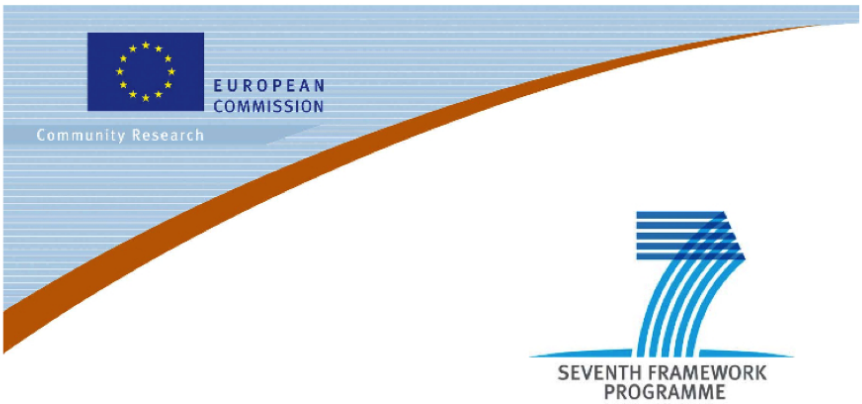


Private Public Partnership Project (PPP)

Large-scale Integrated Project (IP)

D1.1 Technical Specs. & Technology Evaluation

Project full title: Future Internet Core
Project acronym: FI-Core
Contract No.: 632893
description: Detailed technical specifications for o2S2PARC based on user requirements (e.g. from questonnaires) and technology evaluation.

Contents

Executive Summary

This document is related to Deliverable D1.1 of the IT'IS SIM-CORE proposal: "D1.1 -Detailed technical specifications for o2S2PARC, based on user requirements (e.g., from questionnaires) and technology evaluation". The requirements for the SIM-CORE platform (o2S2PARC) were established through two activities: 1) contacting SPARC teams to establish their modeling needs and 2) evaluating the relevant software technologies necessary for the effective implementation of the SIM-CORE, mostly by creating feasibility prototypes. The related Milestones are M1.1, M1.2, and M1.3. These activities (methodology, results, conclusions) have now been completed and are summarized in this document. From this, high level platform functionality specifications and concrete framework architecture specifications are derived.

The requirements are very much aligned with prior expectations and only minor adaptation of the proposal and possibly a change in prioritization is suggested, as elaborated in this document. The technology evaluation has resulted in a clear idea of the technologies and approach that should be applied to implement the SPARC SIM-CORE platform. The design of the platform will enable maximal flexibility with regard to the wide variety of existing and envisioned user-generated modeling services. It will also offer user-friendly interfaces, allowing users to engage on different levels with the platform, ranging from simple execution of existing models to the advanced generation of services with fine-grained control over dedicated user-interface elements, depending on expertise. Due to the modularity of the chosen approach, it will be simple to extend and adapt the platform at later time-points, and it will be feasible to revisit some of the choices as technology evolves and to replace layers or components of the implementation without the need for a complete redesign of the platform.

Introduction

This document is related to Deliverable D1.1 of the IT'IS SIM-CORE proposal: "D1.1 -Detailed technical specifications for o2S2PARC, based on user requirements (e.g., from questionnaires) and technology evaluation". The requirements for the SIM-CORE platform were established through two activities: 1) contacting SPARC teams to establish their modeling needs and 2) evaluating the relevant software technologies necessary for the effective implementation of the SIM-CORE, mostly by creating feasibility prototypes. The related Milestones are M1.1, M1.2, and M1.3. These activities have now been completed and the results are summarized in this document. After presenting the requirement gathering approach and effort, the obtained information is summarized, and conclusions are drawn. The technology evaluation approach and activities are introduced, results are presented, and conclusions are reached. From this, high level platform functionality specifications and framework architecture specifications are derived.

The SPARC awardee requirement collection was discussed during the DRC meeting (NIH Directorate, Bethesda, 18-19 October 2017) with the SPARC management and the DRC CORE teams. Based on that discussion, the following approach was chosen: i) an open-ended questionnaire that will guide the modeling-needs requirement collection, with a focus on aspects pertaining to the SIM-CORE, is compiled; ii), that questionnaire is shared with other COREs to collect input, clarify potential overlaps, and for further refinement ; iii) interviews are subsequently scheduled with SPARC awardee teams, in which the questionnaire serves as guide for the requirement collection (however, the discussion is driven by the specific needs of the awardee teams, which can require increased discussion depth in some areas, while other questionnaire topics might not be applicable); and iv) based on the interviews, additional information (publications, models) is requested from the awardee teams.

