ROS Framework for Distributed Control of Networks of Dynamical Systems

M. Rossi*, A. Jokić†
* sees.ai, United Kingdom
† Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb, Croatia
mrossi@sees.ai, andrej.jokic@fsb.hr

***Abstract*—In this paper we present a software package developed to accommodate flexible and efficient modelling and simulation of networks of dynamical systems with distributed control structures. The package is implemented within the Robot Operating System (ROS), which is becoming a standard software implementation tool in robotics, mechatronics and wider. The developed package is characterized with the following features: implementation of distributed controllers with both off-line predefined control laws in closed form and with on-line iterative solving of an optimisation problem (distributed model predictive control); modular structure which allows the user to easily modify the subsystems in the network, as well as the number of subsystems; allows for straightforward implementation of controllers onto real physical systems.**

***Keywords*—control systems, dynamical networks, robot operating system, distributed control**

I. INTRODUCTION

With the recent advances in communication and information technologies, and the development of new generations of sensors and actuators, a whole new set of systems started to emerge, which were unfeasible only years ago [1]. Some examples of such systems are: the so-called smart structures, composed of large amounts of sensors and actuators mounted on structural elements with the purpose of vibration dumping [2] or fluid flow control [3]; adaptive optics [4]; smart electrical power grids [5]; platoons of vehicles on automated motorways [6]; or large groups of ground mobile robots, unmanned aerial vehicles, or autonomous underwater vehicles, that collaborate towards a common goal [7], [8], [9]. The common characteristic of those systems is that they are composed of a relatively large number of spatially distributed dynamical subsystems, which interact with each other trough physical interconnections and/or communication links. Such systems are called networks of dynamical systems, or, shorter, dynamical networks.

The main focus of today's research in this area is the development of methods for control algorithm synthesis [1]. In most cases the challenge is not the design of subsystems, which can be very reliable when operating individually, but instead the design of local control laws, possibly in combination with distributed coordination schemes, for achieving common objectives at the network level. At the moment there is still no well understood, mature and widely applicable theory that offers scalable, robust, and reliable solutions to real-life network control problems. The research is currently still scattered, as problems of this nature are being explored also in biology, economics, sociology, game industry, etc. [10].

This paper contributes to the field by introducing a software framework that can be used for development, simulation, and real-life implementation of control algorithm for networks of dynamical systems. The framework allows modelling networks in the Python language, while the framework itself runs within the Robot Operating System (ROS).

The framework, called Dinsdale, is available as free software, under the General Public Licence, at `https://github.com/mross-22/dinsdale`.

II. CLASSIFICATION OF DYNAMICAL NETWORKS

To have a better understanding of the complexity of the field of distributed and decentralised control, and therefore to recognise the potential of the framework presented in this paper, it is useful to be familiar with the different types of networks present in control systems today.

A. Control structures

In the past decade, it has been acknowledged that fully centralised control structures are not capable to cope with the complexity of large spatially distributed systems. A centralised control structure is one in which one central control unit collects all the measurements and sends commands to all actuators. Such structures can guarantee globally optimal results of the controlled network, and their design is a mature field with a very well developed underlying theory. It has been proved that many practical problems with this structure can be formulated in terms of convex optimisation problems, and therefore efficiently solved [11]. However, some of their main limitations are that they are not scalable, not robust to failures, and are often practically impossible to implement.

The opposite of centralised structures are decentralised ones. In this case every subsystem is controlled by a local controller, and there is no collaboration between local controllers. Their main advantage is that no long distance communication is required, which makes them theoretically ideal for practical implementation. They also have some significant disadvantages, like the possibility to offer only suboptimal solutions which could possibly be

far from the global optimum, resulting in very inefficient behaviour. Using this structure the control synthesis is a non-convex problem, and there are no constructive algorithms for solving it [11].

The structure which has been recognised as often the most suitable for control of such large systems is the distributed structure. In such structure, each subsystem is controlled by a local controller which, apart from operating with locally available measurements, cooperates with a usually small set of neighbouring controllers. The controller communication network topology is in most cases required to be the same as the plant interaction network topology, but there could be exceptions.

