Masaryk University
Faculty of Informatics



Distributed-Battery Energy-Balancing Simulator

Bachelor's Thesis

Kristián Kumor

Brno, Spring 2021

Masaryk University
Faculty of Informatics



Distributed-Battery Energy-Balancing Simulator

Bachelor's Thesis

Kristián Kumor

Brno, Spring 2021

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Kristián Kumor

Advisor: Prof. RNDr. Jiří Barnat, Ph.D.

Acknowledgements

I would like to give my special thanks to Jan Mrázek for his support with the application and testing, which helped a lot. I also want to thank Jiří Barnat, my advisor, for the feedback and for directing me throughout the work. Furthermore, another person that deserves my thanks is Andrej Džubek for his explanations of electrical engineering.

Abstract

In modular robotics, robots are created using many small modules, with each possessing its own battery. In order to increase the operational time of the robot when the power consumption of the modules is asymmetrical, power-sharing strategies were introduced. This thesis aims to create a simulator where it is possible to test the performance of custom strategies on custom topologies, which represent the interconnected modules in a modular robot. This is achieved using the Qt framework, a tool for the creation of graphical interfaces, dynamically linked shared libraries to insert strategies as functions at run-time, and the JSON format used to store the results of the simulation. After presenting the created application, it contains a case study done on three implemented strategies, measuring their efficiency in four different topologies.

Keywords

Qt, GUI, CLI, simulation, JSON, RoFI, self-reconfigurable, robots, modular

Contents

Introduction **1**

1 Preliminaries **3**
1.1 Power sharing . . . 3
1.1.1 Self-configurable modular robots . . . 3
1.1.2 Strategies for dynamic power sharing . . . 3
1.1.3 Batteries . . . 4
1.1.4 Current division . . . 4
1.2 Qt . . . 6
1.2.1 Qt Framework . . . 6
1.2.2 Qt Widgets . . . 6
1.2.3 Qt Core . . . 7
1.2.4 Graphics View Framework . . . 9
1.2.5 JSON Support in Qt . . . 10
1.2.6 Multi-platform file support . . . 10
1.3 Shared libraries in C/C++ . . . 11
1.3.1 Shared libraries . . . 11
1.3.2 Dynamic linking . . . 12

2 BatSim application **13**
2.1 Installation . . . 13
2.2 Usage . . . 13
2.3 Graphical User Interface . . . 15
2.4 User-made strategy . . . 17
2.5 Simulation details . . . 20

3 Case study **23**
3.1 Setup . . . 23
3.1.1 Strategies . . . 23
3.1.2 Topologies . . . 24
3.2 Results . . . 26
3.3 Evaluation . . . 30

4 Conclusion **33**

Bibliography **35**

A An appendix **37**

List of Tables

List of Figures

1.1 *A graph showing the changing speed of charging a Lithium-ion battery. The figure is acquired from an article written by Isidor Buchmann, CEO of Cadex Electronics Inc.*[2]. 5

1.2 *Image shows how the widgets are placed and look in the compiled application. Widget classes are in blue. Notice the layout of the widgets. The figure is acquired from Qt Documentation*[7]. 7

1.3 *This is an abstract example of the connection mechanism. Connect function takes an object providing the signal and the signal itself, then the other object and its slot. The figure is acquired from Qt Documentation*[8]. 8

2.1 *A screenshot showing the application with a basic topology and Move tool selected. Nodes are green rectangles. The yellow rectangle is a selected node, with its settings window opened, containing work and wait operations in its time plan.* 14

2.2 *A screenshot of visualized results. The purple node is the sender. Yellow lines with arrows show the direction of the energy flow. Bottom slider shows which turn is displayed.* 16

2.3 *An example strategy code. The export macro makes the strategy compatible on both Windows and Linux, but it does not have to be present on Linux at all. The rest of the function definition is essential for shared libraries to function. This example uses a static node interface pointer variable to unset the receiver the next time the strategy is called.* 19

3.1 *Here is a circle topology with 17 nodes, where the power can flow using two ways.* 24

3.2 *A snake topology is often used as a centipede robot. There are 18 nodes in total. The power always flows through one path.* 25

3.3 *A star topology can form a spider, for example. Every arm has five nodes, 26 nodes in total. Here we test only the decentralized configuration.* 25

3.4 *This is a hexagonal spider web topology with 25 nodes. The power can flow through a lot of different paths.* 25

Introduction

Power balancing can be very vital for distributed battery systems. These are systems with several battery units which can initiate energy transfer between themselves. In practice, this can be implemented in self-configurable modular robots. Every robot has a computing unit capable of processing information upon which a decision will be based, whether to send or receive energy. Modular robots are small modules that can be connected together to create larger robots. One example of such robots is the RoFI platform, developed at the Faculty of Informatics of Masaryk University[1]. In the case of unequal energy consumption in the modules, sharing power can significantly extend the whole robot's operational time, based on the consumption inequality.

Every power-sharing strategy programmed into a module can have different results. Therefore we want to speed up the process using a simulation, which works with a custom topology of a distributed battery network and applies a user-made strategy to control the power balancing in the simulated modules. The primary focus are self-configurable modular robot networks, but it is possible to simulate other distributed battery networks using the simulator as well.

The first chapter describes the details of modular robots and power balancing, including some basic strategies already developed and some of the technicalities of lithium-ion and lithium-polymer batteries, which dominate the power storage in robotics. Mainly their charging speed[2].

The following section is an overview of the Qt framework[3] used in the application. It is based on the C++ language and is commonly used to create a graphical user interface (GUI). However, this is not the only use of the framework, as it offers other tools for cross-platform development, networking, databases, and many others. The essential features we use are:

- JSON processing.
- Dynamic library support.
- Tools for widget-based GUI.

- Signals and slots, used for asynchronous programming of the buttons and other user interaction with the program.

In the end, we describe dynamic linking and shared libraries in C/C++.

The second chapter shows the BatSim simulator itself, focusing on the controls and explaining the GUI. It also includes a short example of a power-sharing strategy for modules. Then it explains the background of the turn-based simulation and its structure, located in the simulation library.

The last chapter is a case study, where we compare three implemented strategies for power-sharing and their effectiveness in four different topology types with centralized or decentralized clusters of active nodes and dynamic or static power consumption.

The thesis is about simulating power balancing in distributed battery systems, such as self-reconfigurable modular robots, using a user-defined topology and strategy while also showing how much the strategy effectiveness can differ based on the topology structure. The simulation can be used to find new, more efficient power-sharing strategies in modular robotics.

1 Preliminaries

1.1 Power sharing

1.1.1 Self-configurable modular robots

Modular robots consist of smaller modules that can be put together to perform tasks the individual modules could not. This means they can be reassembled to do different tasks, unlike the classic robots built only for specific tasks, which are, however, usually more efficient at what they do. This limits modular robots to research, rescue missions, or colonization since industrial usage usually requires more productivity than flexibility.

In addition, self-reconfigurable modular robots can be connected to each other without direct user intervention, therefore screwing the modules together, meaning they have a docking mechanism. This mechanism allows them to connect to another module anytime they are programmed to do it. In order to perform a complex task, the robots need to communicate with each other and synchronize the movement, either centrally in a core module that controls the others or individually using the information from the rest of the robot.

1.1.2 Strategies for dynamic power sharing

Since all of the modules have their own battery but their power usage differs, some modules will deplete their energy sooner than others, potentially making the whole robot unusable. We call this the maximum operational time. It can be significantly prolonged in the case of a few parts with very high energy consumption and others with minimal or none, using dynamic power-sharing between the individual components. Of course, in a robot with all parts having the same power consumption, it would not have any effect. This method is based on setting the receiving and sending modules. The others let the energy pass through themselves.

