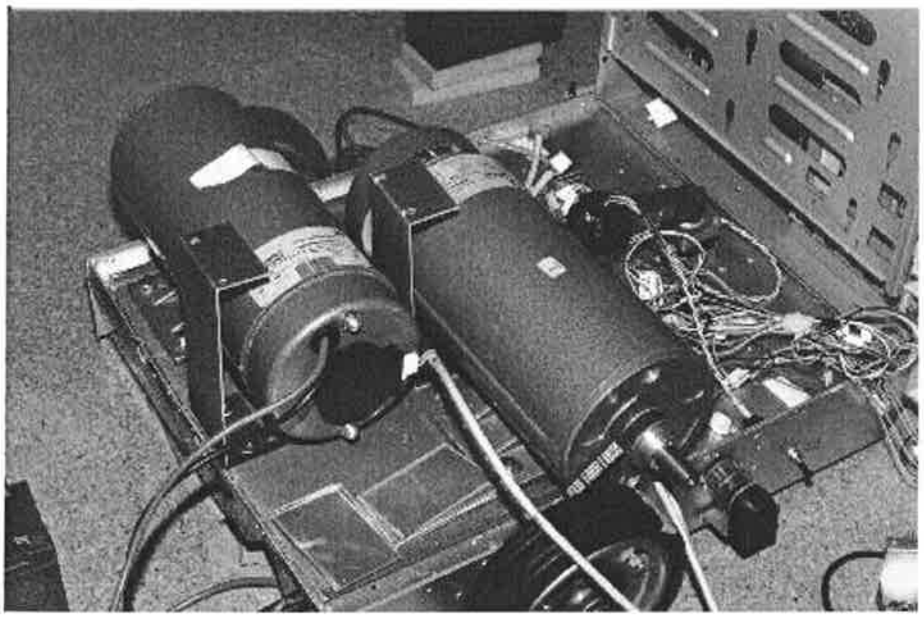


Figure 3.4 The two motors of Centauro, each one connected to its own wheel.

Batteries

There are three batteries of lead type. The computer and the two motors each have their own battery. This guarantees that the three units will not affect each other through the power system. A thorough description of the choice of batteries can be found in [Lima *et al.,* 1997].

Power controller

The power controllers for the motors "NCC Professional 120 Motor Controller" are built by 4QD. More information is found at [http://www.argonet.co.uk/users/4qd/].

Interface to the power controller

The interfaces to the power controllers were built at IST. They are connected to the ISA-bus that was introduced in the early eighties in the IBM PC. They let you set either desired position, velocity or acceleration of each motor. You may read the exact position of the motors. This gives you the possibility to control Centauro.
The interface is programmed by sending the command to the port and then to wait for the interface to acknowledge the command. This behaviour makes the communication with the power controller a bit slow. A lot of time would have been saved if the communication had used interrupts instead.

Computer on Centauro

The computer on Centauro is a standard PC with a special power supply. The power supply is connected to one of the 12 V batteries and gives 5V and 12V.

Processor	Am486DX4-S
Memory	4MB
Hard disk	52MB
Network	10Mbit Ethernet added to make remote control possible
OS	MS-DOS

3.3 System software

Centauro is a real-time system. It interacts with reality with the control loop. The movement is fed back from the camera, via the computer and finally to the motors.
The communication between Centauro and the Petri net executor is of high importance. Centauro always has to react immediately, especially when the operator is controlling Centauro directly. The operator is then sending packets to Centauro with the desired velocity of each wheel. Centauro has to maintain these velocities until the next packet arrives. It is therefore of even greater importance that Centauro is able to treat the packets at the right time so that it doesn't maintain old velocities. The safety is always an important issue when working with robots, especially when working with robots of this strength.
There is an obvious need of a real-time operating system. Centauro has previously run DOS without any real-time support. All the drivers and old applications for Centauro have been developed for a standard DOS operating system.
The solution I found was to write the application in a single thread. This approach has many drawbacks, but I was able to re-use most of the old drivers and control routines. The result is a reliable system that is easy to program.
In chapter 2 I have described how the functionality of the mobile robot could be divided into primitive tasks and actions. This concept requires some kind of mechanism in the mobile robot to offer these actions and primitive tasks to the Petri net executor.
This mechanism is described in the next section. In chapter 2.3 I stated that no parameters were allowed in the primitive task requests. The data should instead be spread in a separate structure. This structure is also described here.

Primitive task server and Data server

Centauro was planned to use the primitive task server and the data server described in [Veiga and Grácio 1998] that was developed in parallel with this project. It was not sure that this server would be ready for use in Centauro when needed, and the decision was then made to develop a smaller combined task and data server in this project. This was to be able to finish on time. The new server had to be compatible in the socket interface. This let me still be able to use their Petri net executor instead of developing one of my own.

Protocol of the Primitive task server

The primitive task server only uses one socket, which receives commands and sends acknowledgements. Every primitive task may have its own socket connections as long as they handle them with care.

There are six different commands, identical to the ones in [Veiga and Grácio 1998].

Name of command	Byte 1	Byte 2	Byte 3	Byte 4	The rest
INIT	MSG ID		0	‘I’	60 bytes of server name
REQ	MSG ID		0	‘R’	2 bytes action id
ACK	MSG ID		0	‘A’	2 bytes action id
RES	MSG ID		0	‘S’	2 bytes of action id and 1 byte of error code
END	MSG ID		0	‘E’	Nothing
USR	MSG ID		0	‘U’	User data

Every packet is filled out to be 64 bytes in total. This will not give any performance problems since the Ethernet packets are never smaller than 64 bytes.

Protocol of the Data server
The design described in [Veiga and Grácio 1998] that was used, used different sockets for the data server and the task server. There are drawbacks to this, the most important being the race condition, since you do not know which packets that will arrived first on the different sockets. The race condition can be handled by implementing semaphores in the Petri net with the primitive actions. This is easily done since every variable is non-dividable. The data server is only partially implemented in this project. The protocol of the data server is further described in [Veiga and Grácio 1998].

Design
The design of the combined primitive task and data server is very simple. It is basically a loop that checks incoming packets and translates them into orders. An example of an order sequence could be:

- Start primitive action 4
- Stop primitive action 8
- Change value of variable 5 to $A7_{Hex}$
- Stop primitive action 4

The use of a single thread in a real-time system requires some kind of co-ordination. This is done by the combined server, since it is the only central server and it has also a natural connection to all the other parts of the system. The call to the active functions is done by round robin.

3.4 Basic functionality

The basic functionality is offered through the primitive task server described in the previous chapter. All the primitive actions that are offered by the primitive task server have to be implemented in a certain way. This is in order to structure their behaviour so that the combined primitive task and data server can handle them all.

Every primitive action should have the following functions:

• Create function	Allocates resources that will be needed during all the execution like opening stream.
• Initialisation function	Initialises every execution of this primitive task.
• Step function	Executes one step of the primitive task.
• Exit function	Cleans up after every execution of the primitive task, and send back the reinforcement learning code. This code should be success, error or failure as described in [Lima and Saridis, 1996].
• Destroy function	Cleans up what is left from the Create function.

Look in Appendix A.5 for a more detailed description. The primitive actions that are described in this chapter are all solving one specific primitive task each. They are only covering a minimum of functionality. It is possible to implement more than one primitive action per defined primitive task. It is also supposed to be more than one primitive action per defined primitive task.

`DirectControl` – Primitive Action

One of the main goals for this project was to make it possible for an operator to control Centauro manually in real-time by simply moving a fancy joystick. This part is implemented as a primitive action offered by the primitive task server. The Space Mouse is further described in Chapter 5.3.
`DirectControl` is supposed to react to the movements that the operator makes on the joystick. The terminal – where the operator is sitting – sends updates every 250:th millisecond how the operator wants Centauro to move.
The messages that are received have the format:

Name of field	Size of field	Type of field	Comments
Speed	32 bits	Float	
Rotation	32 bits	Float	
Time stamp	32 bits	Long int	Not in use
Reserved		Long int	Not in use

The system must deal with the unreliable behaviour of Ethernet and radio connections. `DirectControl` checks constantly that it is receiving the "velocity packets" in a steady pace. It stops both motors as soon as it suspects that a packet is lost or delayed.

