TDDD07 - Robolab Rescue Lab
Group A4

Michel Jérémy
micje093@student.liu.se

Diallo Boubacar Sidy
boudi616@student.liu.se

12 December 2016

Contents

1 Introduction **2**
1.1 Tasks . . . 2
1.2 Goals . . . 2

2 Lab 1 **2**
2.1 Conditions of measurement and analysis . . . 3
2.2 Schedule design . . . 4
2.3 Schedule evaluation . . . 4

3 Lab 2 **5**
3.1 Starting the communication at the right time slot . . . 5
3.2 Priority definition . . . 6
3.3 Admission control . . . 6

4 Optimisations **8**
4.1 Optimisations on admission control . . . 8
4.2 Optimisations on offline scheduler . . . 9

5 Conclusions **10**

6 Appendix **11**
6.1 Appendix 1: measurements . . . 11
6.2 Appendix 2: scheduler.c . . . 12
6.3 Appendix 3: task_communicate.c . . . 13
6.4 Appendix 4: Optimized task_communicate.c . . . 14

1 Introduction

This report is going to cover our experimentation and analysis on the Robolab rescue lab series. This lab series emulates the context of a team of rescue robots, along with some of the challenges they face. For this, we are provided with a robot for each team. The robots can navigate thanks to a set of tasks in a closed area containing tags on the floor that it can read.

1.1 Tasks

The very first and most important task is the **Mission task**. It is responsible for controlling every other tasks, starting or stopping them. It also generates data to be sent with the Communicate task. Here are the underlying other tasks:

- **Navigate**: controls the robot ability to go toward a new tag
- **Refine**: refines the position of the robot using the tags
- **Report**: sends victim location to the communication task
- **Control**: takes care of the low level communication
- **Avoid**: avoids obstacles
- **Communicate**: sends data on the wireless channel

1.2 Goals

Our goals for lab 1 will be to design an offline schedule and to evaluate its performance. Lab 2 is going to go a bit more in-depth, and we will have to make the communication task execute at the right time slot. We will also make sure that the amount of data sent never exceeds the time slot (admission control).

2 Lab 1

Lab 1's goal is to design an offline schedule and to evaluate its performance. The first thing to do is measuring the length of each task in different conditions. Knowing how much time a task needs to execute is crucial in building an offline schedule, because once the scheduler is started there is no way to impact it in any way, nor to stop a task's execution.

2.1 Conditions of measurement and analysis

We decided to make the measurements in two different conditions: in a corner and in the middle of the area. Elevating the robot in the air was discussed, but we decided not to do it because that would not be a case of normal robot utilisation, unless of course the robot somehow falls from a higher place (but it is not designed for it, nor is it possible in the enclosed lab area). The measures were taken using the time library given in the source code.

The results are in **Appendix 1: measurements**. The first 10 measures are taken in the middle, while the last 10 are taken in a corner.

When analysing the measurements, the first step was calculating the worst case execution time. This was done by looking at the results and doing some statistics.

Some tasks (Refine, Communicate) show a very low standard deviation when compared to the order of magnitude of the values. Others however deviate a lot more, like mission going from 0.00293 to 0.159912, which is a difference of 54 times. However, some values appear to be "flukes", and maybe should not be relied upon.

Still looking at the Mission task, the median is at 0.003906, with the Q3 at 0.003967, which is very far from 0.159912. This value appears only once, and could be an accident. The reason these "abnormal values" should not be taken into account when defining WCET is that there would be a waste of time where no task would be executed most of the time, aside from these rare cases. Finding a balance between avoiding waste of time and not going over the execution time of another task was the main difficulty of this exercise.

We went on this reasoning for every task until obtaining the following WCETs, in milliseconds:

TASK	Navigate	Avoid	Report	Mission	Control	Refine	Com.
WCET	1	17	0.005	0.005	5	11	1.5

Figure 1: WCET of the different tasks

We now had to define the periods. We decided these using the data given in the compendium and our own reasoning:

- **Navigate** is a core task, allowing the robot to move from one tag to another. It should be ran as often as possible.
- **Refine** is more or less the same as report, it is important but not needed at every minor cycle.
- **Report** is only useful when on a victim tag. It may not be ran as often as possible like navigate.