Categories of topics considered within the questionnaire include: a) data generation; b) general modeling; c) modeling organ anatomy, nerve trajectories, connectomes; d) modelling and simulating electrophysiology;

e) coupling electrophysiology and multiphysics simulations; f) online platform: user perspective; g) postprocessing: visualization and analysis; IT resources, contribution to development; h) project-specific issues; i) miscellaneous.

Extensive interviews have taken place with groups specifically pointed out by the SPARC management as being particularly large efforts with an already significantly present modeling component (Howard/colon (Joel Bornstein, Melbourne); Shiv/cardiac (Kember Nova Scotia/Clancy UCI); Powley/stomach (Matt Ward, Indiana/Purdue); Bolser/lung (Morris?, UF); Grill/vagus (Duke); Jenkins/IR (CWRU)). Those interviews typically lasted two hours and occasionally were followed by further discussions. Contact with other teams and related requirement collection has taken place in the context of the SPARC DRC Launch and PI meeting (16-17 November 2017), as a result of direct referral by the SPARC management to the SIM-CORE of teams planning modeling related submissions, and by IT'IS initiative. These meetings were typically shorter and did not go through the full depth of the questionnaire topics. During the interviews, at least two experts from IT'IS were present. Following the interviews, the experts always had a follow-up discussion to identify potential areas, in which the current SIM-CORE vision might require adaptation.

Team PI	Location and Date
Osborne	Washington (DRC meeting), 18/10/2017
Clancy, et al.	Webex and Phone 11/2017
Bornstein	Webex 8/11/2017
Powley, et al.	Webex 22/12/2017
Howard	Washington (SfN) 10/12/2017
Grill, et al.	Washington (DRC meeting)
Horn, et al.	Washington 16/11/2017, Webex 18/01/2018, Webex 24/01/2018, repeated mail exchanges
Jenkins, et al.	Webex 17/1/2018
Kember (Ardell group)	Skype 14/12/2017
People at DRC PI meeting	15,16/11/2017
Lazzi	Webex 14/12/2017
Yoshida	Phone 1/11/2017
Pitts	Skype 12/12/2017