`DirectControl` uses the following algorithm:
1. Wait for connection from Centaur Control
2. Receive a message, reset motors if no message is received in 550ms

3. Decode the message and send the velocities to the motors.
4. Take a picture with the camera
5. Extract the track out of that picture
6. Check current speed
7. Send back speed and track position to the terminal
8. Repeat step 2 to 7 until interrupted

`DirectControl` can only exit on the command from the operator. It returns success to Centaur Executor and stops the motors when it receives the command to stop `DirectControl`.

`FollowTrack` – Primitive action

To follow the track closely, Centauro must have knowledge of its own measurements. The `FollowTrack`-algorithm always tries to center the track to the point p in Figure 3.5.

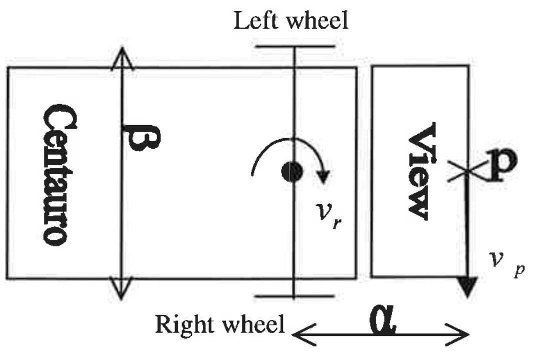


Figure 3.5 A Sketch of Centauro and what it can see through the mirror.

With this information Centauro can move itself in relation to the point p by rotating around its own axis.

v_r = The angular velocity in radians per seconds

v_p = The velocity of the point p caused by the rotation of Centauro, v_r.

$$v_r = \arctan\left(\frac{v_p}{\alpha}\right)$$

The point p is located in the most distant part of the image that is seen through the mirror. Its vertical movement in the Figure 3.5 is only affected by v_r.

Centauro should center the track with a certain velocity but since the track is not always aligned with Centauro, it also has to compensate for the drift imposed by the nonalignment.

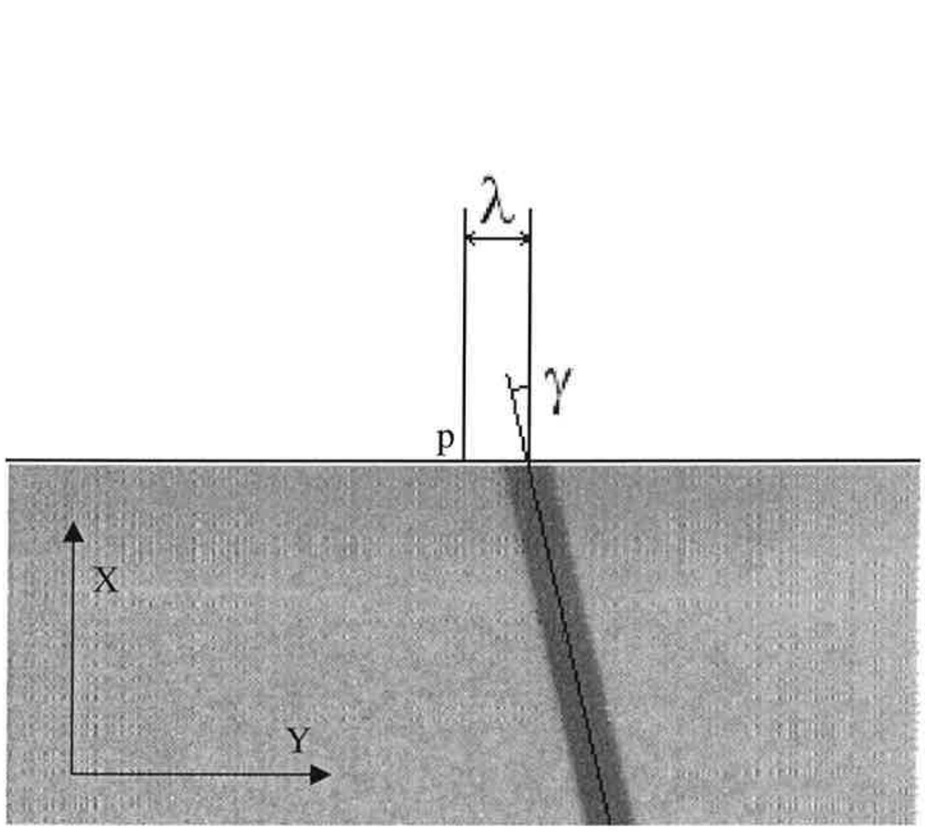


Figure 3.6 The view of the track that Centauro sees, with the displacement and angle of the track.

$\dot{\gamma} = v_r$ and γ makes the track move to the left at a speed of

$v_{dev} = \tan(\gamma)v_c$

where v_c is the velocity ahead (velocity common mode). The controller tries to center the point p at a given rate by

$v_{recenter} = c\lambda$

where c is a constant. We want p to move with the velocity $v_{recenter}$ to center the track subtracted by v_{dev} to compensate for the non-aligned track. This gives

$v_p = v_{recenter} - v_{dev}$

Since Centauro is a differential-drive vehicle

$$v_c = \frac{v_{left} + v_{right}}{2}$$

and

$$\dot{\gamma} = \frac{v_{left} - v_{right}}{\beta}$$

Out of this we get the desired velocities of left and right wheel to be

$$v_{left} = v_c + \frac{\arctan\left(\frac{v_{recenter} - v_{dev}}{\alpha}\right)\beta}{2}$$

$$v_{right} = v_c - \frac{\arctan\left(\frac{v_{recenter} - v_{dev}}{\alpha}\right)\beta}{2}$$

The function `GetTrack` described below obtain λ and γ. `GetTrack` gives the values of θ_0 and θ_1 in the formula $y = \theta_0 + \theta_1 x$, and from this it is easy to get the values of λ and γ.
There are two different possibilities for `FollowTrack` to exit. It returns success when it finds an end of a track, and it returns error when it loses the track because of other reasons.

`GetTrack`

The function `GetTrack` returns the position and direction of the track. It also detects if the track is not present. It only uses the part of the image that is viewed through the mirror. This gives enough information for following the track. It uses every tenth line of the image and gives back position, direction and error of the estimation.
Centauro knows four rules to recognise a track:

1. A track is dark on light background.
2. A track is between 15 and 25 mm wide.
3. A track has big difference between the track and the background.
4. A track has the same background on both sides.

These four rules give the position of the track on all the tested lines. The position of the track on every row analysed are put in a ten by one matrix. This matrix is used to find θ_0 and θ_1 in the equation
$y = \theta_0 + \theta_1 x$
with the least-square-method.
It would be interesting to look at the lower part of the image that shows the more distant view. In this way Centauro would be able to accelerate on a straight track and slow down close to an end or a curve. This should of course be a separate primitive task.

`SearchTrack` – Primitive actions

The set of primitive actions `SearchTrackLeft`, `SearchTrackRight`, `SearchTrackAhead` and `FollowTrack` are all very important for the navigation of Centauro. `SearchTrackLeft`, `SearchTrackRight`, `SearchTrackAhead` and `SearchTrackBack` will be described in this section. They are all very similar and are all supposed to be used at crossroads with 1, 2 or 3 exits like in Figure 3.8.