There are several basic power-sharing strategies. Highest to lowest strategy chooses the module with the most battery capacity and starts transferring to the least full battery. Another strategy, proposed by Dr. Raja Humza Qadir[4], consists of sharing energy to the direct

neighbor if their energy is lower than a specified threshold and the power of the sender itself is larger than the threshold. A more complex, near-optimal strategy, based on the calculation of theoretic operation time of every node neighborhood and then choosing the roles for every individual module, was proposed by Chi-An Chen from the University of Southern California [5].

1.1.3 Batteries

With the robot modularity, one can create many different topologies, and strategies can differ in efficiency based on the topology. In order to thoroughly test a strategy, it would require much time till all of the batteries deplete in the real world. This is where a simulated environment would be a reasonable choice. A simulation can test much faster and on a greater scale, with minimal investment, since we are not required to have any robots at our disposal. Since simulations are only simplifications of reality, it is vital to focus only on the essential aspects of power-sharing mechanisms.

Batteries used in modular robots nowadays are either lithium-ion or lithium-polymer. Notable is their efficiency, percentage of energy not lost when charging and discharging, ranging between 80 to 95 percent. Another factor is the charging speed, which is influenced by the charged capacity percentage, with higher amounts slowing the charging speed. Before reaching the 70-80 percentage, the charging speed is constant. After that, the speed gradually slows down to zero.

1.1.4 Current division

Another concept related to power flow through the network is the current divider and Kirchhoff's circuit laws. Kirchhoff's current law states that in a node, the sum of incoming currents is equal to the sum of outcoming currents. The current division dictates how the power gets split between the parallel wires. In order to simplify the concept, power will flow through every valid path from point A to point B. The exception being edges with opposing currents in both directions. In such a case, the currents are either equal, and both are nullified with the power flowing elsewhere, or one is stronger and will remain, unlike the weaker one.

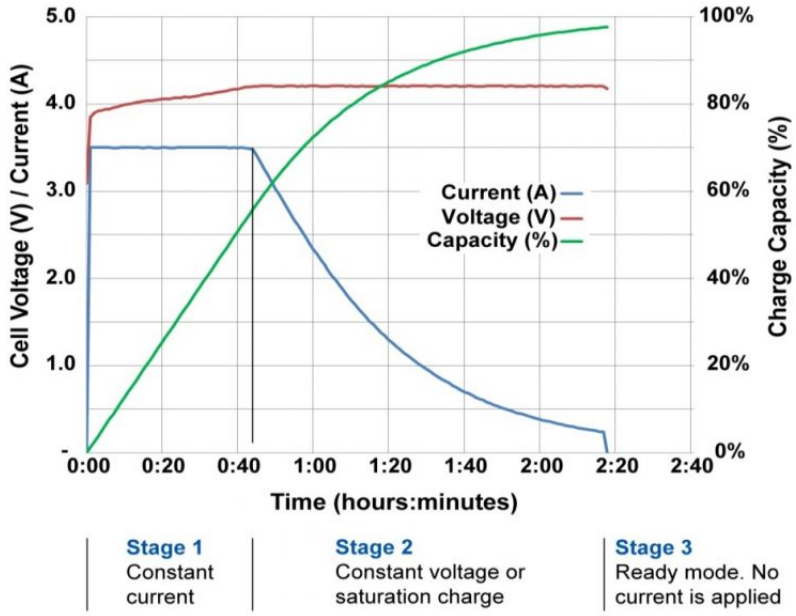


Figure 1.1: A graph showing the changing speed of charging a Lithium-ion battery. The figure is acquired from an article written by Isidor Buchmann, CEO of Cadex Electronics Inc.[2].

The general formula for the current I_X in a resistor I_X, with total current entering the network being I_T, is:

$$I_X = \frac{R_T}{R_X + R_T} I_T \tag{1.1}$$

Where R_T is a combination of other n resistors total resistance:

$$\frac{1}{R_T} = \frac{1}{R_1} + \frac{1}{R_2} + \ldots + \frac{1}{R_n} \tag{1.2}$$

Optionally, we can use the alternative formula for I_X, where R_T is the total resistance of all resistors, including R_X:

$$I_X = \frac{R_T}{R_X} I_T \tag{1.3}$$

1.2 Qt

1.2.1 Qt Framework

Most applications need some graphical user interface(GUI) rather than a command-line (CLI), which is not enough in some cases. The most known tool for creating a GUI in the C++ language is the Qt framework[3]. It is a viable tool for both new and experienced programmers, despite its extensive array of features, which are not limited to interfaces but cover many cross-platform development aspects. These include, for example, support for JavaScript Object Notation (JSON) files, SQL integration, networking, 3D rendering, displaying SVG images, high-level multi-threading, unit testing Qt applications, and more. All these features are separated into modules.

The following sections in this chapter are focused on modules that we used. These include mainly Qt GUI, Qt Core, and Qt Widgets. We will also mention Qts Integrated Development Environment (IDE), the Qt Creator[6]. Outside of the usual debugging, refactoring, and autofill tools, it provides forms to speed up GUI development significantly.

The Qt framework comes with both the commercial and free license. However, it is legally forbidden to mix a commercial license with a free license in one project or product. As is customary, the free license is an open-source one. Therefore, commercial distribution is not allowed.

1.2.2 Qt Widgets

The most vital part of GUI creation is the Qt Widgets[7]. It is mainly intended for traditional desktop environments. Besides creating windows, it also offers rich text processing, hardware-accelerated graphics, data visualization using Qt Charts, or embedding web content into the application through Chromium-based WebEngine. We will mainly focus on the widgets themselves.

Widgets are the core parts used in GUIs that can display data and receive user input. They can also be placed inside one another, with the parent not embedded in any other widget, called window. The programmer can also create his own custom widget classes using the existing ones.

Because we do not want to place every single widget in the window on precise pixel coordinates, we place the widgets into a layout. Layouts automatically arrange child widgets in the space they have assigned. This also supports window resizing, so the widgets adjust accordingly to every monitor resolution and manual windowed mode size. Widget settings can be customized to set a minimum or maximum size in both directions so that the layout will not make the widgets too small or too big.

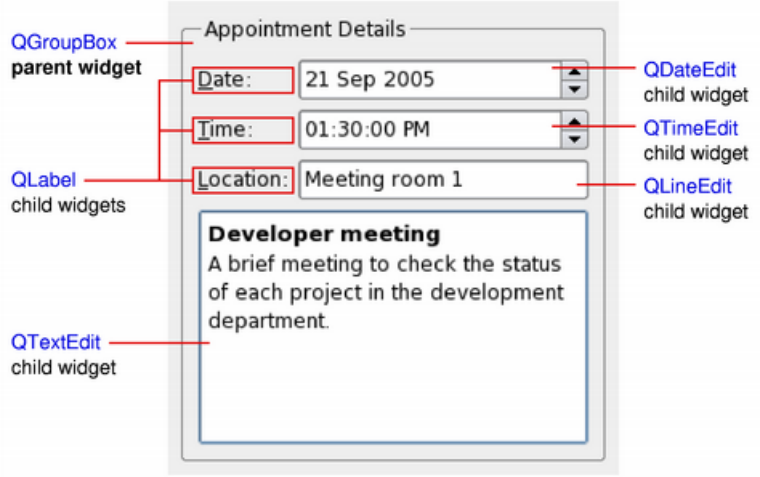


Figure 1.2: Image shows how the widgets are placed and look in the compiled application. Widget classes are in blue. Notice the layout of the widgets. The figure is acquired from Qt Documentation[7].