- **Computational resources** : All teams stated that initial computational resource needs will be moderate, with existing SPARC related models currently running on simple desktop machines. Typically, only few people will initially perform modeling and not many simulations will have to run in parallel. However, in the longer term, parameter sweeps, optimizations (primarily of stimulation parameters and configurations), simulations with large number of neurons/cells, and simulations covering large (simulated) time periods are planned. For that purpose, support for high performance computing resources has been requested, with the NSGportal (https://www.nsgportal.org//) mentioned specifically as a potential resource worth supporting. One team will require supercomputing functionality that is beyond what SPARC can provide. Teams foresee increasing over time the number of people working in modeling related areas and hence expect increasing computational burden. However, no concrete estimates of required computational resources in the longer term could be obtained. Being able to run simulations in the cloud is seen as valuable and by some group as necessity.
- **Simulation implementation** : Current simulators are implemented as C, C++, Java, Matlab, Python, or Excel codes. In some cases, workflows frameworks (such as Kepler) are used to compile code, manage data, and post-process results and similar workflow support has been requested. Linux is the most common environment used by groups that implement their own compiled codes.
- **Stimulation physics** : Electromagnetic and acoustic exposure, thermal, and light propagation modeling have been requested, with electromagnetic modeling most prominently demanded, followed by acoustic exposure. The other two physics were requested by single teams. Support for thin layers and anisotropy in the electromagnetic modeling is seen as a must. For thermal modeling, consideration of perfusion and thermoregulation effects is fundamental. Stimulation can be local or remote, and thus exposure and propagation needs to be modeled on macro- and micro-anatomical levels. A few teams consider adding own biomechanical models in the future.
- **Physiological modeling** : Physiological modeling will include body physiology, organ physiology, and peripheral nervous system (PNS) electrophysiology models. In addition, modeling of tissue evolution (damage, interface effects) is planned. Organ models are most frequently realized as coupled ordinary differential equations (ODEs) of actors (cells, neurons…), but can also be finite element-type partial differential equation (PDE) models or even black-box models without known equations (e.g., extracted through machine learning). PNS models include compartmental/cable-equation-type models, connection strength models (stimulation/inhibition), and population activity models, with activity levels, firing rates, or transient action potentials as primary coupling quantities. Other coupling quantities include measures of physiological activity and field distributions. Support for different types of myelinated and unmyelinated fibers is requested. Both, simplified and morphologically detailed neurons must be supported. NEURON developed by Yale University is the only software commonly used by the contacted teams for nerve electrophysiological modelling. Other software packages, such as NetPy, are used by single groups. Support for coupling NEURON models with spatio-temporally varying electromagnetic, acoustic, and thermal exposure has been requested. Computing compound action potentials would help with validation against experimental data. The MAP-CORE is also heavily advocating support for CellML/SBML-based solvers.
- **Species** : Contacted teams are interested in modeling involving anatomy/physiology from humans, rats, mice, monkeys, guinea pigs, cats, rabbits, sheep, and donkeys. An additional poll was launched on the SPARC SLACK communication platform. The highest interest is clearly in human, rat, and mice models.
- **Anatomical models** : Geometric anatomical models on the organ as well as the nerve microstructure scale have been requested. Parameterization of anatomical shape would be useful to selected teams. The possibility of morphing the anatomical geometries to produce variability and to account, e.g., for organ motility, has been requested. Integrating computational models and measured/simulated data in reference anatomical models is seen as sensible and helpful. Multiple teams expressed interest in a detailed spinal cord anatomical model.
- **Simulation coupling** : All of the described workflows required/used by the contacted teams can be implemented as pipelined workflows. Most, but not all, computational models benefit from convenient separation of time scales between PNS and the different organ physiological processes, which permits iterative solving rather than tightly coupled solvers. When bidirectional or closed loop coupling is employed in the models, it is always implemented in the form of coupled ODE solvers running in a single service (for efficiency reasons) and hence can be handled as part of a pipeline architecture.
- **Visualization** : The typical type of viewers (2D plots, slice field viewers, surface viewers, 3D rendering…) have been requested. The possibility for users to create their own advanced visualization modules has not been requested.
- **GUI** : There is an important demand for functionality that allows users to enhance their services/models with a user-friendly interface, without advanced coding knowledge. The requested interactivity is limited to standard functionalities, such as parameter specification, message display, searching, visualization, process

submission and monitoring. No group currently sees a need to be able to implement own specialized forms of GUI interactivity (e.g., an interactive 3D picker). A scripting interface complementing the GUI would be appreciated by some of the groups. There is unforeseen demand for a flowchart-like editor for graphically generating, displaying, and editing coupled model components (e.g., neural networks, physiological influences).

- **Sharing** : All groups expressed willingness to share their models. In some cases, it has been requested that models be kept private while awaiting publication of results or while going through (e.g., internal) approval processes. In other cases, it has been requested that models can be shared without need to divulge the underlying implementation, or that sharing can be limited to research purposes. No request for the ability to charge for model use has been made. However, the possibility of licensing solvers developed by SPARC teams prior to the SPARC program for use on the platform has been brought up. Collaborative model development between groups involved in the same organ system has been mentioned as desirable.
- **Image processing** : Segmentation support has been requested and multiple teams would be interested in using microscopy images to generate nerve microstructure models. Image processing is used by the SPARC awardees for various tasks (video analysis, calcium expression, neuron morphology reconstruction…), but for those tasks the teams will continue using their established tools (ImageJ, Imaris, Neurolucida…). One team expressed willingness to share self-developed video analysis functionality in the form of a SIM-CORE service, but it is unclear if other parties would be interested in such functionality. Only one team expressed interest in using a potential SIM-CORE nerve-tracing tool in the future.
- **Meta-modeling** : There is a high demand for meta-modeling functionality. This includes, parameter studies, parameter tuning, optimization, uncertainty assessment, and uncertainty propagation. Multiple teams expressed interest in control functionality (e.g., closed- loop control) and model order reduction functionality.
- **Quality assurance** : Functionality that allows to reproduce previously performed studies is judged important, as well as ways of assessing the degree of model validation.
- **Support** : Most teams have requested extensive support with model development ("What should we measure to create a model?"), in moving existing models to the platform, and with platform use (training events, contactable support team).
- **File formats** : File format standards exist for electrophysiological measurements and image data. Results are frequently stored in ascii files, Matlab, or Excel form. Other than that, teams did not express fixation on supporting specific file formats and seem to be flexible on adapting their models as required. Some of the teams pointed out that communities, such as the human brain project, have established standards for specific purposes.
- **Maintainability** : Long term sustainability of the platform is seen as major argument for moving models onto the SPARC platform. Teams were, however, not willing to concretize if in the long term they would be willing to pay for such a service. They do see the platform as highly valuable beyond SPARC.
- **Platform development** : While most teams are interested in creating and offering their own services, contributing to the open source development of the actual platform is generally only seen as something that could be of interest in student projects. Making use of a scripting interface is deemed as of potentially high interest by contacted parties, but not concretely planned.
- **Analysis** : Availability of analysis functionality (e.g., Python- based analysis scripts) is requested. Services that run Matlab or similar tools for analysis purposes are requested by most teams. Other softwares that are currently used for analysis purposes include R, Statistica, SPSS, and graphpad. Data transfer between tools is frequently a significant effort and time consuming and supporting analysis pipelines within the online platform would be valued.
- **Privacy** : The contacted teams do not foresee privacy issues related to the data or images that they require in the context of computational modeling.