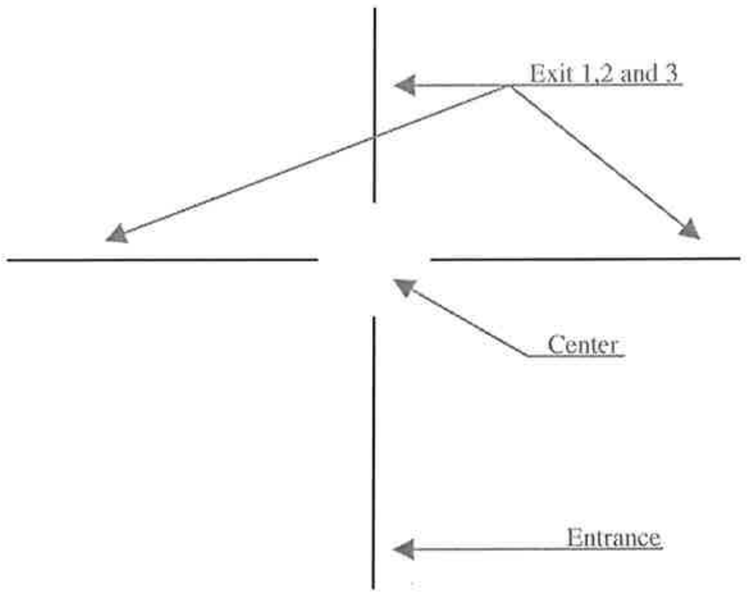


Figure 3.8 A crossroad with one entrance and three exits. The entrance is always defined by where Centauro is entering the crossroad at any specific time.

Centauro is supposed to follow a track into the crossroad. Centauro will of course find the end of the track in the middle of the crossroad. It will then have its camera looking at the white center of the cross road. It can now choose an exit by turning left, right or to go straight ahead.
The `SearchTrack` primitive actions all work the same way. They are looking in the mirror and turn the wheels until they can find a track or they reach a timeout. The velocities that were used in this project can be found in Table 3.1.

Table 3.1 Check the wheel velocities

	Left wheel	Right wheel
`SearchTrackLeft`	–20cm/s	+20cm/s
`SearchTrackRight`	+20cm/s	–20cm/s
`SearchTrackAhead`	+40cm/s	+40cm/s
`SearchTrackBack`	–40cm/s	-40cm/s

These primitive actions return success as soon as they have found any track that is recognised by `GetTrack` (described above) two times in a row. And they report error if they can not find a track before their internal timeout is passed. The way these actions work, we do not know whether the detected track will be centered and/or aligned. This could be solved by making the actions more complex. However, this would violate the ambition to make the actions as primitive as possible. Instead, each successful `SearchTrack` should be followed by either of

- `FollowTrack` directly
- `CenterTrack` and then `FollowTrack`

- `CenterAndAlignTrack`[1] and then `FollowTrack`
- `DirectControl`

In this way we get a more flexible system where the programmer easily can change the smoothness of Centauro.

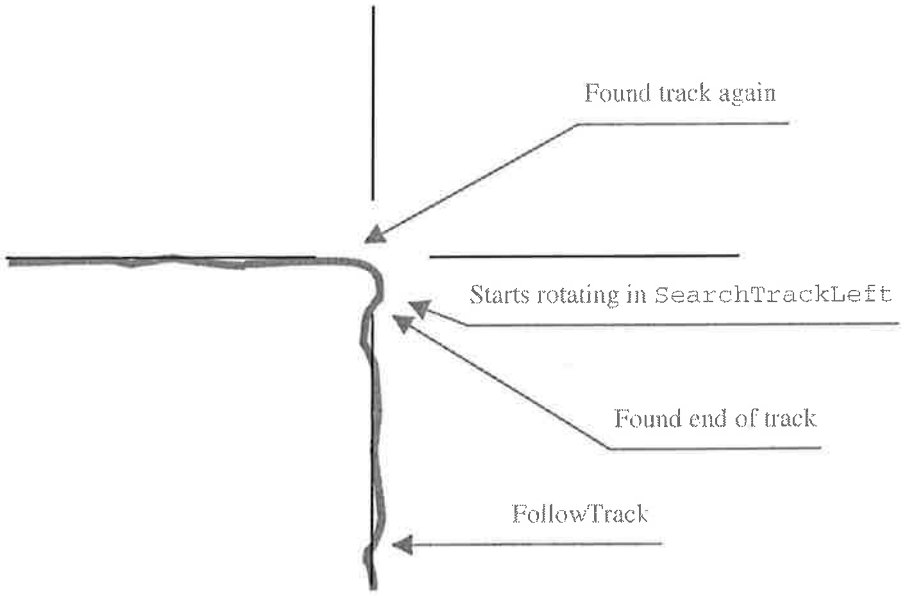


Figure 3.9 Centauro entering a crossroad from below. It chooses to look for the track to the left and then follows it.

`CenterTrack` - Primitive action

Even if `FollowTrack` centers the track while moving forward, it is usually a good idea to center the track in some way before we run the `FollowTrack` primitive action. It may also be useful in other situations.
This primitive action will constantly look in the mirror to check where the track is. It will report an error if it looses the track and a success when the track is within an interval that is defined as the center. It uses the whole track but centers only the part of the track that is in the most distant part of the mirror. This is because we get a better behaviour this way than to center the part of the track that is closer to Centauro.
`CenterTrack` does not move Centauro in the forward direction, but it rotates Centauro around the point just between the wheels. The speed of rotation is proportional to the displacement of the track. This gives a satisfactory result. Care should be take not to use a proportional constant that is to high.
`CenterTrack` turns off the motors and return success when both:

- the track is centered within an internal interval
- the velocity is considered calm enough

[1] This action is not implemented in this project.

This is to prevent Centauro from having any kinetic energy left after `CenterTrack` is finished. `CenterTrack` returns error if it does not manage to center the track within an internal timeout.

`MoveTo_Blend` - Primitive action

`MoveTo_Blend` is just an example of how to implement a primitive action. This primitive action does not do anything it does just go in and out. It does always return success. `MoveTo_Blend` is a good example to build new primitive actions on though and it is available as a primitive action offered by Centauro.

`SendPhoto` – Primitive action

`SendPhoto` was partly developed to ease the development of the other primitive actions that Centauro offers. It was especially helpful in the development of the image recognition routines. The user defined packets in [Veiga and Grácio 1998] were not used, since they were not available at the time.
The protocol is really simple. SendPhoto opens a server socket[2] that will accept one client. It sends the size of the image after the connection is established and then sends the image. It closes the socket as soon as the whole image is transferred.
The modems had some problems with the flow control at high loads. This features forced me to lower the load, and a reasonable load should be five lines in each step[3].
`SendPhoto` does always return success after finished transmission.

`TakePhoto` - Primitive action

`TakePhoto` is another primitive action that is mostly used during development of the different other primitive actions. The main area of future use will be to log pictures at strange occasions, like if non-foreseen errors would appear or if the operator needs a still picture.
`TakePhoto` just takes a photo where the entire image is updated. This photo is taken in one step, which makes it abuse the recommended maximum time to execute one step. The time it takes to take a picture 0.2 seconds. `TakePhoto` does always return success after finished transmission.

`TurnLeft` - Primitive action

I decided to translate this old function into a primitive action mostly to demonstrate the position control of Centauro but also since it is an easy way to turn a bit to the left.
`TurnLeft` uses position control and turns Centauro a certain number of degrees to the left. It does not control that the primitive action is

---

[2] Server socket is a socket that clients can connect to. The client and the server are absolutely equal after the connection is established.

[3] A step is the smallest part of an action that will be run from central scheduler. This is further defined in Appendix A.5.

done but instead it waits a certain amount of time before it returns success.

`TurnRight` - Primitive action

`TurnRight` is a lot like the primitive action `TurnLeft` with the big difference that this primitive action turns to the right.

`GoForward` - Primitive action

`GoForward` is also very similar to the two primitive actions described above. Sets a new desired position on both wheels that is a constant value greater that the current position.

`StopCar` - Primitive action

`StopCar` is also a very primitive action but still a useful one. This primitive action is executed in one step and stops both motors. That is done even if some other primitive action is currently using the motors. This behaviour gives the possibility to paralyse Centauro by sending a slow stream of requests for the `StopCar` primitive action. `StopCar` has no error control and will always return success.