- **Mission** controls the underlying tasks, so it should be ran at every period.
- **Control** should be ran at every period, as it controls the low level communication
- **Avoid** must be ran no quicker than every 100 or 150 milliseconds, to avoid overheating the sensors.
- **Communicate** must be executed once (and only once) a second. The specifics for communicate will be seen in lab 2, but for now the time at which it is executed does not matter.

This resulted in the following table, showing both final WCET and periods:

	NAVIGATE	AVOID	REPORT	MISSION	CONTROL	REFINE	COMMUNICATE
WCET (ms)	1	17	0.005	0.005	5	11	1.5
Period (ms)	100	200	200	100	100	500	1000

Figure 2: WCET and periods of the different tasks

2.2 Schedule design

Now, we are going to show how we built the schedule. We started by figuring out what our major period should be. The communicate task, with a period of 1000ms and the longest task, fit easily in our schedule and became the major period.

The minor period then had to be divisible by the major period. We had trouble deciding between 50ms and 100ms, but after trying out both we found out that a 50ms minor schedule sometimes introduced some cycles longer than intended and thus skipping a cycle (once every 20 major cycles or so). This could be because of an inaccuracy of our measurements, or a lack a sample.

This lead us to choose a minor schedule of 100ms, which never had any issue. Building the schedule was simply a matter of using modulo operator and a for loop increasing by the value of the minor cycle at each iteration. The scheduler.c file featuring our scheduler is available in **Appendix 2: scheduler.c**. Note that the part of the code calling usleep concerns Lab 2, which will be covered soon.

The code is fairly easy to understand: at the beginning, we set the major cycle and minor cycle to their values. Next, we start the scheduler and execute the right tasks at the right moments using modulo operator. At the beginning of each loop, we synchronise with the right timeslot (we had robot 8, so 8^{th} timeslot). Finally, after each for loop iteration, we wait for the end of the minor cycle and go on to the next iteration.

2.3 Schedule evaluation

Our first iteration featured a 50ms minor cycle, along with a 1s major cycle. Our measurements showed that it should have been able to work. However, the schedule was overran

about a tenth of the time. This was measured using the time library available to us, by looking at the time difference between the beginning of the major and the end of it (incrementing by what was needed for the end of the minor cycle each time). The culprits mainly seemed to be both Refine and Avoid, taking too much time.

Using the 100ms minor cycle, we had no schedule overruns from our observations. When stopping the robot, it was stopping in less than 1 second, which is within the allotted time in the application requirements.

We had some issues with the actual localisation of the robot on the Mission Control Software. Sometimes, the robot was jumping from one place to another, and then resynchronising with the current, correct position. The victims, however, were found at the right place, and the robot was navigating from one victim to another relatively seamlessly. The issue may simply be caused by the fact that the software is local to the computer and has some trouble synchronizing with the robot in the test area.

3 Lab 2

Lab 2's goals are to define a priority order and to implement the new task_communicate.c using these priorities. We should also make sure the communicate task is executed at the right time, which is for us at the very beginning of time slot 8, because we were using robot 8. Finally, we should perform admission control to make sure the 125ms time slot is never exceeded.

3.1 Starting the communication at the right time slot

The very first difficulty was finding a good way to start the communicate task at the right moment. What we did was, at the beginning of each loop, execute two instructions that synchronized our scheduler with the next time slot. Basically, after the end of a major period it is looking for the next 875ms time slot.

Additionally, we needed to modify the schedule a bit and put communicate as the very first instruction, so that it was executed at the right moment.

The major problem with this implementation is that it is going to execute communicate at the very beginning of the time slot. Our experimentation showed that the go ahead signal was not always being executed at the beginning, but a few milliseconds later (less than 10). That means we are potentially losing some information at the beginning of the time slot when calling communicate task a bit too soon.