B. Subsystem connection typology

The subsystems of a dynamical network can be physically coupled or decoupled. The former means that they directly affect each other's dynamics. Such systems range from inverted pendulums interconnected with springs, to complex electrical power grids, or the Internet.

If the systems are not physically coupled, it means that their interaction is through information exchange or measurements. They are connected by common goals or constraints, rather than physical links. Some examples are multirobot systems, vehicles platooning on motorways, and sensor networks.

C. Subsystem connection topology

Each subsystem has a certain set of neighbours, with which it interacts. This set can be either constant or change over time. In the former case the network topology is said to be static, while in the latter dynamic.

An example of subsystems that interact in a dynamic network topology are mobile robots which have a limited interaction range, due to limitations of sensing or communication equipment. This means that the set of each robot's neighbours is a function of their relative positions, which change over time.

D. Subsystem typology

The subsystems connected in a network can be all of the same type, or of different kinds. If the network is composed of identical systems, it is called homogeneous. Examples of such systems can be sensor networks or swarms of robots, given that all agents are of the same type. An interesting subset of such networks are spatially invariant networked systems, in which not only the subsystems are all equal, but also the dynamics of the system does not change moving along any spatial axis. An example are large segmented telescopes [12].

Networks composed of more than one type of subsystems are said to be heterogeneous. This is the case in the majority of applications, like electrical power grids, communication networks, or teams of different autonomous vehicles.

E. Control laws

The last important classification that is important to describe in this chapter is the difference of how the control laws of subsystems are being computed. The first possibility is that the controllers are predefined, which means that the control laws are computed offline, before operation. The second possibility is online solving of optimisation problems. In this case, each controller is computing its control output by solving an optimisation problem for every sampling period. This requires the controllers to iteratively exchange information between them inside one sampling period.

III. PROBLEM DEFINITION

Control of dynamical networks is a rapidly developing area of research, where most of the effort is put towards the development of a unified underlying theory. However, a universal software framework for development, simulation, testing, and practical implementation of such control algorithms is still missing. For this reason the transition from a general simulation to implementation on real systems requires big efforts and time investments. For example, it can be the case that people working on the theoretical development lack the programming skills required to implement complex distributed systems on specific hardware. Furthermore, even when having excellent programming skills and knowledge of embedded systems, this transition often requires a significant amount of time, having to reformulate the control algorithms in a form suitable for the particular system, and having to deal with communication protocols and limitations.

This has been the main motivation behind the work presented in this paper. The goal was to explore the possibilities of creating a software framework that will allow development of distributed and decentralised control laws for both simulated and real networks of dynamical systems, minimising or completely removing the transition from one to the other. The framework should be fully flexible to accommodate all the possible characteristics of dynamical networks.

A similar goal has already been achieved in robotics, by the Robot Operating System (ROS). The problem there was that a lot of robot control algorithms were being developed, but most of the time they were impossible to reuse across various systems and to connect with other existing software. Therefore, a lot of effort had to be put in the implementation process, often ending up rewriting existing software due to lack of portability. This is where ROS has contributed by offering a thin middleware which acts as a universal platform for software development, without being invasive (the software does not have to be designed specifically to work with ROS).

Since ROS already offers a standard systems to make different software communicate, and since there already exists a large amount of software packages for it, from hardware drivers to high level functionality, it has been

decided to use it as the base for developing the dynamical network control framework.

IV. ROBOT OPERATING SYSTEM

This is a very brief overview of the main concepts of ROS. A more detailed introduction, with focus on distributed system development, can be found in [13]. More in-dept information on ROS, as well as tutorials, can be found online, on the ROS website [14], and in books, like [15].

ROS is a framework primarily intended for development of robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a variety of robotic platforms [16]. ROS is free software, licensed under the permissive BSD licence. It was created, and, from 2007 to 2013, developed, and maintained mostly by Willow Garage, a research laboratory and technological incubator which produces hardware and free software for service robotics [17]. Since 2013 the project has been transfered to the Open Source Robotics Foundation (OSRF) [18], a non-profit organization born as a Willow Garage spin-off.