4 Communication

Remote control requires some kind of communication. For a mobile robot it is a good idea to use wireless techniques. But also stationary robots could be controlled with wireless techniques, just to get rid of the wires.
Most of the communication will be short packages that we want to transmit fast with a high reliability.

4.1 Network

All communication is done over the Ethernet network at the department. This network has been extended with a radio link to Centauro. Ethernet is not a good protocol for communication in real-time systems, since there is no guarantee on delivery time or that the data is delivered at all. These drawbacks are however minimised there is no other equipment connected to the Ethernet.
The radio link is built up by a radio modem that transmits the signal to an identical radio-modem that is connected to Centauro. The modem model is WCL3670 from AAEON. They have the capacity of 2 Mbit/second but no flow-control. The software TCP/DOS from IBM is also added to Centauro to supply it with the TCP/IP protocols in DOS.

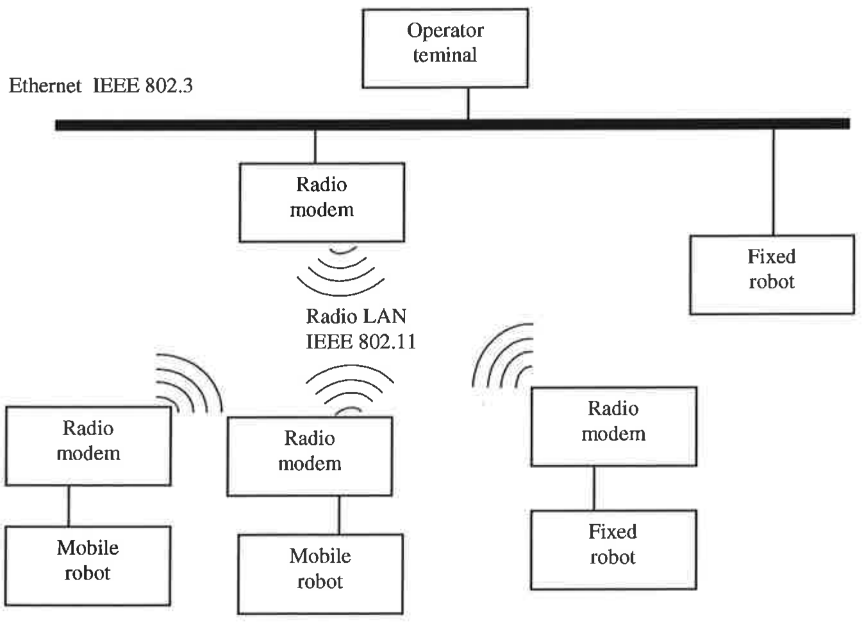


Figure 4.1 Draft of how the communication architecture could be.

4.2 Lost packets and delays

The TCP/IP protocol suite contains the necessary communication functionality for Centauro. Centauro needs a secure way to send error free data. The streams that TCP/IP offers should assure that the data

that is sent will arrive. However, they do not guarantee when the data will be delivered.
The CSMA/CD and CMSA/CA protocols both imply delays varying with time. This gives a hazardous situation if the processes running in Centauro require information at certain times. Special care has therefore been taken to avoid serious consequences with delayed packages. `DirectControl` in Chapter 3.4 does stop immediately when the packages are not arriving on time and all primitive action that uses the motors do reset them after use.

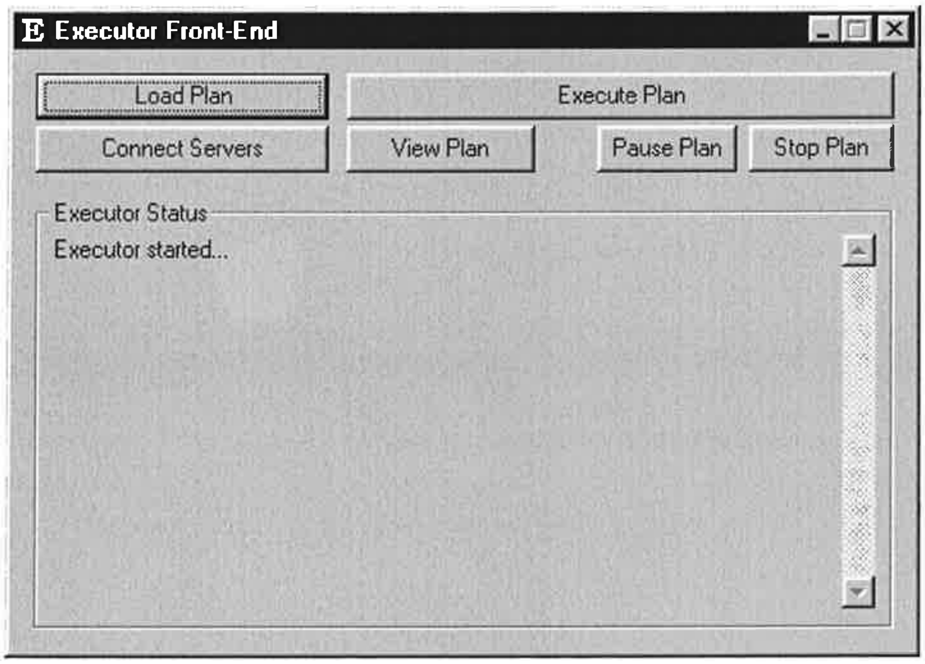
5 Operator console

As described earlier there is a large need of easier ways to remote control robots remotely. An important part of this is of course the how the operator interacts with the robot via the terminal. As mentioned in Chapter 1 the operator can control Centauro in two different modes:

- Centauro executing a Petri net
- Centauro receiving explicit movement commands

The Petri net runs in the program Centaur Executor. Manual control is done with the program Centaur Control. Both programs are described in this chapter.

5.1 Centaur Executor



Executor started...

Figure 5.1 The status window from the Petri net executor.

Centaur Executor is the Petri net executor. The operator may follow the execution of the Petri nets, but should also be able to control the execution of the Petri nets. It should be possible to:

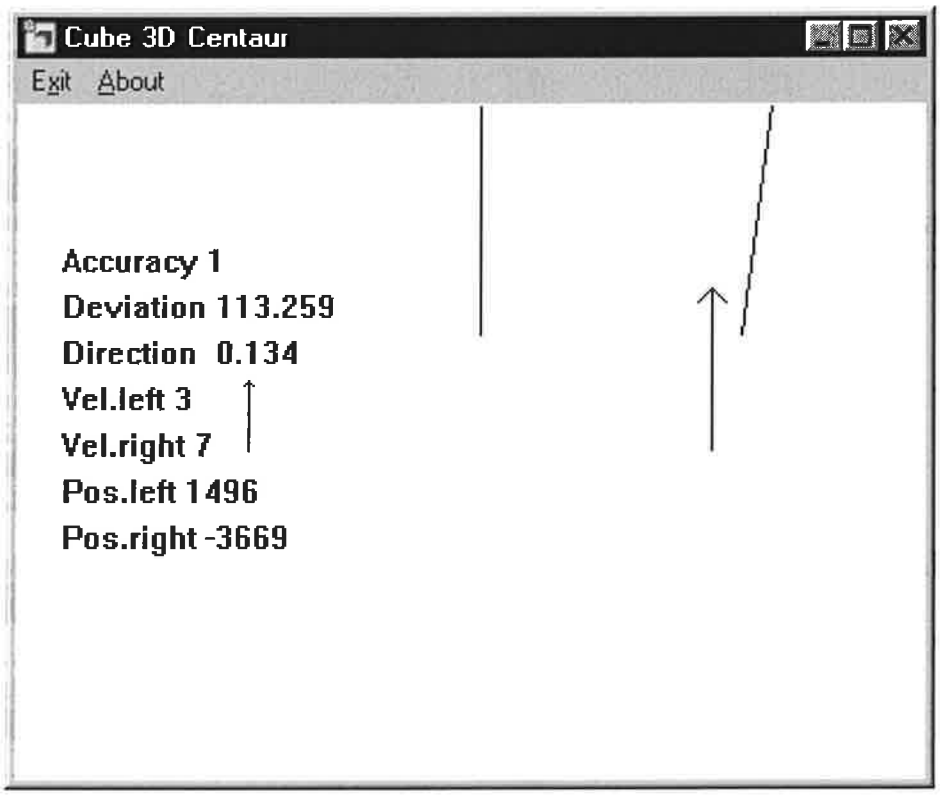
- Add tokens at places.
- Remove tokens from places.
- Stop the system at an emergency.