The willingness to participate in the requirement gathering interviews was high. There is important enthusiasm about both, the possibility of sharing own models in a friendly way, without having to invest a lot in developing user-interface functionality, and the potential of getting access to powerful solvers, meta-modeling functionality, electrophysiology models, and analysis tools. Quality assurance aspects, data and model integration, user-friendliness, computational infrastructure, anatomical models, and support for workflows are other highly valued components. The platform concept, as developed throughout the SIM-CORE application process appears to be very well suited to the requirements, with few extensions being required (support for "supercomputing", workflows (s. side-car concept below), nerve microstructure modeling, support for thermal, biomechanical, and light propagation modeling (IT'IS does not have solvers for the latter two), machine learning, additional animal anatomical models, as well as a detailed spinal cord model, modeling of fields generated by neural activity for sensing, and further meta-modeling (control, model order reduction)). Certain aspects do not seem to currently be of high priority (scripting interface, image-processing beyond segmentation xxx). Providing users with proper support will be fundamental to assure the widespread adoption of the platform.

To decide on the proper methodology for the SIM-CORE platform, a technological evaluation period was fixed, during which different technological alternatives were explored by prototyping components of the foreseen platform (frequently resulting in demonstrators). In the process, advantages and disadvantages of those alternatives were established and the collected experience was used to decide on the structure and suitable technology to be used for the implementation of the SIM-CORE platform. This forms the basis for the related part of the specifications.

The results on the **technology evaluation** starts with an overview of the design of the framework in order to define every logical part/concept of the system addressed in this review. It follows with a brief description of methodology adopted the review process. Then, every logical part/concept of the system is addressed separately:

- Front-end : A review on client-side frameworks.
- Web-server : A review on server-side frameworks.
- Communication models : A review on intra- service communication models.
- Computational services : A review on model for a sustainable computational framework.

To conclude with some demonstrations and recommendations:

- Demonstrations : Frameworks in action
- Conclusions : Wrap up and recommendations

Pre-selection of technologies

The technology evaluation started with a pre-selection of technologies (i.e. libraries, framework, toolkits, models, …) for the different components of the SIM-CORE platform, namely the front-end (client-side), the web back-end (server-side), the communication models and the computational services.

The technology evaluation was mostly conducted by four experienced senior developers, with additional input from open-source, full-stack development, software quality management, and software (development) management experts.

The criteria used for the pre-selection consisted mainly of three factors:

- **Completeness**: the selected framework must include functions or accept extensions that support features, which are key for its purpose. For instance, a suitable web-framework must include features for routing, database models, web-security, etc., which cover the essential functionality expected for the server-side.
- **Sustainability**: the selected technology should be long-term maintainable as an open-source project. Key measures of sustainability considered here are its popularity, the support of the developers community (i.e., documentation, fixes, releases, …), and the license terms.
- **Productivity**: we believe that high productivity can be achieved with the right combination of a suitable framework (e.g. complete, easy- to-use, and sustainable) and the effectiveness of reusing existing functionality and previous experience of the SIM-CORE team with specific technologies. For that reason, the pre-selection should incorporate options that combine both novel as well as already established programming paradigms (e.g. languages, design-patterns, or even specific frameworks).