It is important to note that ROS is not a replacement for a computer operating system, but a middleware between the robot hardware and the control algorithms. From the robotics point of view however, it offers all the services and abstractions expected from an operating system, which explains its name. Some of the key services are hardware abstraction, low level device control, implementation of commonly used functions, process management, and package management. It also provides tools and libraries for obtaining, building, writing and running code across multiple computers [14]. At the moment, ROS runs on top of UNIX-like operating systems, with the best support on the Ubuntu GNU/Linux distribution and its derivatives.

Software in ROS is distributed in packages and stacks of packages. A package is simply a directory with a certain structure, which can contain executables, libraries, configuration files, or anything else.

At runtime, ROS manages the software inside what is called the ROS Computational Graph, which is basically a network of processes that can communicate with each other. The main concepts are: *Nodes, Master, Parameter Server, Messages, Topics, Services, Bags.*

The programs (processes) that are being executed are called nodes, while the edges of the graph are the interactions between them. The topology of the graph can be dynamical: the graph can be fully modified at runtime.

The main program (server) which allows the interaction between nodes, because it contains all the names and addresses, is called Master. The Parameter Server is a part of the Master which serves as a place for storing publicly available data in a centralised location. It is not very fast, so it is not used for runtime communication, but rather for storing some universal parameters.

There are two methods for communication between nodes: the asynchronous publishing of messages to topics, and the synchronous communication through services. A node can send a message by publishing it to a given topic. The topic is a name to identify the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There is no limit to the number of publishers and subscribers for a topic. Services are used in cases when request/reply communication is required. A node can offer a service under a specific name, to which any other node can send a request and wait for the reply.

The last fundamental concept that is important to understand are bags. Bags are files used for storage and reproduction of logged data. They offer a convenient way to store all kind of data during execution, which is very valuable for development and testing. The data from a bag file can be reproduced in the order it was collected, which allows for example testing different algorithms on the same data set.

V. THE DINSDALE FRAMEWORK

This section offers an overview of the Dinsdale package for the Robot Operating System (ROS), a complete framework to solve the problems presented in this paper. It gives the possibility to implement control algorithms which can equally work on both simulated and real networks of systems, without the need to modify the code. Its flexibility also allows both the controllers and the plants to be interconnected in any desired way, allowing implementation of all control structures from sec:control.

An additional characteristic is that networks of dynamical systems (controllers and simulated plants) are fully contained inside Python packages, completely independent from both Dinsdale and ROS. This allows separate development and testing, and encourages reuse of system models for any other purpose. The core of the Dinsdale framework itself is a ROS wrapper around those Python packages, which takes care of the execution, coordination, communication, and data logging, and which gives a standard interface for integration with other ROS software.

Python's popularity within the scientific community is increasing very rapidly, often at the expense of proprietary scientific computing software, making it the obvious choice for the end user side (system modelling). Python is, however, not only being used for implementation of dynamical networks, but the entire framework has been written in it. Its beauty and flexibility enabled the development of a powerful core functionality, while simultaneously keeping it relatively simple for users to understand, modify or extend.

ROS has been chosen as the underlying framework because it is rapidly becomingh the *de facto* standard in the robotics community for implementing control structures. This offers the possibility to integrate control algorithm developed in Dinsdale with other ROS packages, e.g. the ones for low-level control, localisation and mapping, or various virtual reality visualisation software. It is also

possible to use standard ROS tools (e.g. `rostopic`, `rosbag`, or `rqt`) to inspect the dynamical network, or to store and reproduce data.

The Dinsdale package is free software, with all of its components released under the GNU General Public Licence (GPL) version 3, as published by the Free Software Foundation. Dinsdale is hosted on GitHub, at `https://github.com/mross-22/dinsdale`.