The operator can control the execution of the Petri net by moving tokens between the different places. This lets the operator change the execution very directly and for example start `directcontrol` by moving the token to that place. He can then later put Centauro back in to its normal program after he is finished with the interference. Centaur

Executor is further described in [Veiga and Grácio 1998] under the name Petri net Executor.

5.2 Centaur Control

Centaur Control is the program that the operator should use to control Centauro directly. It is started from Centaur Executor as soon as Centaur Executor orders `directcontrol` from Centauro. The operator can simply push, drag and rotate the Space Mouse and Centauro will follow. Basically, this is a control loop where the operator is the main control algorithm.
When the operator pushes the space mouse away from the operator, Centauro receives a command to go ahead. The camera takes new pictures that are decoded into the position and direction of the track (if it is visible). This information is sent back to the operator console, where the operator sees what happens and may adjust his commands via the Space Mouse. Figure 5.2 shows the main window in Centaur Control.The features are presented as the text to the left. Accuracy is how much doubt Centauro has about the track. The higher value the more doubt. Deviation is how many pixels that the track has deviated from the ideal position. Direction is in what direction the track is pointing. The vertical line is the ideal position and the sloping line is the actual position and direction of the track. Vel.left is the velocity that Centauro has on the left wheel, which is also represented with the small arrow to the left. Vel.right is the velocity of the right wheel that also is represented as an arrow. Pos.left and pos.right are the positions that these wheels report right now.



Accuracy 1
Deviation 113.259
Direction 0.134
Vel.left 3
Vel.right 7
Pos.left 1496
Pos.right -3669

Figure 5.2 The main windows from Centaur Control.

Centaur Control follows this algorithm:

1. Initiate a timer that sends an empty message every quarter of a second.
2. Initiate the Space Mouse to send all updates to Centaur Control.
3. Connect to `DirectControl` primitive task in Centauro.
4. Receive a message from Space Mouse or the timer.
5. Create a message with current time and desired speed of left wheel and right wheel from last update from the Space Mouse.
6. Send the message to `directcontrol` in Centauro.
7. Repeat 4,5 and 6 until interrupted
8. Close connection to `directcontrol` in Centauro.
9. Close Space Mouse

5.3 Space Mouse

Space Mouse is developed by SPACE CONTROL in Germany. It is a six axis joystick which give you the possibility of full 3D control of objects – translation and rotation.

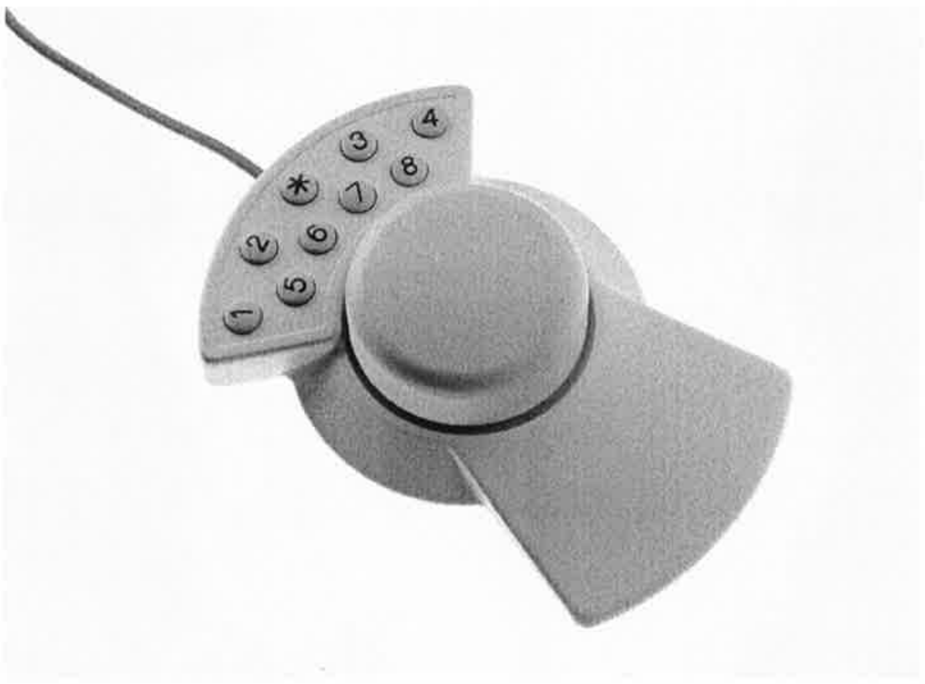


Figure 5.3 Space Mouse from Space Control in Germany.

Input from the Space Mouse

The vehicle has far less degrees of freedom than the Space Mouse. Centauro is moving around in the X-Z-plane but can only move in one direction at the time. It has to rotate to move in another direction. At every specific time Centauro has full freedom in two dimensions, rotation around the Y-axis and movement along the Centauro-axis - its own axis. The Centauro-axis is defined to be in the same direction as the camera of Centauro is looking. The Centauro-axis is always inside the X-Z-plane.

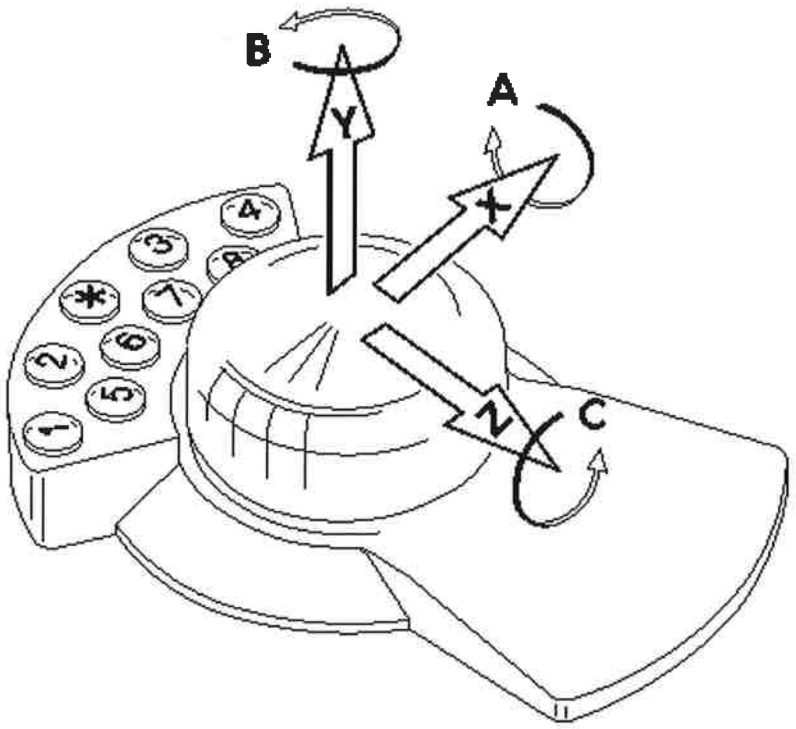


Figure 5.4 Six degrees of freedom

With the full freedom in these two dimensions we are able to move wherever we want to go in the X-Z-plane and also to rotate in any direction around the Y-axis. We have found a new degree of freedom, but we do not have full freedom in all the three degrees of freedom. This is because the rotation expands the Centauro-axis to the X-Z-plane. Look in Figure 5.5 for an example.