1.2.3 Qt Core

Buttons require signals and slots to function. These are used for communication between objects in Qt. Every click on a button sends a signal to the widget containing it or a manually connected method. In order to react, we need a slot for this signal. The slot is a method containing code that will be executed after it receives the signal. This is an alternative to the classic callback technique. Compared to callbacks, they are slightly slower. However, the mechanism is type-safe. Meaning, the signature of a signal has to match the slots' signature. Therefore the compiler will detect a type mismatch. Also, the program

does not have to catch a signal using a slot, nor does a slot need any signal to be assigned to it. Furthermore, several slots can react to one signal at once and vice versa. Every class that inherits from `QObject` or any of its subclasses can contain the mechanism, including `QWidget`.

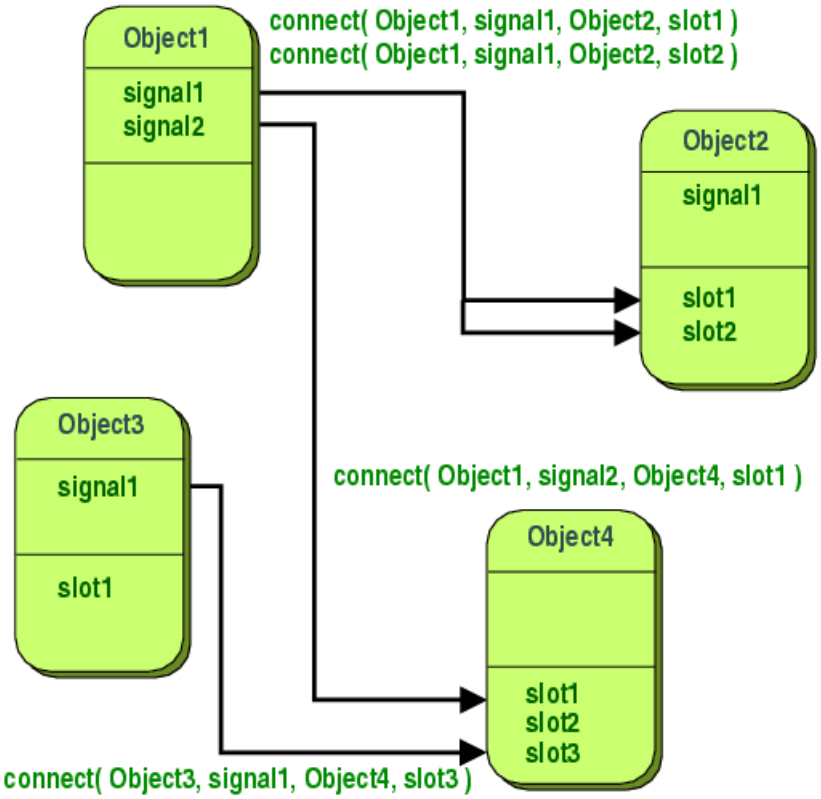


Figure 1.3: This is an abstract example of the connection mechanism. Connect function takes an object providing the signal and the signal itself, then the other object and its slot. The figure is acquired from Qt Documentation[8].

A signal is emitted by an object when its state has changed in a programmer-specified way using the emit statement. When it happens, the connected slots are executed immediately, and the code after the emit statement will run after all slots have returned, unless queued connections are used, in which case the slots will execute later. If more than one slot is connected to a signal, they will execute in the order

they were connected. All signals are strictly void; they cannot have return values.

Slots are normal C++ functions. The only addition to them is the ability to be connected to signals. Even if the slot is private, it can be invoked by a signal from an unrelated class.

1.2.4 Graphics View Framework

A topology visualization requires having a framework capable of managing, rendering, and interacting with custom graphical objects. The `QGraphicsView` and `QGraphicsScene`, the latter being a part of the former, provide ways to implement this. Besides the quality of life functions like zooming and rotation support, the graphical items we create can be interacted with by handling key events, mouse, or even track mouse movement. To discover items on the view field quickly, the view uses a Binary Space Partitioning (BSP) tree.

The `QGraphicsScene` acts as a container for the items, with the following responsibilities:

- Providing a fast interface for the item management.
- Propagating events to items.
- Managing item state.
- Rendering.

Items can be added by simply calling `QGraphicsScene::addItem()`, which takes an item pointer. However, it is important to note that the class will take over the items' management, including the memory destruction. Placing an item, managed by a unique pointer, as an argument might result in memory issues since both the unique pointer and the `QGraphicsScene` class manage their own memory.

The view itself is a widget that visualizes the content of a scene. Several views can be attached to one scene to provide different views into the same data. It also receives the user input and sends it to the scene.

All items share one base class called `QGraphicsItem`. Many other classes inherit from it, which are representing different geometrical objects. For example, `QGraphicsRectItem` represents rectangles. If we want to create a custom item class, we can inherit from any existing item classes and override their methods. These methods include mouse press, double click, hover, wheel or context menu events, key-

board input, or collision detection. Items can also contain other items. Parent items' transformation is inherited by its children as well.

1.2.5 JSON Support in Qt

Although JavaScript Object Notation (JSON) format indicates the use of the JavaScript language, nowadays, it is used widely as a readable data exchange format. Qt provides its own C++ API to parse and save JSON data. JSON is used to store data in a structured manner. The basic data types it supports are:

- double
- bool
- string
- null
- array
- object

A JSON value can have any of the previously mentioned data types. The text representation is relatively simple. Arrays are enclosed in square brackets and objects in curly brackets, while commas separate entries in both, and the separator between keys and values is a colon. More information about JSON support in Qt can be found in the Qt documentation[9].

The JSON format can be helpful, especially when exchanging information between different programming languages, where we do not have the option of straightforward, direct exchange. Additionally, data can be saved using it, as the syntax is reasonably readable.

1.2.6 Multi-platform file support

When we want to call a function at run-time, we can use the `QLibrary` class. This class contains a shared object file. Depending on the operating system, the extension can be `.dll` on Windows, `.so` on Linux, or `.dylib` on Mac. Since Qt is a multi-platform framework, `QLibrary` tries to add the file suffixes automatically based on the operating system. Therefore, only the name is required, omitting the suffix, and the same code will work on multiple systems.

To use the library, we dynamically load a file by calling `load()` and then resolve a symbol inside the file using the `resolve()` method. The

output of the `resolve()` method is the C function contained within the loaded file. This process is called dynamic linking and will be explained in the following chapter.

The same multi-platform support is present in the `QFile` class, which works similarly to the standard library `std::fstream`. Unlike it, however, `QFile` is more high-level and offers another class, `QFileInfo`, for more information about the file. The file paths are handled automatically in a system-specific way. Therefore the program does not need a different code for every OS. There are, however, issues with the file permissions.

"Qt's understanding of file permissions is limited, which affects especially the `QFile::setPermissions()` function. On Windows, Qt will set only the legacy read-only flag, and that only when none of the `Write*` flags are passed. Qt does not manipulate access control lists (ACLs), which makes this function mostly useless for NTFS volumes. It may still be of use for USB sticks that use VFAT file systems. POSIX ACLs are not manipulated, either." (Qt documentation, 2021)

1.3 Shared libraries in C/C++

1.3.1 Shared libraries

A classic static library is linked to the program at compile-time and makes a copy of its own code the program uses, which is stored in the resulting executable, increasing the overall size. Also known as `.a` in Linux, or `.lib` in Windows, files. This approach makes the execution quicker, as the code is already linked and present in the program at run-time.

A different type of library is a shared one. The code is stored only in the library itself and is referenced by programs using it at run-time. It is possible to have several executables using one shared library this way. This makes the binaries smaller than in the case of the static approach. Furthermore, the programmer does not have to recompile everything when making changes only to the library or the program using it. Linux uses `.so` (shared object), Windows `.dll`, and OS X `.dylib` files. These vary in their implementation and therefore speed, but their usage is very similar[10].

1.3.2 Dynamic linking

A shared library can be statically linked or dynamically linked. The latter offers the advantage of having the possibility to create plug-in systems, where an application takes a user-made library as input and executes it without the need to be recompiled. The performance costs are greater than the static linking since a part of the linking process needs to be performed every time the executable runs. Mainly finding all the used dynamically linked symbols in a symbol table and resolving them. This process is slightly cheaper in the Windows DLL (Dynamic Link Library) implementation because it does not have to map and relocate the dynamic linker in all of the processes before linking [10].