A better implementation would have involved waiting for the go_ahead to pass to 1 each time. However, we failed to find a good way to to this in time.

Anyway, now we could see the communicate task being executed at the right moment in the mission control software. The new problem was that the data sent on the channel were overlapping with the next time slot (too long), and not necessarily in the right order. This is what we are going to talk about in the next part.

3.2 Priority definition

We needed to define a priority list for the 5 message types:

- **Victim report**: priority 1, because it is the main goal of this Robolab rescue application: finding victims and reporting it.
- **Location**: priority 2, because knowing there is a victim is potentially more important than locating it. It is also important for the next go_ahead. It could be priority 1, depending on the line of reasoning.
- **Pheromone map**: priority 3. It is less important than location or victim reports, but it is important for other robots not to visit an already visited place.
- **Stream data**: priority 4. The stream data should only be sent if there is room for it.
- **Cmd data**: priority 5. For future use?

3.3 Admission control

With the message order decided, we implemented it using C code in the task_communicate.c file. An example of our process to get each message type to its right place is available in **Appendix 3: task_communicate.c**.

With the message order fixed, we needed to take care of one last thing: the amount of data sent, so that it never exceeds the time slot duration.

Our solution was to limit the amount of pheromone map to 8, which is an arbitrary value that seemed to work well. We guessed that more messages would increase pheromone map accuracy, but we needed some room for stream data too.

After multiple tries and optimising the amount of data that could fit in the timeslot, here is what our Mission Software control showed at this point:

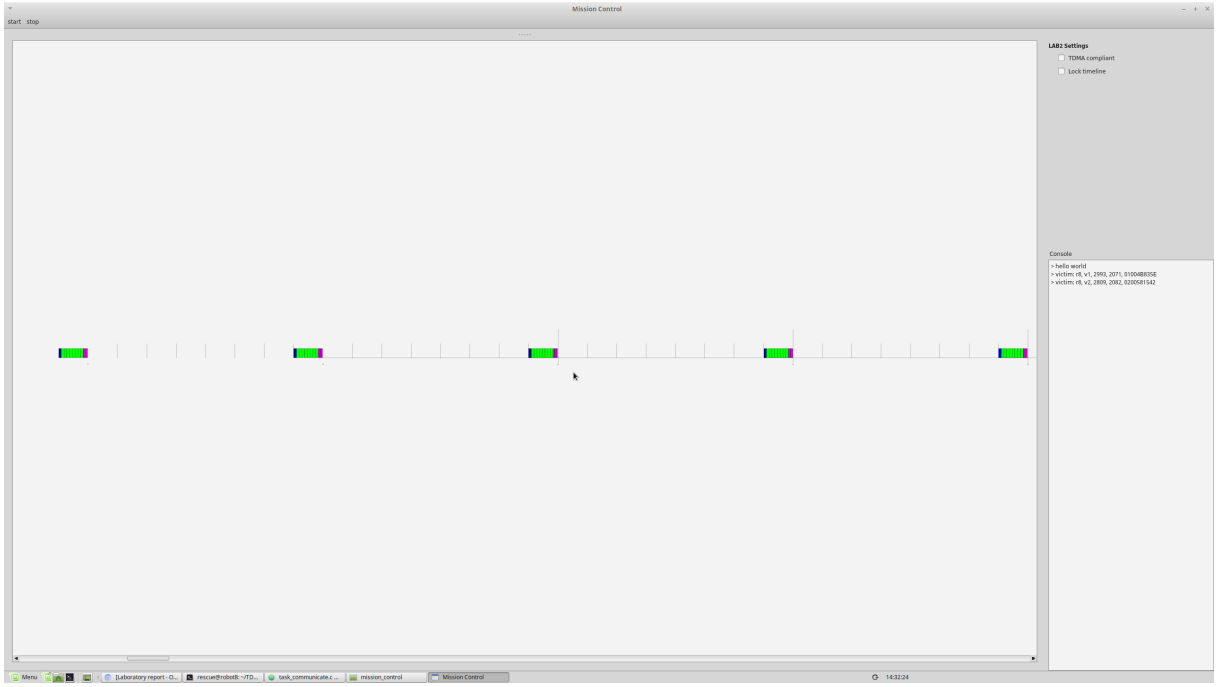

Figure 3: Timeslot compliance with multiple cycles

As seen in figure 3 above, no data go over the next timeslot, and it starts at the right moment. Here is a zoomed version on one typical timeslot:

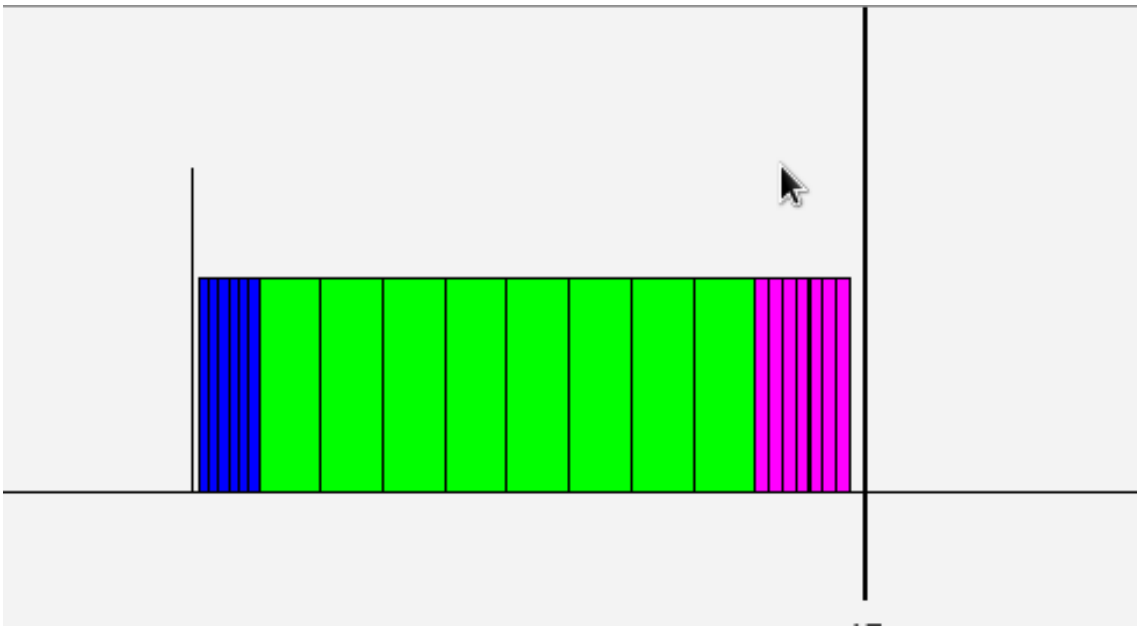

Figure 4: Timeslot compliance one one cycle

The blue data is the location data, sent at the beginning. Green data is pheromone map and stream data. Red data are victim reports, but there are no victim report in this figure. They are at the very beginning when present.

However, our solution consisting in limiting the maximum amount of each message type is not perfect because we are wasting bandwidth in some cases, which is going to be highlighted in the next section, Optimisations.

4 Optimisations

4.1 Optimisations on admission control

Our solution currently takes into account that the robot can find at most one victim. This is what the time slot looks like with one victim:

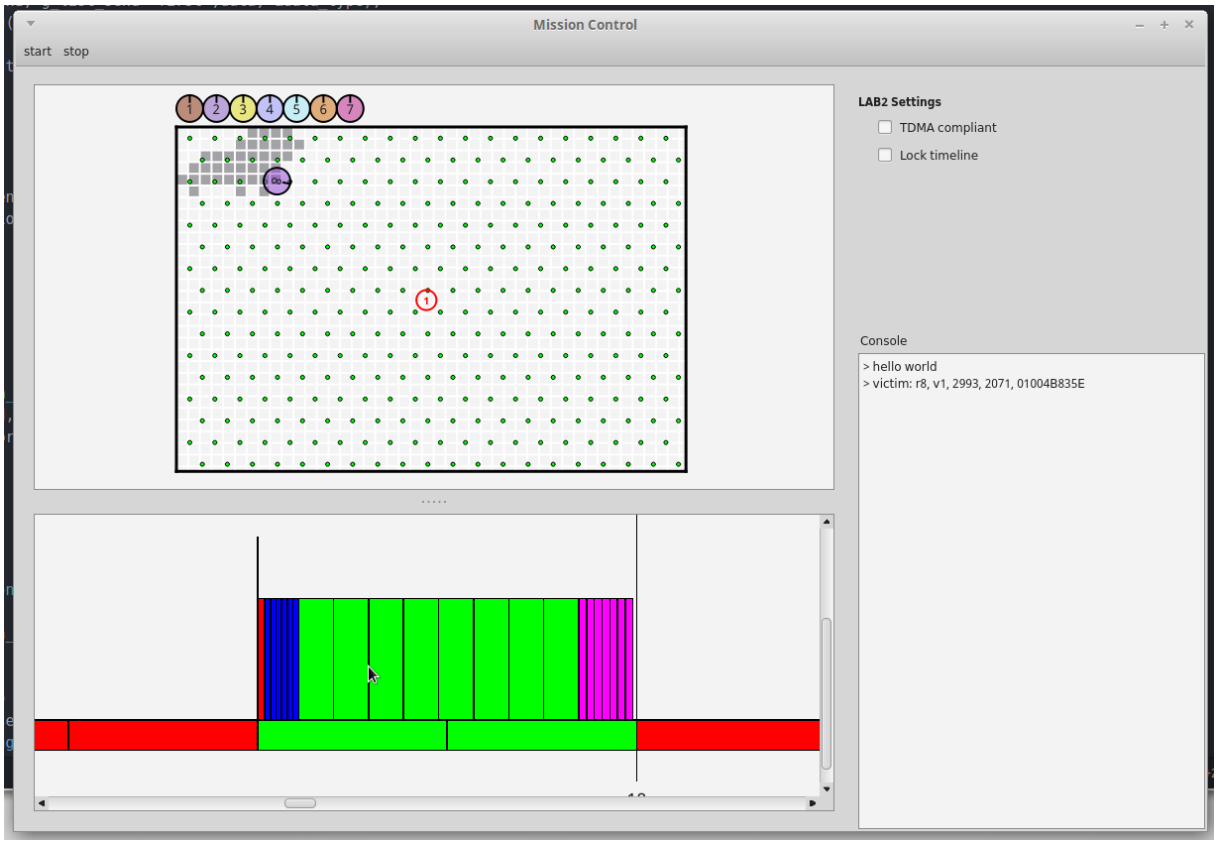

Figure 5: Bandwidth usage with one victim

After the demonstration, we noticed that we could optimise our task_communicate a bit, to increase the bandwidth usage. In the case there is no victim, there is going to be room for one more stream data. Likewise, if there is more than one victim found in a cycle, we are going to overlap with the next timeslot. A better implementation would feature modifying the code a bit, and limit the amount of stream data to 8 - victim_count. 8 stream data is what the timeslot can fit if there is no victim count. Based on experiments, a stream data message is the same size (or very close) than a victim report message.
Such optimization is available in **Appendix 4: Optimized task_communicate.c**.

4.2 Optimisations on offline scheduler

In our implementation of the scheduler, we are not always efficient. Let's imagine that the program starts at a time modulo 1000 equal to 950ms. That means it is going to sleep until the next timeslot, which is 875ms (we did the demonstration on robot 8). The program is thus going to wait 925ms doing nothing. A better implementation would take into account if the timeslot has passed yet or not at the first iteration. If it has not yet passed, keep

with the current instruction (wait until 875ms). However, if it has passed, we could wait until the end of the current second and then start with the program. This would make the program idle for 50ms (in the case of 950ms) instead of 925ms. However, we failed to find out how to do it.