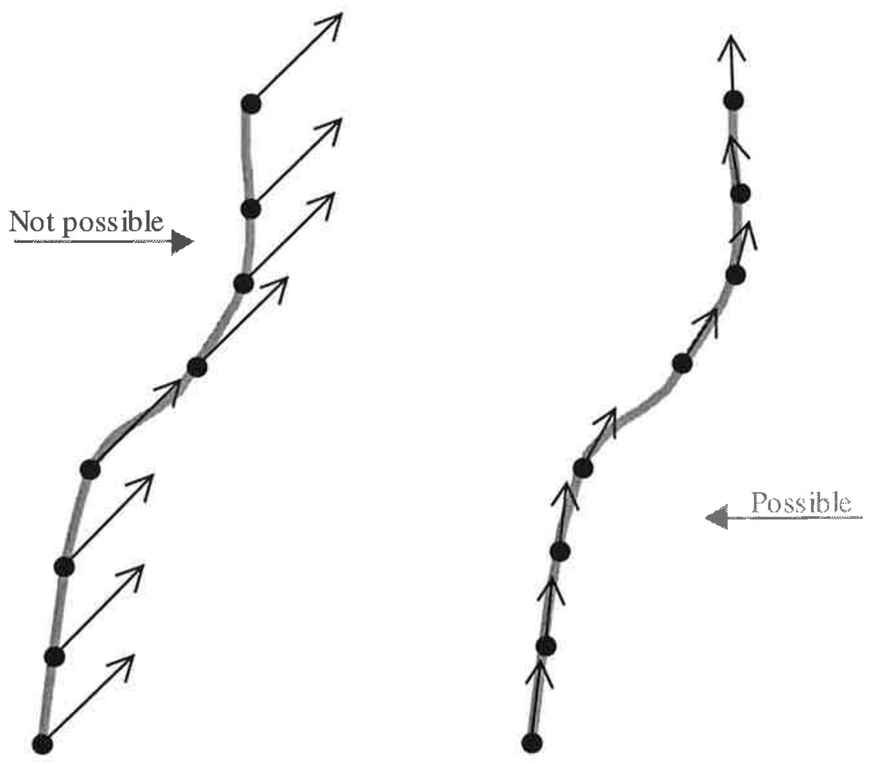


Figure 5.5 Examples of what trajectories that are possible and not possible. The grey line is the track, the black ball is Centauro and the black arrow is the direction of Centauro.

We have some different ways to transform the movements of the Space Mouse to desired movements of Centauro. The way I chose was to use the rotation around the Y-axis and the movements along the Z-axis of the Space Mouse. Rotation of the Space Mouse makes Centauro rotate and movements along the Z-axis makes Centauro move along the Centauro-axis. The more you twist and turn the Space mouse the more does Centauro twist and turn. Using this transformation of the signal from the Space Mouse to the vehicle doesn't cancel any of the degree of freedom of the vehicle.

Commands at keyboard of Space Mouse

Space Mouse has a small keyboard as Figure 5.3 shows. I have only chosen to use three of these keys. The keyboard of Space Mouse has the following functions

1. Resets the motors of Centauro
2. Not in use
3. Not in use
4. Turns on and off the feature display
5. Not in use
6. Not in use
7. Not in use
8. Exits Centaur Control

6 Case study

In LRM[4] there is a track made up by electrical tape on the light coloured floor. This track is sketched in Figure 6.1. I have made up a small mission for Centauro to test the system that I have built up. The mission is to follow the track from point A to point C. At C the operator is supposed to control Centauro to perform some kind of docking and then to give back the control to Centauro after the operator is done. Centauro should then return to the original position following the track back.

This is a quite small mission but we must divide it into smaller parts that match the primitive tasks that Centauro offer. This is quite easily done after a look at the track.

We assume that Centauro is placed between A and B at the start. It would be a good idea to first center the track under Centauro and then follow the track to the turn at B. At B Centauro should search for the track to the right, center the track and then follow the track until the end.

At D Centauro should just let the operator perform direct control until the operator is finished. The instructions to the operator includes that he should leave Centauro looking at the track in the direction towards C. Centauro should now center the track and follow it back to A.

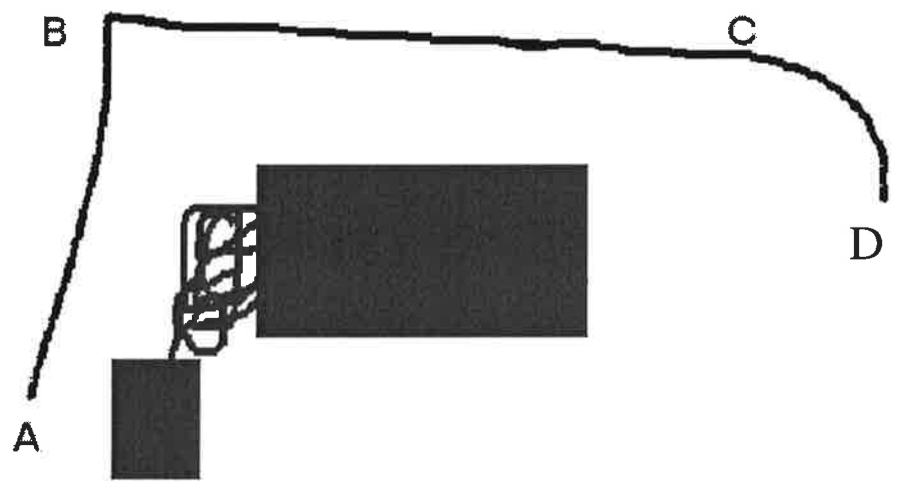


Figure 6.1 Sketch of the track that was used for testing Centauro.

[4] LRM (Laboratorio de Robotica Movel) - Research lab at ISR at IST - Lisbon.

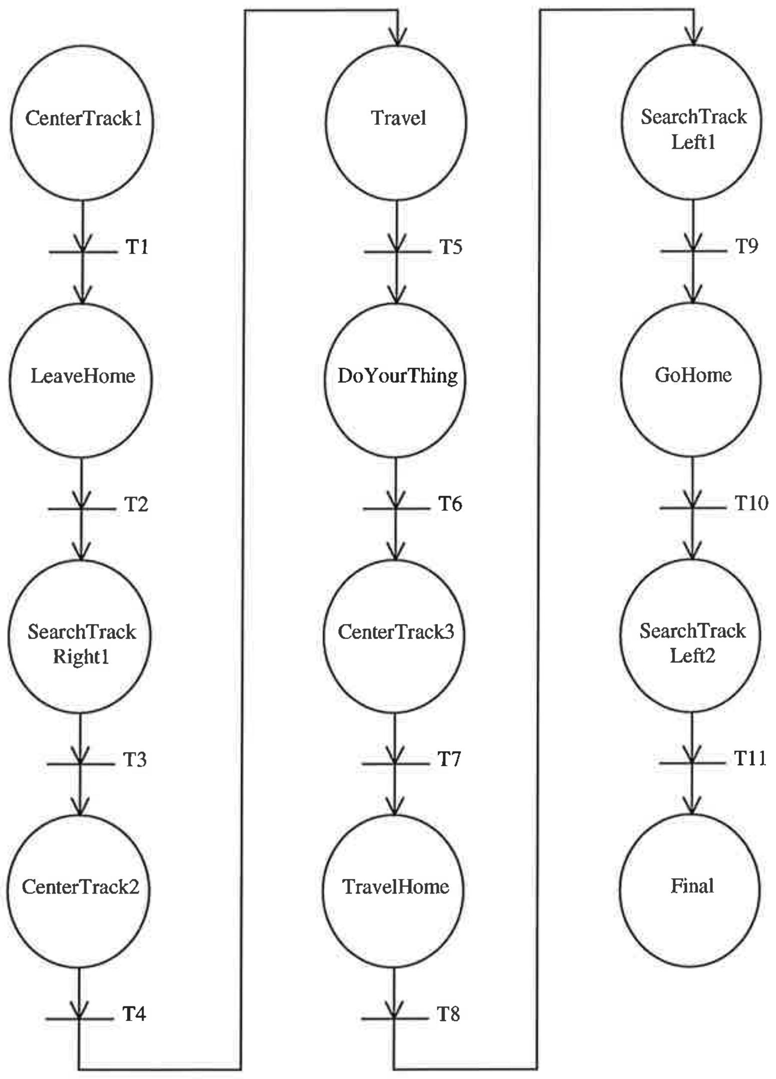


Figure 6.2 The Petri net used for control of Centauro

This mission tests most of Centauros capabilities. It shows that the set of primitive tasks that Centauro offers makes it possible to do simple operations. It tests if the primitive tasks are defined in a good way, that is, if my ideas that I presented in chapter 2.2 of how the mission should be divided into primitive tasks are good. It also tests if the `FollowTrack` primitive action is able to follow bended tracks, which it does. The listings of this mission are available in appendix D.