To sum up the benefits:

- Dynamically linked libraries are easier to create and update than static ones.
- The semantics of dynamically linked libraries can be closer to unshared libraries.
- Dynamic linking allows loading and unloading routines at runtime.

Explicit dynamic linking in a program to load a shared library in ELF can be done using `dlopen()` to load the library and `dlsym()` to find the address of a symbol that is required. These functions are wrappers that call back into the dynamic linker. When a library is loaded via `dlopen()`, a similar symbol resolution and relocation is performed as with any other library so that the dynamically loaded program can call back to routines already loaded in the caller of the shared library contents. This is done without stopping or restarting the program. Windows uses `LoadLibrary` and `FreeLibrary` instead. Getting an address of the function is done using `GetProcAddress`[10].

2 BatSim application

We created an application named BatSim, which simulates the aforementioned power-balancing in distributed battery networks. There are both GUI and CLI versions of BatSim. The CLI version is very simplified and does not offer manual topology creation using a set of tools and saving it for later use in a JSON file, as the GUI version does. Both do offer a way to start the simulation and produce its results stored in another JSON file. Strategies need to be programmed and compiled as a shared library. Some are already pre-programmed in the application and will be compiled automatically with the rest of the BatSim source code.

2.1 Installation

The Qt framework library with Qtwidgets and Qtcharts modules is required to compile the source code, which can be acquired through a free license on the Qt website [3]. Other requirements are CMake, a C++ compiler with c++20 support, and either Windows or Linux operating system. The source code and further installation instructions can be found on Gitlab at `https://gitlab.fi.muni.cz/xkumor/batsim`. The CMake will install both the CLI and GUI versions of the software, together with some example strategies.

2.2 Usage

When we want to create a basic circle topology with every node having the same energy consumption and run the simulation, the process would contain the following steps:

- Create how many nodes are required using the Create Nodes tool and left-clicking.
- Link them in a circle using the Link tool. Yellow nodes are the currently selected ones.
- Choose the Edit tool and select half of the nodes. Then right-click on any of the selected ones. This opens the settings window for all of the selected nodes. Add an action. Since we want a constant

energy consumption, we can leave the number of ticks at 1. The other number that is asked is the energy consumption per tick. We can put a 5 in there and confirm. After adding the action, we can close the window.

- Now click on the menu and import a strategy from a file.
- Run the simulation.
- Evaluate the results. Moving the slider at the bottom of the screen shows every turn of the simulation in detail.

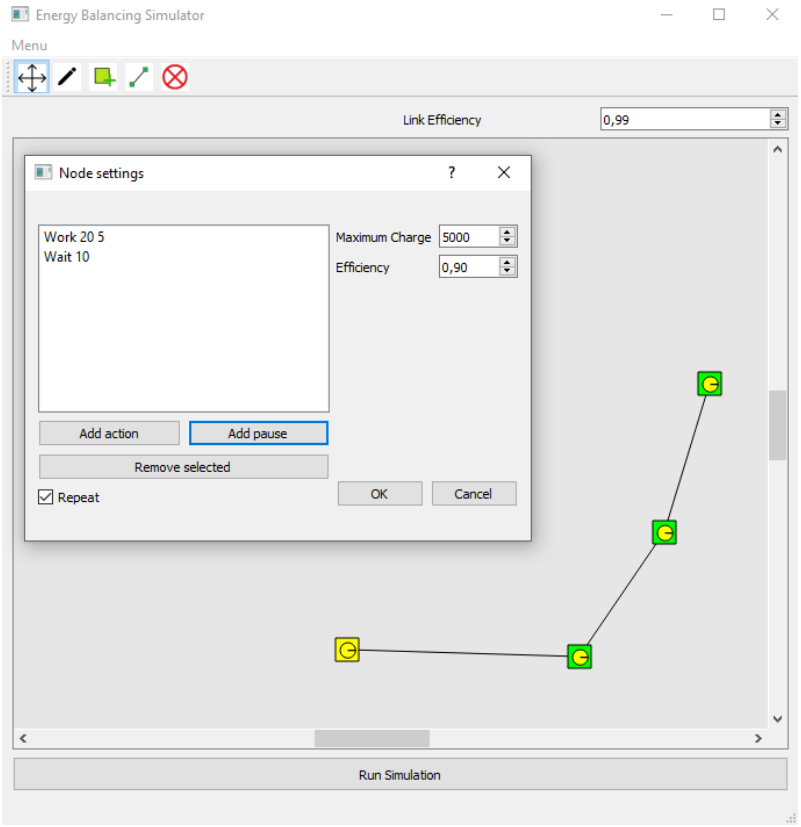


Figure 2.1: A screenshot showing the application with a basic topology and Move tool selected. Nodes are green rectangles. The yellow rectangle is a selected node, with its settings window opened, containing work and wait operations in its time plan.

2.3 Graphical User Interface

The opening screen shows a topology screen, where the topologies can be created. Here, the user can create topologies using customizable nodes. These nodes can contain a custom power consumption behavior to simulate the power usage the user wants. Additionally, it is possible to set the battery charging and discharging efficiency. They can also be linked together, meaning they are connected together using a wire to transfer power between each other. For such editing, it offers various tools:

- Move - to move the nodes or the screen by dragging.
- Edit - to be able to select and edit multiple nodes at once.
- Create Nodes - creates a new node by left-clicking.
- Link - after selecting a node, which turns the color to yellow, and clicking on another node, this tool will link the two nodes. This represents the wiring.
- Delete - deletes objects after they are clicked.

Right-clicking a battery node using all tool modes, except Delete, opens up an interface called Node settings. Here, the battery consumption behavior can be set by creating actions, using energy consumption per turn and the number of turns it will last. Additionally, a pause can be inserted. By default, nodes do not consume any energy when there are no actions set. The Repeat checkbox makes the actions repeat after they all end. Efficiency dictates the battery's energy loss when charging and discharging.

When multiple nodes are selected, and one of them is right-clicked, the settings will be propagated over all of them. However, if the selected group does not share the exact same actions, the current ones will not be displayed, and only new ones can be added. Deletion of items is disabled in this case.

The menu in the left top corner allows us to save or load a topology. Since JSON files are used, they are also readable in a text editor. The save file is composed of an array containing efficiency, id, maximum capacity, an array of neighbor ids, a plan object, and the node's X and Y positions. Using the Import strategy option and selecting the shared library containing it, the application loads the user-made function and makes it ready to use by the simulation. The last item on the menu is the Load simulation result, which will import saved results

of a simulation and open a new window showing them. This file is composed of both the topology mentioned above and the result array itself. The array items represent turns. Each turn is another array, holding information about every node in the topology. This includes remaining energy, the efficiency of the last transmit, and the receiver's id if the node sends power this turn. If the id is a negative one, no receiver is set, and therefore, the node is not sharing energy.