5 Conclusions

At the end of the lab sessions, we successfully reached all of the goals. The offline schedule works and the robot is able to find and report the victims. The communicate task is called at the right time slot, allowing data to be sent at the right moment. The mission control software shows good results and time slots are never exceeded. We have some issues however, most notably with the synchronization of the current robot location, which we believe to be a bug due to the application being hosted client-side, and not on the robot.

Our solution for admission control is not very elegant either, because we are hard-coding the amount of stream data and pheromone map messages to be sent. This is fine when there is at least the same amount of messages of each type as the limit we set, but if there is for example less pheromone map we are essentially wasting bandwidth, where we could potentially send one more stream data message. If we had more lab this issue could be fixed.

This project allowed us to understand how scheduling could work in practice, even though we are aware that this is only a toy example. While we do have a somewhat working solution, there is definitely a lot of room for improvement.

6 Appendix

6.1 Appendix 1: measurements

MEASUREMENTS

		NAVIGATE	AVOID	REPORT	MISSION	CONTROL	REFINE	COMMUNICATE
Middle	1	0.815918	16.361084		0.006104	4.052979	10.414062	1.335205
	2	0.645996	12.174072	0.003906	0.003174	2.566895	10.424072	1.246826
	3	0.721191	13.13916	0.003906	0.159912	3.173828	10.583984	1.231934
	4	0.427979	15.807129	0.005859	0.003906	1.776855	10.434814	1.338135
	5	0.624023	13.855957	0.27002	0.003906	4.569092	10.36499	1.250977
	6	0.709961	9.61792	0.005859	0.003906	4.177979	10.425049	1.375
	7	0.810059	13.918945	0.00708	0.00293	3.544189	10.562988	1.292969
	8	0.591797	10.591064	0.00293	0.00415	3.47583	10.583008	1.396973
	9	0.61499	14.491943	0.003906	0.003906	2.49707	10.289062	1.562012
	10	0.734131	14.189941	0.00415	0.003906	2.220947	10.631104	1.369873
Corner	11	0.814941	12.583008	0.00293	0.00415	3.595947	10.666016	1.274902
	12	0.738037	11.826904	0.003906	0.003906	3.313965	10.476807	1.428955
	13	0.483154	14.11084	0.003906	0.00293	5.462891	10.688965	1.5271
	14	0.873047	15.947021	0.004883	0.003906	5.075195	10.716064	1.302979
	15	1.10791	13.745117	0.00293	0.003174	4.11792	10.48291	1.25
	16	0.785156	13.231934	0.00293	0.003906	3.566895	10.553955	1.26001
	17	0.446045	13.282227	0.00415	0.003906	3.156006	10.562988	1.420166
	18	0.781982	12.004883	0.003906	0.003174	3.452881	10.571045	1.417969
	19	0.817871	16.358887	0.003906	0.00293	3.288086	10.510986	1.270996
	20	1.236816	10.145996	0.00415	0.00415	3.366943	10.343018	1.488037
	Average	0.7390502	13.3692016	0.0181691	0.0115966	3.52261965	10.51429435	1.3520509
	Median	0.736084	13.513672	0.003906	0.003906	3.4643555	10.5324705	1.33667
	Q3	0.8151853	14.2654415	0.0045165	0.003967	4.06921425	10.583252	1.41851825
	Deviation	0.1919545	1.89984074	0.0593713	0.03403266	0.87853028	0.114940291	0.0960415695