7 Conclusions, future work

The goals that were defined for the project were all met. This project managed to construct a system on Centauro that offers primitive actions. These primitive actions are later translated to primitive tasks by the external petri net executor. The architecture was taken from [Lima and Saridis, 1996].
The case study described in Chapter 6 was defined and done successfully proving the usefulness of the control system that is built up in Centauro.

7.1 Future improvements

This project shows that it is possible to build up a robot with primitive actions and primitive tasks. The goal has not been to build a commercial mobile robot. I have been forced to just implement a minimum, enough to test all the tricky parts. There are many new primitive tasks to be defined for Centauro and yet more primitive actions to implement. There is no redundancy among the primitive actions implemented in this project. Every primitive task has exactly one primitive action implemented. Of course Centauro should offer many primitive actions for every primitive task that is defined. An interesting project would then be to extend Centaur Executor to handle reinforcement learning and to implement a set of redundant primitive actions for every primitive task that is defined for Centauro.
Besides this Centauro offers an interesting platform to define more primitive tasks and to implement more primitive actions. A short list of handy primitive tasks could be:

- Align with track
- Dock
- Advise new safe velocity
- Build map

The most interesting of these primitive tasks would in my opinion be “Advise new safe velocity”. It would be a very neat way of demonstrating both how to build up parallelism, and also how the data should flow between the different primitive actions. The primitive task “Advise new velocity” would then run in parallel with for example `FollowTrack`. “Advise new velocity” would constantly calculate the maximum advisable velocity from lower part of the images taken with the camera. `FollowTrack` would read that velocity from a shared variable.
There is also some more tedious work that should be done to secure future use of Centauro. The wiring on Centauro is not as reliable as it should be. The power supply that transforms the battery output to the computer can sometimes give trouble, see Appendix B. There might also be other problems that are not yet identified. All this makes it advisable that the next project on Centauro should start with a general check of the wiring.

Further I would like to see Centauro co-operate with other robots. A parallel project to this one was to build up a set of primitive actions for the Puma manipulator[5]. It is of course hard not to see what possibilities this would give by set Centauro and the Puma to co-operate in the same Petri net.
Figure 7.1 show an example of how the co-operation between the Puma and Centauro could be. Here the Puma is picking parts from a central storage, giving them to Centauro. Centauro delivers the parts to the three different workstation guided by the tracks

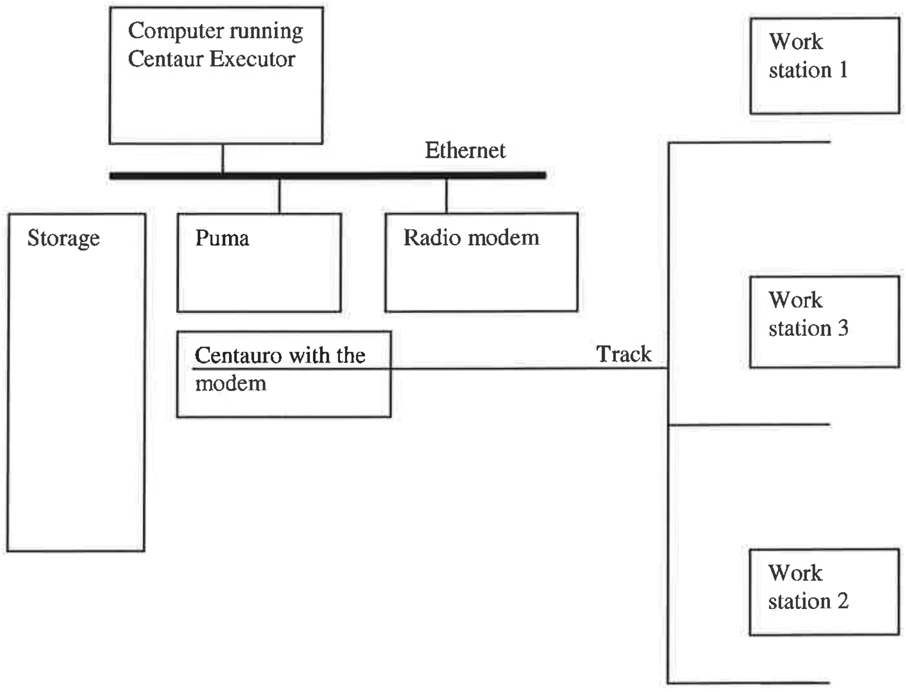


Figure 7.1 An example of how the Puma and Centauro could co-operate delivering parts from a central storage to three different work stations.

5 This puma manipulator is positioned in the Intelligent Control Lab at IST.

8 References

DAVID, R. and H. ALLA (1992): *Petri Nets and Grafcet: Tools for modelling discrete event systems*. Prentice-Hall International (UK) Ltd.

LIMA, P. and G. SARIDIS (1996): *Design of Intelligent Control Systems Based on Hierarchical Stochastic Automata*. World Scientific Publ. Co.

GOLOMBEK, M. P. (1998): "THE MARS PATHFINDER MISSION AND SCIENCE RESULT." *29th Lunar and Planetary Science Conference.*

LIMA P, P. SILVA, J. SANTOS and C. CARDEIRA (1997): "Centaur: A Vision-Based Autonomous Mobile Robot." *Proc. of European Advanced Robotics Systems Development (EUREL),* IEE Press.

GRÁCIO H and V. VEIGA (1998): "Coordenacao e Monitorizacao de Tarefas para um Ambiente Distribuido Utilizando Redes de Petri", *Final Year Project Report - Computer Science Program, Instituto Superior Tecnico, Lisbon Technical University*

FEI-YUE WANG and G. SARIDIS. (1993) "Task Translation and Integration Specification in Intelligent Machines", *IEEE Trans. on Robotics and Automation, RA-9 (3): 257-271.*

A How to use the source code

Here are some short descriptions on how to use the different source files that control Centauro.

A.1 LM

LM.C contains routines to send and get information from the 4QD-controllers. It has a mixture of different procedures using different interfaces but they are all on a very basic level. Add the procedures you need in the future.

A.2 CAMERA.C

CAMERA.C contains all procedure that relates to the camera and feature extracting of the camera. Might be divided in two parts when more feature extraction procedures are implemented.

A.3 CONFIG.C

CONFIG.C Reads configuration files.

A.4 CONSTS.C

Contains constants about Centauro and the world.

A.5 PRIMITIV.C

Contains all the primitive actions. The primitive actions have to be implemented in a certain way to work with the data and primitive task server. The header of each function should be like

```
error MoveTo_Blend(int * currentstate)
```

Where the current state can be six different values.

0. Do nothing
1. Initiate the primitive action at program start. Should change the current state to 0 after this initialisation.
2. Initiate the primitive action to be executed once. Should change the current state to 3 after this initialisation.
3. Takes one step of the primitive action. Should do a small part of the work. Should only change the state when done. When done with the work the current state should be changed to 4.
4. Should finish the execution of the primitive action and return the error code in error. The current state should be changed to 0.
5. Should finish the primitive action at program termination. Set the current state to 0.