A. Package structure

```
dinsdale
├─bags
├─msg
│  └─Floats.msg
├─scripts
│  ├─node_controller
│  ├─node_plant
│  ├─node_simulation_time
│  ├─node_topology_plants
│  ├─result_analysis
│  └─setup_simulation
├─src
│  └─dinsdale_system
│     ├─input_parameters
│     │  ├─A_controllers.txt
│     │  ├─A_plants.txt
│     │  ├─controllers_iterative.txt
│     │  ├─plants_topology.txt
│     │  ├─system_types.txt
│     │  ├─T.txt
│     │  └─x0.txt
│     ├─tools
│     │  ├─__init__.py
│     │  ├─read_matrix.py
│     │  └─result_analysis.py
│     ├─__init__.py
│     ├─controller.py
│     ├─plant.py
│     └─plants_topology.py
├─CMakeLists.txt
├─LICENSE
├─package.xml
├─README
└─setup.py
```

Fig. 1: Dinsdale directory tree

The Dinsdale package is a directory containing all the source code and data files required to run dynamical networks inside a ROS computational graph. The minimal structure of Dinsdale is illustrated in Fig. 1. All the source files can be divided in three categories:

- System description ⟶ `./src`
- Logged data ⟶ `./bags`
- ROS wrapper ⟶ everything else

The Python package containing the entire definition of the dynamical network that is being simulated is `./src/dinsdale_system`. Inside it, the user defines the dynamics, control laws, and topology of the network. `./src` can contain more than one system, but only the one in the `./src/dinsdale_system` package is executed. This allows to store multiple systems, and choose the one to use by simply renaming its directory.

Each time a simulation is set up, a directory containing the current date and time in its name is created inside `./bags`. In that subdirectory all the data logged during the simulation will be stored, inside ROS `.bag` files. The user does not necessarily need to access or manipulate those files, as Dinsdale has a script which allows data analysis directly in Python.

Fig. 2 gives a somewhat simplified overview of the Dinsdale architecture, which is the topic of the rest of this chapter.

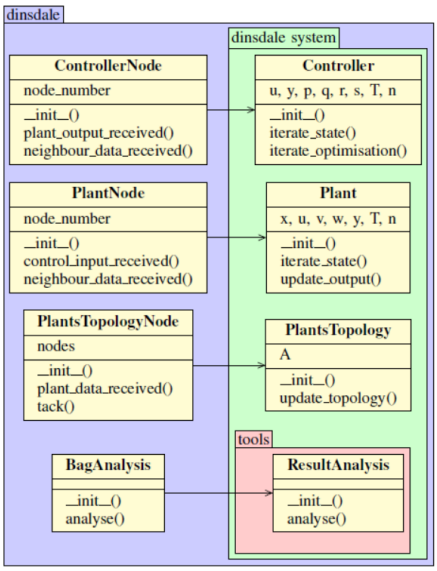

Fig. 2: Dinsdale directory tree

B. System definition

The dynamical network is defined in a Python package called `dinsdale_system`, located inside the `src` directory of the main Dinsdale package (see Fig. 1).

1) Plants: Each plant in the network is defined by a `Plant` class. This class is located inside `plant.py`, and its structure is shown in Fig. 2. It is necessary to define as many classes, each in its own file, as the number of different types of plants in the network.

Table I explains what the attributes of the class are. All, except for `n` and `T`, are of the `numpy.matrix` type. They need always to have the shape of column vectors, but their length is not limited.

As shown in Fig. 2, the `Plant` class has three methods:

- `__init__(n, x0, T)` – The initialisation of a `Plant` instance. It initialises all the attributes, and executes the initialisation code set by the user. This method is called only once, at the beginning of the execution.
- `iterate_state()` – The user here sets the equations for the update of the plant's states `x` and output

TABLE I: Plant attributes

Plant	
x	plant states
u	input from controller
y	output for controller
w	input from other plants
v	output for other plants
n	ordinal number of the node
T	sample time

for other plants `v`. This method is executed when new inputs from the controller are received.