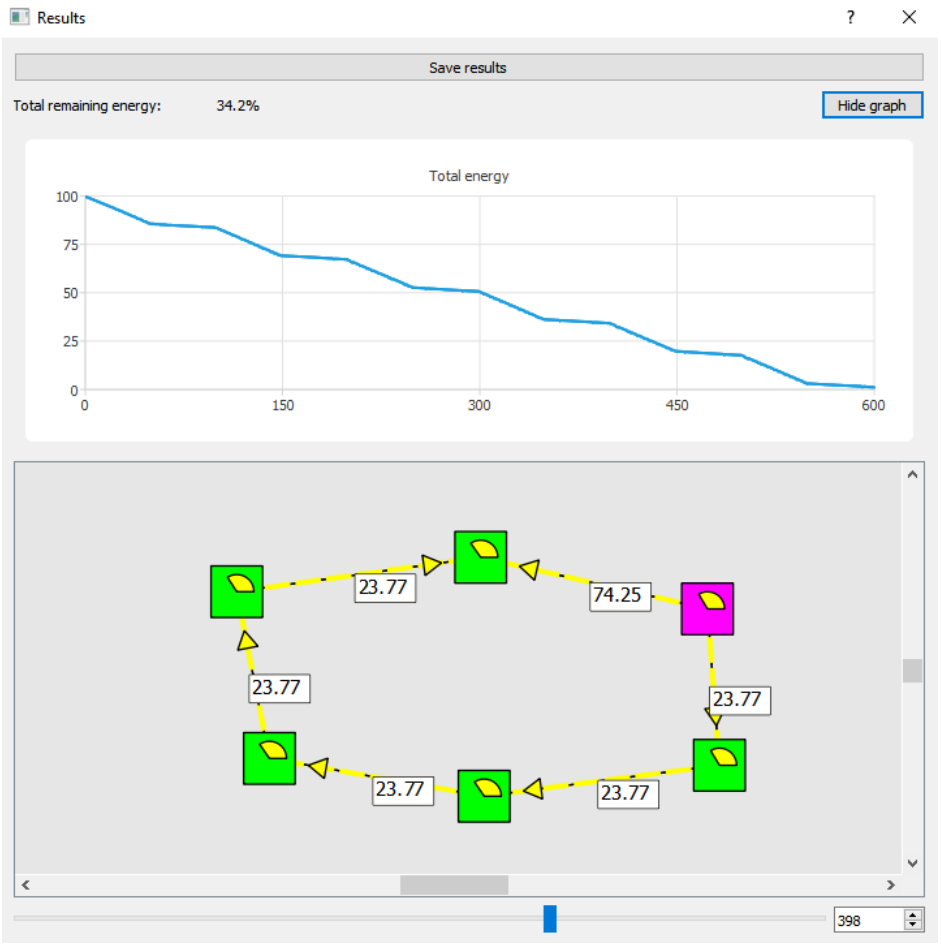


Figure 2.2: A screenshot of visualized results. The purple node is the sender. Yellow lines with arrows show the direction of the energy flow. Bottom slider shows which turn is displayed.

The Run Simulation button at the bottom will execute the simulation itself using the strategy and the topology on the screen. Before clicking, make sure that there is only one topology with no unconnected nodes and a node with power consumption exists. If no strategy is set, the simulation will not use any power-sharing. After some time, depending on the number of nodes, connections, and battery capacities, the Results window will appear. At the bottom, it shows a time slider. If moved, a graphical representation of the simulation's state will be rendered at the selected time frame. The yellow pie turns into a red one when the battery capacity reaches the last 10 % so that the amount can be viewed more accurately. Additionally, if there is a sender, paths of the energy are displayed for every turn, including the amount of power going through every connection. If we want to review the overall energy consumption in time, there is a button in the upper-right corner to display a graph with this information. The Y-axis represents the percentage of the total energy stored in the topology.

By using the JSON format for saved topologies, it is possible to create scripts for topology generation in Python or another scripting language. This is useful when users want to create extensive but regular topologies, which would be time-consuming to make in the GUI application. In this case, knowledge of simple manipulation of JSON files and the syntax[11] is required.

2.4 User-made strategy

If the user wants to create his own power-sharing strategy, it is required to have it in its own source file and then link the simlib library to the project and include the "`strategy_interface.h`" header in the code. Linking can be done using CMake's `target_link_libraries`. After that, the strategy function has to be declared in the source file using one of the following:

- `extern "C" void strategy(ebs::InterfaceVector &nodes)`
- `extern "C" __declspec(dllexport)`
  `void strategy(ebs::InterfaceVector &nodes)` (Windows)

This is necessary for the creation of a dynamically linked shared library. The insides of the function are up to the user, using the interface methods for the nodes contained in the function argument, which is

a vector (very similar to a list) of node interfaces. All these methods are located in the interface. They allow for getting information about the node charge, maximum capacity, current consumption, battery efficiency, currently set receiver, distance from it, and a vector of direct neighbors. Another method makes it possible to set a receiver.

To initiate power-sharing, setting a receiver in one of the nodes is required. The node that sets the receiver will be the sender, and the simulation will automatically transfer power each turn. There should be only one transfer at a time to share power efficiently unless the power is directly transferred between neighbors with all other batteries and wires disconnected from the network at the time of transfer. Otherwise, there would be multiple inefficient transfers in real life, which the application does not support. This means the receiver should be manually unset in the sender node the following turn if it is no longer a viable sender. Also, the method setting a receiver will throw if the node is not reachable.

Typically, a strategy consists of getting information about nodes, evaluating which node needs power and which one can provide it, and then setting the receivers. Since the strategy is executed every turn, it is recommended to reset all of the receivers before each turn. Another option is using static variables to store information from previous turns. However, running a second simulation with the strategy unchanged will not remove the static variable contents from the previous simulation.

An example strategy will find a node with the most available capacity and a node with the least available capacity by iterating through the nodes. Then it sets the less powered one as a receiver in the most powered node. The strategy remembers the sender node between calls in order to unset the receiver in another turn and reevaluate who will transfer again.

After writing the strategy file, it is required to compile it as a shared library (`.so` or `.dll`) which is possible using `add_library` in CMake instead of the usual `add_executable`.

```cpp
#include "strategy_interface.h"

#define THRESHOLD 110 // send power when the difference is higher than this

#ifdef _WIN64
#define EXPORT __declspec(dllexport)
#else // linux/mac
#define EXPORT
#endif

/**
 * @brief Sends energy from the most powered to the least powered battery.
 */
extern "C" EXPORT void strategy(ebs::InterfaceVector &nodes) {
    using namespace ebs;

    // reset all set receivers
    for (auto &node : nodes) {
        node->setReceiver(nullptr);
    }
    NodeInterface *smallest = nodes.front();
    NodeInterface *biggest = nodes.front();

    for (auto &n : nodes) {
        if (smallest->getCharge() >= n->getCharge())
            smallest = n;

        if (biggest->getCharge() < n->getCharge())
            biggest = n;
    }
    if (biggest->getCharge() - smallest->getCharge() > THRESHOLD)
        biggest->setReceiver(smallest);
}
```

Figure 2.3: An example strategy code. The export macro makes the strategy compatible on both Windows and Linux, but it does not have to be present on Linux at all. The rest of the function definition is essential for shared libraries to function. This example uses a static node interface pointer variable to unset the receiver the next time the strategy is called.

2.5 Simulation details

The structure of the simulation is based on three classes:

- Timeplan - is a vector of actions, where an action is a subclass, consisting of a duration (number of turns) and the amount of energy consumption per turn. A pause has zero consumption. The Timeplan class holds information on whether to repeat the actions after they all end and what action will be executed the next turn.
- Node - represents the modules, together with their batteries, keeping information about the current charge capacity, battery efficiency, its own Timeplan, pointer to the receiving node if set, and the handles to its neighbors.
- Simulation - the main class that handles running the simulation, energy flow in the topology, containing the nodes and offering methods to the GUI tools.

Since it is turn-based, the strategy is run every turn, setting the receivers in every node and therefore choosing the roles. All nodes have a routine that is executed each turn as well, which includes consuming energy according to the Timeplan, checking if a receiving node is set. If true, the node initiates sending. Otherwise, it checks whether the remaining capacity is below zero, either returning a false boolean value to stop the simulation or a true to continue. Additionally, this process involves collecting information about the nodes in each timeframe.