The error codes returned when the primitive action finishes should be as defined in PROTOCOL.H that is imported from Petri net executor.
If the primitive action is not finishing - changing current state from 3 to four and returning a valid success-code according to PROTOCOL.H – then it should return 99.

```
/*
 *                          ACTION_MOVETO_Blend
 */
/* Action Task that implements the Move To function */
/* of the PUMA */
error MoveTo_Blend(int * currentstate)
{
   switch (*currentstate)
   {
     case 0:
     break;
     case 1:
     //create
       PrintStr("MoveTo_Blend CREATED\n");
       *currentstate=0;
     break;
     case 2:
     //init
       PrintStr("MoveTo_Blend INITIATED\n");
       *currentstate=3;
     break;
     case 3:
     //step
       PrintStr("MoveTo_Blend STEPPED\n");
       if (...done with the action...)
          *currentstate=4;
     break;
     case 4:
     //exit
       PrintStr("MoveTo_Blend EXITED\n");
       *currentstate=0;
       return __LEARN_RESULT__SUCCESS__;
     //break;
     case 5:
     //destroy
       PrintStr("MoveTo_Blend DESTROYED\n");
       *currentstate=0;
     break;
   }
   return 99;
};
```

B Appendix – Start-up of Centauro

Make sure that you have the following:

- Three fully loaded 12V batteries
- One Centauro robot with at 3670 radio-modem
- One more 3670 radio-modem
- One console-computer running Windows 95 or something similar
- One space-mouse

Press the emergency button, turn of the computer switch (on the side of Centauro chassis). Connect all the batteries. Turn on the computer with the switch on the side of Centauro. Check that the yellow diode on the modem flashes rapidly. One red diode is shining constantly and that the other one seems to flash randomly (weakly).
When the boot-sequence is over execute the batch-file C.BAT to start Centauro. You should get a lot of information on the screen ending with opening primitive task and data-server. Take a deep breath. Be prepared to repress the emergency-button while you release it. If Centauro moves repress the button immediately.

Characteristics	What is wrong	How to correct it
When turning on the computer nothing happens.	The battery power doesn't reach the computer	Check that the battery for the computer is connected to the power-supply, that the power-supply is connected to the motherboard, hard disk, disk drive and CPU-fan.
When turning on the computer almost nothing happens.	Probably a short-circuit or a bad connection in the battery power-supply.	Touch the cables that go to the most inner part of the power-supply and wish that the connections will work better.
When starting C.BAT nothing happens.	Probably something wrong with the LM-board.	Turn off the computer recheck the cables and move the LM-board to the right position.
When C.BAT is started it is still not possible to connect to Centauro from another computer.	Probably something wrong with the modems.	Check that the light from the modems look right. If it doesn't reboot them until it does. The server-modem should always be turned on before the client-modems.
Centauro doesn't reply to PING.	Same as above.	Same as above.

C Appendix – How to configure the modems

To configure the modems can be quite hard since there are a lot of different alternatives and the manual doesn't say anything about which that are possible, but it is easy to do with the following instructions.
To configure the modems you need

- One modem of model 3670 or 3671
- One white cable. Ordinary twisted-pair cable
- One power-supply for the modem
- One computer with an Ethernet-card
- The diskette marked "WaveCell WCL 3670 Security and Diagnostic Version 5.2"

Please note that you should finish this procedure within one minute, and also that the security-code should be the same for the modems that want to talk to each other, that is also for the channel. There should be exactly one server-modem. The procedure is as follows:

1. Connect modem with the white twisted-pair-cable to the Ethernet-card
2. Connect the power-supply to the modem
3. Start the program "A:\Aaeon_wg.exe"
4. Chose Control – Config
5. Press GetID. If you get TimeOut just try again.
6. Mark the ID you want to change
7. Change the values
8. Press either SetClient or SetServer depending on what you want to do
9. You should get the result "Expected result" from the program. If you do not get a successive result just disconnect the power from the modem for a short while and then continue from step 5.

When you have configures the two modems put the server at a hub or equal with the blue twisted-pair-cable and a power-supply. Remember to always turn on the server before you turn on the client. Put the client on Centauro and connect it with a white cable and a keyboard-power-cable.
The easiest way to test it now is by using PING. Test it in both directions!

D Source code of the case study

D.1 SERVERS.CFG

This file defines all servers that may offer primitive actions in this setup. I have here defined a server called "Centaur" with the IP name damiao.isr.ist.utl.pt. This is because the system is run in the isr.ist.utl.pt network. I do also define that all the information that the Petri net executor knows about Centauro is read from the local file "centaur.srv".

```
:server Centaur Centaur.srv damiao
```

D.2 EXECUTOR.CFG

This file defines variables that are placed in the Petri net executor.

```
:data NbPlans int
:data NbTimesExecuted int
:data StatusExecutionCounter int array 3
:data FinishExecution int
:data SUCCESS int
:data ERROR int
:data RESULT int
```

D.3 CENTAUR.SRV

Here I have declared all the ports, variables and primitive actions from Centauro. I do also define which primitive actions that solve which primitive tasks.

```
:server_name Centaur
:data_port 1600
:primitive task_port 1700
:data VarTeste int
:data RT int
:primitive_primitive task MoveTo_Blend
:primitive_action MoveTo_Blend 0
:primitive_primitive task MoveTo_Blend
:primitive_action MoveTo_Blend 1
:primitive_primitive task SendPhoto
:primitive_action SendPhoto 2
:primitive_primitive task TakePhoto
:primitive_action TakePhoto 3
:primitive_primitive task FollowTrack
:primitive_action FollowTrack 4
:primitive_primitive task TurnLeft
:primitive_action TurnLeft 5
:primitive_primitive task TurnRight
:primitive_action TurnRight 6
:primitive_primitive task GoForward
:primitive_action GoForward 7
```

```
:primitive_primitive task StopCar
:primitive_action StopCar 8
:primitive_primitive task DirectComm
:primitive_action DirectComm 9
:primitive_primitive task SearchTrackLeft
:primitive_action SearchTrackLeft 10
:primitive_primitive task SearchTrackRight
:primitive_action SearchTrackRight 11
:primitive_primitive task CenterTrack
:primitive_action CenterTrack 12
:primitive_primitive task RunByKB
:primitive_action RunByKB 13
:primitive_primitive task SearchTrackAhead
:primitive_action SearchTrackAhead 14
:primitive_primitive task SearchTrackBack
:primitive_action SearchTrackBack 15
:primitive_primitive task Dock
:primitive_action Dock 16
:primitive_primitive task ReverseTrack
:primitive_action ReverseTrack 17
```

D.4 CENTAURWHOLE.VPP

This file is the actual program that should be executed.

```
:signature WholeTrack
:place CenterTrack1 CenterTrack
:place LeaveHome FollowTrack
:place SearchTrackRight1 SearchTrackRight
:place CenterTrack2 CenterTrack
:place Travel1 FollowTrack
:place Travel2 FollowTrack
:place DoYourThing DirectComm
:place CenterTrack3 CenterTrack
:place TravelHome FollowTrack
:place SearchTrackLeft1 SearchTrackLeft
:place GoHome FollowTrack
:place SearchTrackLeft2 SearchTrackLeft
:place Final null
:transition T1 null
:input CenterTrack1
:output LeaveHome
:transition T2 null
:input LeaveHome
:output SearchTrackRight1
:transition T3 null
:input SearchTrackRight1
:output CenterTrack2
:transition T4 null
:input CenterTrack2
:output Travel1
:transition T5 null
```

```
:input Travel1
:output Travel2
:transition T5a null
:input Travel2
:output DoYourThing
:transition T6 null
:input DoYourThing
:output CenterTrack3
:transition T7 null
:input CenterTrack3
:output TravelHome
:transition T8 null
:input TravelHome
:output SearchTrackLeft1
:transition T9 null
:input SearchTrackLeft1
:output GoHome
:transition T10 null
:input GoHome
:output SearchTrackLeft2
:transition T11 null
:input SearchTrackLeft2
:output Final
:marking InitMark CenterTrack1
:endplace Final
```