- `update_output()` – This method is executed when the new set of input from neighbouring plants `w` is received, and its purpose is to contain the equation for updating the output for the controller `y`.

2) Controllers: Each controller in the network is defined by a `Controller` class. Following the same logic applied to plants, the class for a controller is inside a `controller.py` file. In case of multiple different types of controllers, there will be multiple files, called `controller.py`, `controller_1.py`, `controller_2.py`, etc.

TABLE II: Controller attributes

Controller	
y	input from plant
u	output for plant
q	input from other controllers
p	output for other controllers
s	iterative input from other controllers
r	iterative output for other controllers
finished	iterative communication finished
n	ordinal number of the node
T	sample time

Fig. II explains what the attributes of the `Controller` class are. Except for `n`, `T`, and `finished`, the rest are of the `numpy.matrix` type. They need always to have the shape of column vectors, but their length is not limited.

As shown in Fig. 2, the `Controller` class has three methods:

- `__init__(n, T)` – The initialisation of a `Controller` instance. It initialises all the attributes, and executes the initialisation code set by the user. This method is called only once, at the beginning of the execution.
- `iterate_state()` – The user here sets the equations for the update of the controller's outputs, both for the plant and for neighbouring controllers (`u` and `p`). This method is executed when new data from the plant (`y`) is received.
- `iterate_optimisation()` – This method is executed after `iterate_state()` if the controllers communicate iteratively within one simulation step. This method is usually used when the control law is computed by collaboratively solving an optimisation problem on-line. If this is the case, the vectors used for iterative communication are `r` and `s`. In each simulation step this method is called until `finished` is set to `True`. In the next iteration it will automatically be reset to `False`.

3) Plants interaction topology: The topology of the plants interaction network can be static or dynamic. When it is static, nothing has to be done. In case it is dynamic, the user has to determine the law according to which the topology is changing. This is done in the `PlantsTopology` class (shown in Fig. 2), inside the `plants_topology.py` file (note that there can be only one such file for a network).

The class `PlantsTopology` initialises:

- `A` – The adjacency matrix of the plants interaction network (a square $(n \times n)$ matrix, where n is the number of plants).
- `nodes` – The number of plants.
- `w` – The output from each plant (a list of n elements, each being a one dimensional `numpy.array`).
- `communication` – A list of neighbours of each plant (a list of n lists). It gets populated during the initilaisation by looping through the adjacency matrix. Once the neighbours of each plant are found, in each simulation step the plant will receive their outputs.

The method `update_topology()` is empty by default, and this is where the user can define the laws according to which the topology is changing, in case it is dynamic. The laws could use the data sent by plants, or be functions of time. Another possible application of the `update_topology()` method, apart from having a dynamic topology, is multiplying the values exchanged by plants with some weights, which can also change over time. Whatever its usage may be, the objective in this method is to update the list `communication` with lists of neighbours of each plant in every time step.

4) Parameters: Inside `./input_parameters` there are seven items to be set to fully define the dynamical network:

- `A_controllers.txt` – The adjacency matrix of the controller communication network. The file can be empty if the controllers do not communicate.
- `A_plants.txt` – The adjacency matrix of the plant communication network. Even if the plants do not interact, it has to be of size $(n \times n)$, where n is the number of plants, but filled with zeros.
- `controllers_iterative.txt` – Determines whether the controllers communicate iteratively inside one time step (1 if they do, 0 or empty otherwise).
- `plants_topology.txt` – 0 if the plants topology is static, 1 if dynamic.
- `system_types.txt` – Describes the type of controller and plant for each subsystem. It is a matrix of size $(n \times 2)$, where n is the number of subsystems. The first column corresponds to controllers, the second to plants.
- `T.txt` – A (3×1) vector, the elements of which correspond to the sampling period (used for discretisation of the system dynamics), the real duration of a simulation time step, and the final time of the simulation, all given in seconds.
- `x0.txt` – Contains the initial conditions for each plant. It is an $(n \times m)$ matrix, where n is the number of plants and m the number of states of a plant (in case of heterogeneous systems, the plant with the largest dimension of the state space determines this number).