The energy sending process from sender to receiver involves calculating the maximum charging speed based on the battery's charge level, as mentioned in the figure 1.1. Then the amount is subtracted from the sender's battery. After that, all paths from sender to receiver are found using a modified depth-first search (DFS) algorithm. According to Kirchhoff's laws, which describe how the current is split between the wires, the program assigns a resistance value to every valid path and then, using the current division rule calculates the energy that will go through each individual path and stores the loss caused by wire resistance. While this happens, the program finds cases where the power goes in opposite directions. These are resolved based on the amount of power in each direction. Either both or only one of those paths are then deleted. The changes are summed up, and the

result is multiplied by the receiving battery efficiency. The total energy loss information is then saved and can be accessed using one of the interface methods in strategy. It will also be displayed in the graphical representation of results.

3 Case study

3.1 Setup

For this study, we present five different topologies and three strategies. The focus is to compare how much they prolong the operational time of a network, whether some of them are better in every case, or if any topologies prove difficult for all of them. Topologies are used with different power consumption configurations, mainly centralized or decentralized areas with power usage and either static or dynamic node power consumption. Static consumption consumes the same amount of energy all of the time. Dynamic consumption differs in the amount consumed in time, but the average remains. Decentralized consumption is located all over the topology, with some distance between the nodes. All nodes have a maximum charge of 5000 units with 90% battery efficiency and the links between nodes have 1% loss.

3.1.1 Strategies

The first strategy, named "Highest to Lowest" (HTL), is a simple one. It is based on finding the node with the highest power available and another node with the lowest power. If their current charge difference is higher than the threshold, set to 110, which is ten power units higher than the maximum charge per tick (MCPT) a battery can receive, the transfer is started. At the beginning of the strategy, all receivers are reset as we want only one transfer at a time. Additionally, both the highest and lowest nodes are recalculated every tick of the simulation.

Another one is similar to an existing strategy proposed by an already mentioned professor Qadir[4]. "Direct Neighbor" (DN) strategy initiates transfer only between two direct neighbors without other wires being connected to them at the time of transfer. Therefore no passthrough modules. Every node searches for a neighbor with the highest charge, higher than themselves, and if the neighbor does not already give power to someone else with less charge, it will change the receiver. Again, the power will be transferred only if the charge difference is higher than the threshold mentioned above.

The last, "Longest to Shortest" (LTS), is inspired by the near-optimal strategy[5]. However, it works solely on the estimated lifespan of every node and is similar to HTL. It finds two nodes for each tick, the one with the longest and another with the shortest estimated operational time. The transfer occurs between the two if the charge difference is higher than a threshold, in this case, set to 100. The operational time is estimated by the following formula:

$$Lifespan = \frac{Current\ charge}{Current\ power\ consumption} \tag{3.1}$$

3.1.2 Topologies

Topologies were created using the tools of the BatSim application. They were chosen based on the possible power flow differences and potential real-life usage. Some of the topologies are frequently used in modular robotics, like spiders and centipedes. Used topologies are visualized below:

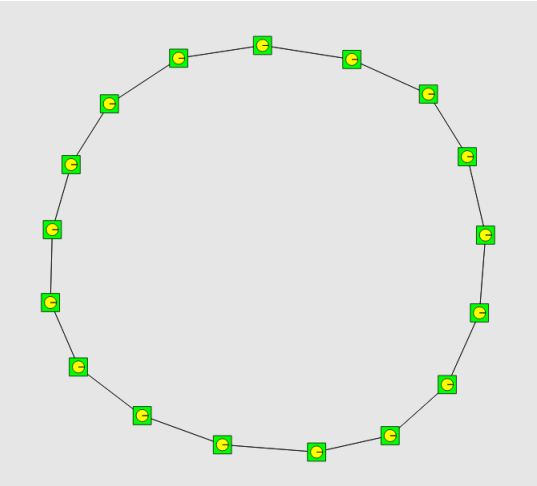


Figure 3.1: Here is a circle topology with 17 nodes, where the power can flow using two ways.

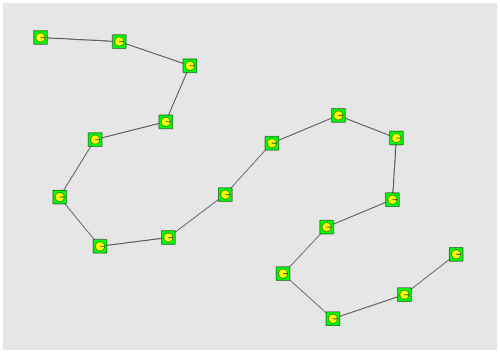


Figure 3.2: A snake topology is often used as a centipede robot. There are 18 nodes in total. The power always flows through one path.

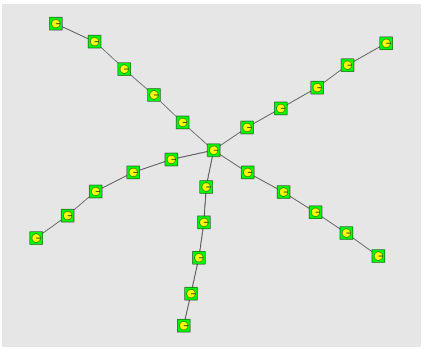


Figure 3.3: A star topology can form a spider, for example. Every arm has five nodes, 26 nodes in total. Here we test only the decentralized configuration.

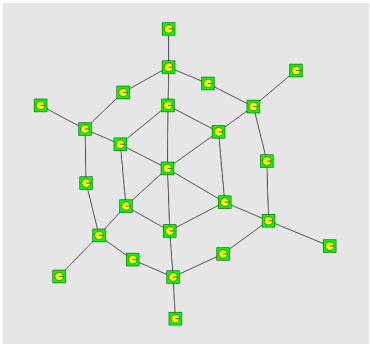
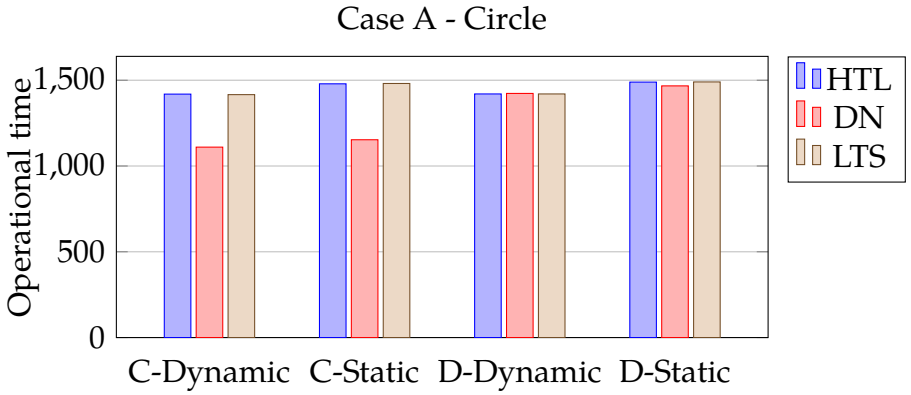


Figure 3.4: This is a hexagonal spider web topology with 25 nodes. The power can flow through a lot of different paths.

3.2 Results

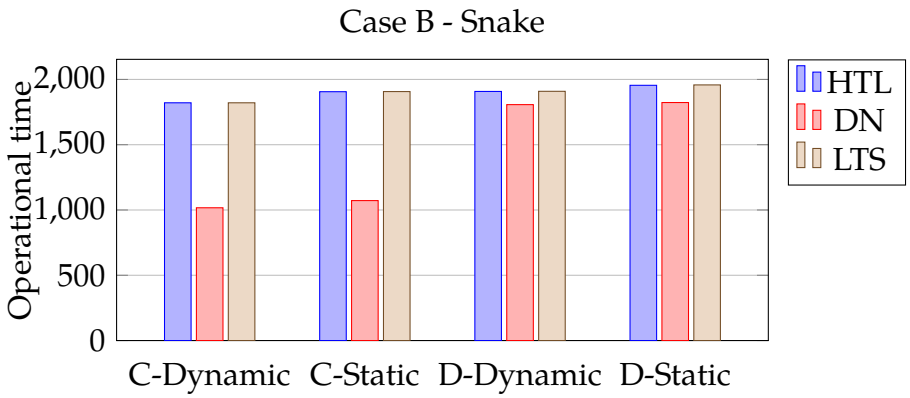
Case A is the circle topology. The centralized configuration includes four nodes next to each other, consuming ten power per tick on average. The decentralized version has these four nodes with some distance between each other. Also, there were no particular expectations in this case, except the DN performing worse than the other two strategies since it performed poorly in the testing phase.