Figure 6: Measurements of the execution times

6.2 Appendix 2: scheduler.c

```
/**
 * Run scheduler
 * @param ces Pointer to scheduler structure
 * @return Void
 */
void scheduler_run(scheduler_t *ces)
{
  static int MAJOR_CYCLE = 1000;
  ces->minor = 100;

  /* --- Write your code here --- */
  scheduler_start(ces);

  while(1) { // loops around
    double now = floor(((long long)timelib_unix_timestamp() % 1000));
    usleep(abs((875 - now) * 1000));
    // synchronize with major cycle period (1 s)
    for (int i = 0; i < MAJOR_CYCLE; i += 100) {
      if (i == 0) {
        scheduler_exec_task(ces, s_TASK_COMMUNICATE_ID);
        scheduler_exec_task(ces, s_TASK_NAVIGATE_ID);
        scheduler_exec_task(ces, s_TASK_CONTROL_ID);
        scheduler_exec_task(ces, s_TASK_AVOID_ID);
        scheduler_exec_task(ces, s_TASK_MISSION_ID);
        scheduler_exec_task(ces, s_TASK_REFINE_ID);
        scheduler_exec_task(ces, s_TASK_REPORT_ID);
      } else if (i % 500 == 0) { // timer = 500, 1000
        scheduler_exec_task(ces, s_TASK_MISSION_ID);
        scheduler_exec_task(ces, s_TASK_NAVIGATE_ID);
        scheduler_exec_task(ces, s_TASK_AVOID_ID);
        scheduler_exec_task(ces, s_TASK_CONTROL_ID);
        scheduler_exec_task(ces, s_TASK_REFINE_ID);
      } else if (i % 200 == 0) { // timer = 200, 400, 600, 800
        scheduler_exec_task(ces, s_TASK_MISSION_ID);
        scheduler_exec_task(ces, s_TASK_NAVIGATE_ID);
        scheduler_exec_task(ces, s_TASK_AVOID_ID);
        scheduler_exec_task(ces, s_TASK_REPORT_ID);
      } else if (i % 100 == 0) { // timer = 100, 300, 700, 900
        scheduler_exec_task(ces, s_TASK_MISSION_ID);
        scheduler_exec_task(ces, s_TASK_NAVIGATE_ID);
        scheduler_exec_task(ces, s_TASK_CONTROL_ID);
      }
      scheduler_wait_for_timer(ces);
    }
  }
}
```

src/scheduler.c

6.3 Appendix 3: task_communicate.c

```
switch(g_list_send->first->data_type) {
 case s_DATA_STRUCT_TYPE_ROBOT:
  // priority 2
  data = (void *)malloc(sizeof(robot_t));
  position = victim_count;
  position_act = 0;
  // insert after last victim count
  if (position == 0) {
    // if 0, insert at beginning (no location msg yet)
    doublylinkedlist_remove(g_list_send, g_list_send->first ,data, &data_type)
    ;
    doublylinkedlist_insert_beginning(prioritized_list, data, data_type); //
    exception
  } else {
    node_act = prioritized_list->first;
    position_act++;
    while (position_act < position) {
      node_act = node_act->next;
      position_act++;
    }
    doublylinkedlist_remove(g_list_send, g_list_send->first, data, &data_type)
    ;
    doublylinkedlist_insert_after(prioritized_list, node_act, data, data_type)
    ;
  }
  location_count += 1;
  free(data);
  break;
}
```

src/task_communicate.c

6.4 Appendix 4: Optimized task_communicate.c

```
case s_DATA_STRUCT_TYPE_STREAM:
 // priority 4
 data = (void *)malloc(sizeof(stream_t));
 if (stream_count < (8 - victim_count)) {
   position = victim_count + location_count + pheromone_count;
   position_act = 0;
   if (position == 0) {
     doublylinkedlist_remove(g_list_send, g_list_send->first ,data, &data_type
    );
     doublylinkedlist_insert_beginning(prioritized_list, data, data_type);
   } else {
     // go to the right position
    node_act = prioritized_list->first;
    position_act++;
    while (position_act < position) {
      node_act = node_act->next;
      position_act++;
    }
    doublylinkedlist_remove(g_list_send, g_list_send->first, data, &data_type)
    ;
    doublylinkedlist_insert_after(prioritized_list, node_act, data, data_type)
    ;
  }
  stream_count += 1;
 } else {
  doublylinkedlist_remove(g_list_send, g_list_send->first, data, &data_type);
 }
free(data);
break;
```

src/task_communicate_opt.c