5) Tools: The additional package `./tools` contains useful tools which are not part of the dynamical network. Two modules can be find inside it:

- `read_matrix.py` – This is the function all other modules use for reading matrices from files. It is placed here to make it available also for the user, in case he or she wants to load some additional matrices.
- `result_analysis.py` – This module is where all data from a simulation is made available to the user for analysis of any kind.

The `result_analysis.py` file contains the `ResultAnalysis` class, shown in Fig. 2. In its initialisation method, the class initialises a dictionary called `data`. This dictionary gets filled by a core Dinsdale class with data stored during a simulation. The keys of the dictionary correspond to names of controller and plant attributes. The value of a key is a list of a list of all values of the specific attribute of every node in the network. In the method `analyse()` the user is free to manipulate this data in any way, e.g. plotting it.

C. Runtime

The `dinsdale_system` package is a pure Python package, which contains the description of a dynamic network. To generate as many instances of its classes as required by the network specification, initialise their values, transform them in ROS nodes, set up the communication networks, and coordinate the execution, there is a wrapper around it. This wrapper is the core Dinsdale functionality. Fig. 2 shows how every class in `dinsdale_system` is being use by Dinsdale.

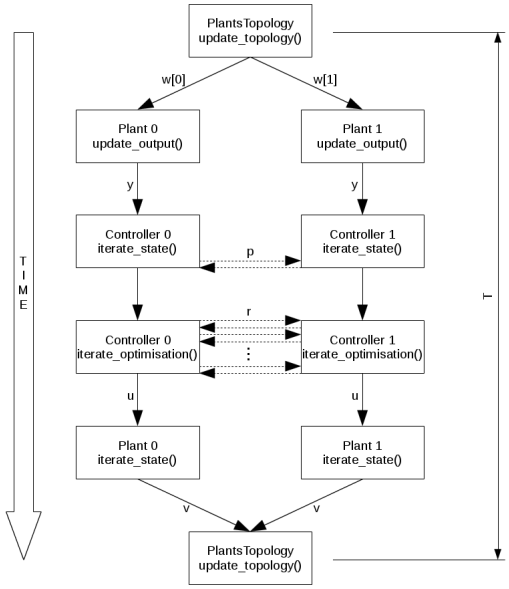

Fig. 3: Execution flow

At runtime, Dinsdale sets up as many ROS nodes as defined by the user, and connects them accordingly in a ROS computational graph. After all nodes are set up in a ROS graph and the simulation time is started, Dinsdale starts simulating the network using the classes defined in `dinsdale_system`. Fig. 3 shows the order in which the methods are being called inside one time step, when simulating a network of two systems. The dashed arrows represent optional communication between controllers (single and iterative). When the controllers are not communicating iteratively, the method `Controller.iterate_optimisation()` is not called at all. The letters near the arrows represent the name of the attribute that is being sent.

D. Replacing simulated plants with real systems

One of the main motivations for the development of the Dinsdale package was to make the transition from simulation to implementation as simple as possible. This was kept in mind during the entire design process, which can be seen for example in the structure of controller nodes, which have all their input and output topics set within their namespaces. This makes them agnostic about who is publishing their inputs and who is using their outputs. Not only this gives the possibility to replace the simulated plants with real systems, but also to use Dinsdale together with the big variety of existing ROS packages, e.g. for simultaneous localisation and mapping, visualisation, image processing, etc.

As ROS is gaining popularity, more and more hardware drivers are available as ROS packages. At the same time, small single-board ARM computers are rapidly being adopted as cheap and efficient control units for a huge variety of systems. Since those computers are able to run GNU/Linux distributions, and therefore ROS, these facts makes it very convenient to use Dinsdale generated controllers for control of real networks of systems.