The results showed that the DN strategy is performing well in the decentralized power consumption case and even being the best one in the case of dynamic consumption by lasting three ticks longer than the HTL. However, this extremely small difference is not enough to choose the best performing strategy. Therefore all three performed equally well in decentralized configurations of the circle topology. Meanwhile, the centralized one confirmed the expectations of being the weak point of DN. The HTL and LTS ended up having equal performance.



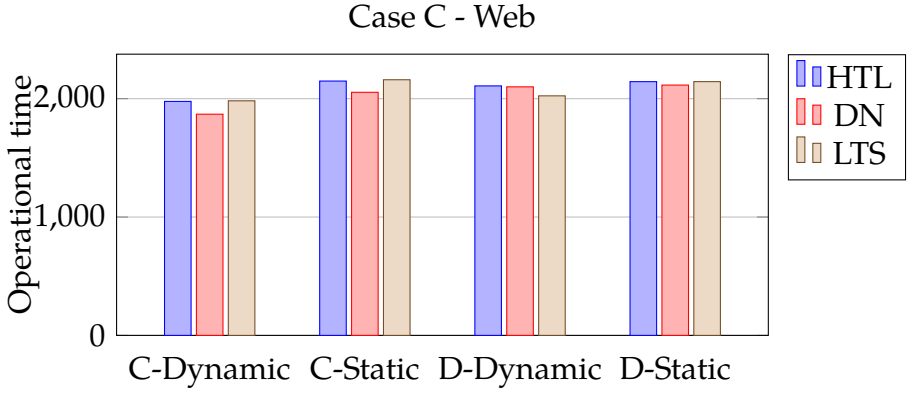
Case B is the snake topology, another expected weakness of the DN strategy. As in the previous case A, there are four nodes with 100 average power consumption per tick. In the centralized configuration, they are all at the lower end of the topology.

As expected, the DN strategy performed even worse than in case A in all configurations. Furthermore, in the centralized one, more than 11% of the total energy in the network was unused. The HTL and LTS were equal again and only left roughly 1% of the power in static and 2-3% of the power in dynamic configurations unused.



Case C with the web topology had five nodes in the middle of the topology, excluding the top and bottom of the innermost hexagon, consuming energy in the centralized configuration. All nodes have ten average power consumption as in previous cases. This topology is special in the number of possible paths when transferring energy as it is very interconnected, which should ease access for the DN strategy especially.

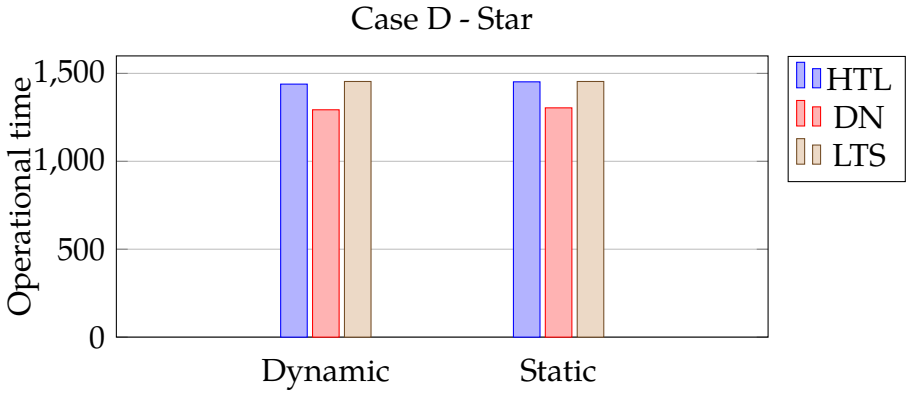
The results showed that the HTL strategy was good in all four categories, while the DN was poor in the first two categories as previously. The LTS did worse at the dynamic configurations, leaving roughly 2% more energy in the network unused. Since there were no differences in the consumption time plans and other topologies did not contain this occurrence, the only remaining factor is the combination of the topology itself and the dynamic power usage.



Case D was only checking the decentralized power consumption to simulate a spider. It contained ten nodes with eight power consumed

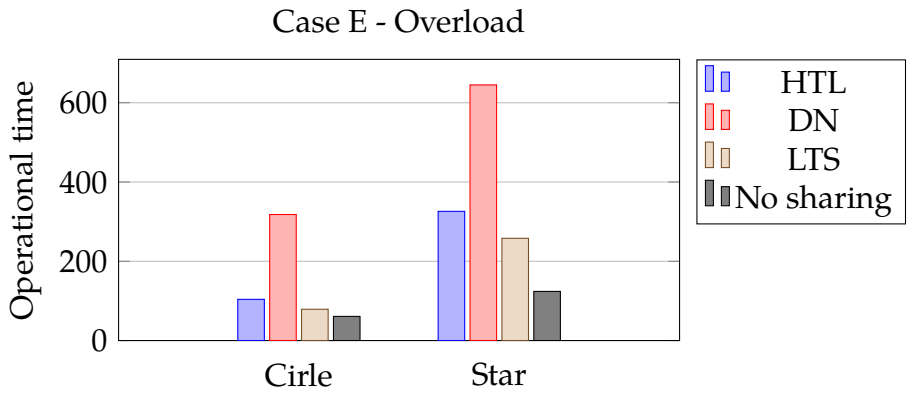
per tick on average at the tips of the arms. This topology is slightly better interconnected than the snake or circle topologies. For the DN strategy, however, the central node is a bottleneck.

Both the LTS and the HTL performed equally well, with the DN working poorly for this topology. However, no notable difference between static and dynamic consumption.



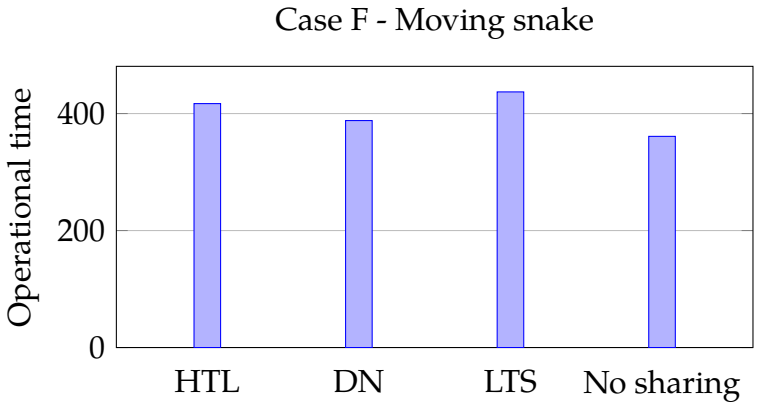
Case E includes circle and star topologies with a higher load than the maximum amount of power charged per tick (100). This means the most effective strategy will be the one that charges the power-intensive nodes fastest. There are also two statically and two dynamically consuming nodes in the circle topology with constant consumption of 20 and 120 for 20 ticks with 10 ticks pause, respectively. The star topology has the central node with a constant 40 consumption and all five tips of arms of 50 consumption for 20 ticks and then a waiting period of 20 ticks. This means that the total power usage has significant pulses. Since the efficiency should be lower, we include the results of no power-sharing as well.