VI. CONCLUSION AND FUTURE DEVELOPMENTS

This paper presented the motivation and explored the possibilities of creating a universal framework for development of networks of dynamical systems. It introduced a new framework, which fully satisfies the requirements, derived by the needs of today's research community. With the testing conducted so far, it has been shown that the Dinsdale framework offers a solid and reliable infrastructure for research and development of dynamical networks. Although the benefits from using such a framework are theoretically clear, they still have to be proven in practice. Feedback from other research groups would be immensely valuable for determining the direction of the future development of this software.

Generally speaking, the need for such framework exists, and considering its foundations are the increasingly popular ROS and Python, its adoption could be very convenient. It will certainly remain available and continue being maintained and developed, at least in the foreseeable future.

Dinsdale is hosted on GitHub, at `https://github.com/mross-22/dinsdale`.

REFERENCES

[1] R. Murray, K. Astrom, S. Boyd, R. Brockett, and G. Stein, "Future directions in control in an information-rich world," *Control Systems, IEEE*, vol. 23, no. 2, pp. 20–33, Apr 2003.

[2] A. Preumont, *Vibration Control of Active Structures: An Introduction*, ser. Solid Mechanics and Its Applications. Springer, 2011. [Online]. Available: http://books.google.hr/books?id=MUQUQyB4bEUC

[3] P. Koumoutsakos and I. Mezic, *Control of Fluid Flow*, ser. Lecture Notes in Control and Information Sciences. Springer, 2006. [Online]. Available: http://books.google.hr/books?id=7T2lrA-YVCwC

[4] R. Duffner, *The Adaptive Optics Revolution: A History*. University of New Mexico Press, 2009. [Online]. Available: http://books.google.hr/books?id=6nJ9PQAACAAJ

[5] J. Ekanayake, N. Jenkins, K. Liyanage, J. Wu, and A. Yokoyama, *Smart Grid: Technology and Applications*. Wiley, 2012. [Online]. Available: http://books.google.hr/books?id=AmpSxODiF7sC

[6] H. S. Witsenhausen, "A counterexample in stochastic optimum control," *SIAM Journal on Control*, vol. 6, no. 1, pp. 131–147, 1968. [Online]. Available: http://scitation.aip.org/getabs/servlet/GetabsServlet?prog=normal\&id=SJCODC000006000001000131000001\&idtype=cvips\&gifs=yes

[7] R. Olfati-Saber, "Flocking for multi-agent dynamic systems: Algorithms and theory," *IEEE Transactions on Automatic Control*, vol. 51, pp. 401–420, 2006.

[8] M. Mesbahi and M. Egerstedt, *Graph Theoretic Methods in Multiagent Networks*, ser. Princeton Series in Applied Mathematics. Princeton University Press, 2010.

[9] F. Bullo, J. Cortés, and S. Martínez, *Distributed Control of Robotic Networks*, ser. Applied Mathematics Series. Princeton University Press, 2009, `http://coordinationbook.info`.

[10] G. Antonelli, "Interconnected dynamic systems: An overview on distributed control," *Control Systems, IEEE*, vol. 33, no. 1, pp. 76–88, 2013.

[11] M. Rotkowitz and S. Lall, "A characterization of convex problems in decentralized control," *Automatic Control, IEEE Transactions on*, vol. 51, no. 2, pp. 274–286, Feb 2006.

[12] S. Jiang, P. Voulgaris, L. Holloway, and L. Thompson, "Distributed control of large segmented telescopes," in *American Control Conference, 2006*, June 2006, pp. 6 pp.–.

[13] M. Rossi, "ROS u distribuiranom upravljanju dinamičkim sustavima," University of Zagreb, Tech. Rep., 2014.

[14] ROS, "Documentation," `http://wiki.ros.org/`.

[15] J. M. O'Kane, *A Gentle Introduction to ROS*. Independently published, Oct. 2013, available at `http://www.cse.sc.edu/ jokane/agitr/`.

[16] ROS, "About ROS," `http://www.ros.org/about-ros/`.

[17] Willow Garage, "About Us," `http://www.willowgarage.com/pages/about-us`.

[18] OSRF, "ROS @ OSRF," `http://osrfoundation.org/blog/ros-at-osrf.html`.