The DN is very dominant in the overloaded situation, with the overall remaining energy in the network below 9% in the circle and roughly 3% in the star topology, despite having weaker results in case D. The operational time was five times longer than no power-sharing in both topologies. Other strategies had over 50% of unused energy, in the case of the circle, even more than 70%. Overall, the first topology proved more difficult for all strategies, as they ended up being very close to the result of no power-sharing. Furthermore, the HTL performed significantly better than the LTS in this case.



Case F is a very specific example where the HTL strategy should theoretically perform worse than the LTS since the workload "travels" across the topology, and it might be better to focus on the actively working nodes than those without any load. Every ten ticks, the following pair of nodes starts consuming power, from the top left to the bottom right end of the snake. The first of the nodes has a consumption of 120 units per tick, and the second one has half of the amount. Since every node has some consumption, no power-sharing should leave roughly one-quarter of the total energy in the batteries, being fairly effective on its own. Therefore we do not expect any significant improvement. However, if there were additional nodes with no consumption, no power-sharing would have poor results.

The results confirmed that there is a case where the LTS works better than the HTL strategy. Although, the difference is not very significant. The DN is the worst one in this case, most likely due to the charge levels being evened out by the character of the consumption itself and the topology itself, which does not favor the DN. All strategies left less than 10% of the power in the network, which is better than in case E.



3.3 Evaluation

In conclusion, the topology had a significant impact on the strategy performance. The same applies to the character of power consumption and its distribution in the network. However, we only tested a small part of what can be tested. Therefore there is still potential for other findings and possible discovery of other weaknesses or advantages of the strategies mentioned in this thesis.

The HTL strategy worked very well in almost all cases, with the exception of cases E and F, although the performance in case F was reasonable. It functions reliably in all types of workloads, except overloading. It is a well-rounded strategy that can be used in low workloads and all presented topology types with good results.

The second one was the DN strategy which proved to be the best choice for overloaded battery systems with high consumption, as shown in case E. However, it only performed well in interconnected topologies, unlike the ones with the minimum possible connections like the snake or star topology. A notable result is a high performance in the second situation of case E, compared to case D, where we got contradicting results in the same topology. This shows that the DN was able to achieve its maximum potential despite the overload since it had roughly the same total remaining energy in the network (around 3%). However, there was an issue with the centralized cluster of power-consuming nodes, where the performance was much lower than in the case of the other strategies. Therefore we can conclude that the best

usage of this strategy is in well-interconnected topologies or when the power consumption is very high and not centralized in one place.

The last one, the LTS strategy, was very similar to the first one in its functioning. However, it performed worse in some cases, and we only found one case where it performed slightly better. This is not to discard the strategy entirely, as the number of possible combinations of topology types, power consumption configurations, and distribution is extremely high. It also performs worse in web-like topologies and does not handle overload well, except in case F, where the moving power consumption made calculating the lifespan advantageous. Overall, is it still a useful strategy in most topology types, except the one mentioned above if the workload is low or very specific, as mentioned in case F.

Dynamic workload proved slightly more difficult for all strategies, which could be an issue with power-sharing in general since it is harder to estimate how much energy to transfer when the charge changes unpredictably. However, this thesis does not have enough data to confirm this theory, as there could be a strategy that works better in this regard. Another limitation was the strict number of senders in the HTL and LTS strategies, which was also the case in one of the previous studies[5].

Since the simulation uses simplified energy flow through the network, the web topologies might have different results if the power loss in the link was significant. This will usually not be the case in modular robotics, where the links are short, meaning there is not much space for more energy loss than 1%. In reality, the loss is even much smaller than that. Therefore the only notable loss comes from the charging and discharging efficiency of the batteries.

4 Conclusion

The main focus of this thesis was creating a power-sharing simulator for self-reconfigurable modular robots, with tools to create custom topologies with customizable power consumption and a way to insert user-made strategy and then review the results in every tick of the simulation. Furthermore, the output data should be in a format that other programs can easily read for further information processing, like the JSON format.

The first chapter of the thesis described the basics of modular robots, the concept of power-sharing, and the physical properties of batteries, together with the basic rule of current division. After that, we presented the Qt framework used for the application GUI and file management and ended the chapter with dynamically linked shared libraries. The application, including the installation, was introduced in the second chapter with some screenshots and basic usage. The third chapter presented the process of the case study with four different topologies, reviewed the results, and evaluated them.

After performing tests on different topologies, with the three strategies which were implemented, we have confirmed the assumption of topologies having a significant impact on the performance of certain strategies and concluded the advantages and disadvantages measured in the case study, mainly the widely usable HTL strategy at low workloads and the operational time extension at high workloads the DN strategy offers. These findings can be used to optimize strategy selection when deciding which one to use in which cases when working with modular robots when the goal is to maximize the robot's operational time.

However, there is still the possibility of improving both the simulation and the tools the application offers. The calculation of current division in topologies that are not parallel or serial either can be made more realistic unless it slows down the simulation more than is viable. The GUI could also offer tools for creating graphs based on multiple parameters, possibly some statistical tools as well. For now, this task is handled mainly by the user creating his own scripts to visualize the contents of a file with the simulation results if the user needs to. Since the work is open-source and licensed under the GPLv3 license[12], it

is possible for anyone to copy and modify the application presented in this thesis under the terms of the license mentioned.

Bibliography

1. *The RoFI Platform* [online]. Brno: ParaDiSe Laboratory at Faculty of Informatics, Masaryk University, 2021 [visited on 2021-03-29]. Available from: https://rofi.fi.muni.cz/.
2. BUCHMANN, Isidor. How to Charge Li-Ion with a Parasitic Load. *Battery Power* [online]. [N.d.] [visited on 2021-04-18]. Available from: https://www.batterypoweronline.com/articles/how-to-charge-li-ion-with-a-parasitic-load/.
3. *Qt | Cross-platform software development for embedded and desktop* [online]. The Qt Company, 2020 [visited on 2021-04-19]. Available from: https://www.qt.io/.
4. QADIR, Raja Humza. *Self-Sufficiency of an Autonomous Reconfigurable Modular Robotic Organism*. 2015th ed. Springer International Publishing, 2015. ISBN 978-3319102887.
5. CHEN, C.; COLLINS, T.; SHEN, W. A near-optimal dynamic power sharing scheme for self-reconfigurable modular robots. In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 5183–5188. Available from DOI: 10.1109/ICRA.2016.7487724.
6. *Qt Creator - A Cross-platform IDE for Application Development* [online]. The Qt Company, 2021 [visited on 2021-04-23]. Available from: https://www.qt.io/product/development-tools.
7. *Qt Widgets* [online]. Qt Documentation, 2021 [visited on 2021-04-13]. Available from: https://doc.qt.io/qt-5/qtwidgets-index.html.
8. *Signals & Slots* [online]. Qt Documentation, 2021 [visited on 2021-04-13]. Available from: https://doc.qt.io/qt-6/signalsandslots.html.
9. *JSON Support in Qt* [online]. Qt Documentation, 2021 [visited on 2021-04-13]. Available from: https://doc.qt.io/qt-6/json.html.
10. LEVINE, J. R. *Linkers and Loaders*. Morgan Kaufmann, [n.d.]. ISBN 9781558604964.

11. CROCKFORD, Douglas. *Introducing JSON* [online]. Ecma International, 2021 [visited on 2021-05-01]. Available from: `https://www.json.org/json-en.html`.

12. *GNU General Public License* [online]. Free Software Foundation, Inc., 2021 [visited on 2021-05-12]. Available from: `http://www.gnu.org/licenses/gpl-3.0.html`.

A An appendix

The attachment file `case_study.zip` contains strategy libraries and JSON files with the topologies used in the case study. Additionally, `source.zip` contains the git repository files of the BatSim application as of May 2021.