Review process

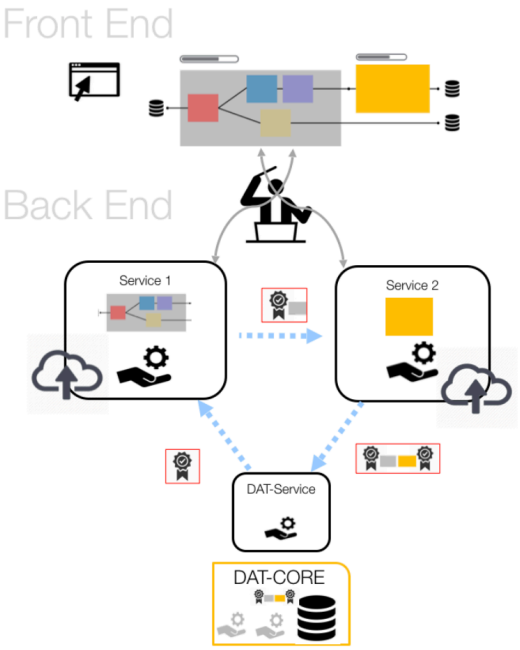
After the pre-selection of technologies, the review adopted a more practical approach. Every group of technologies was tested within a given context defined in a user story. Based on this approach, the evaluation gradually transitioned from simpler to more realistic scenarios, resulting in a collection of demos that are published in the project repository. Among all demos, there are three prototypes that combine different technologies within the full-stack and cover realistic scenarios expected in the final platform. The specifics of each review are documented in place, next to its correspond demo. Nonetheless, for the sake of clarity, in this document we provide a summarized version of each review and the final recommendations.

Design Concepts

SIM-CORE is designed as a web simulation platform that will allow users to perform complex numerical simulations combining computational services developed by the scientific community. It shall work in coordination with its sibling platforms MAP and DAT-COREs that will host scientific models, data and services built, shared and validated by the community.

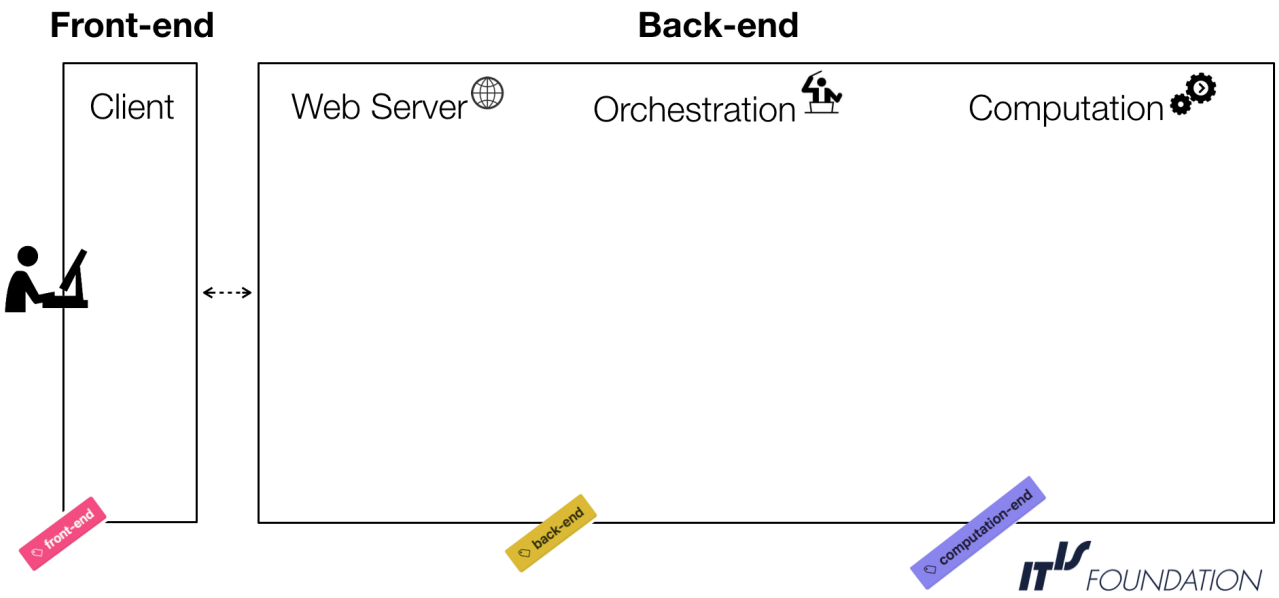
User Story

In order to help understanding the context and functionality intended for the framework, a standard workflow expected from a SIM-CORE user is sketched in the following figure:



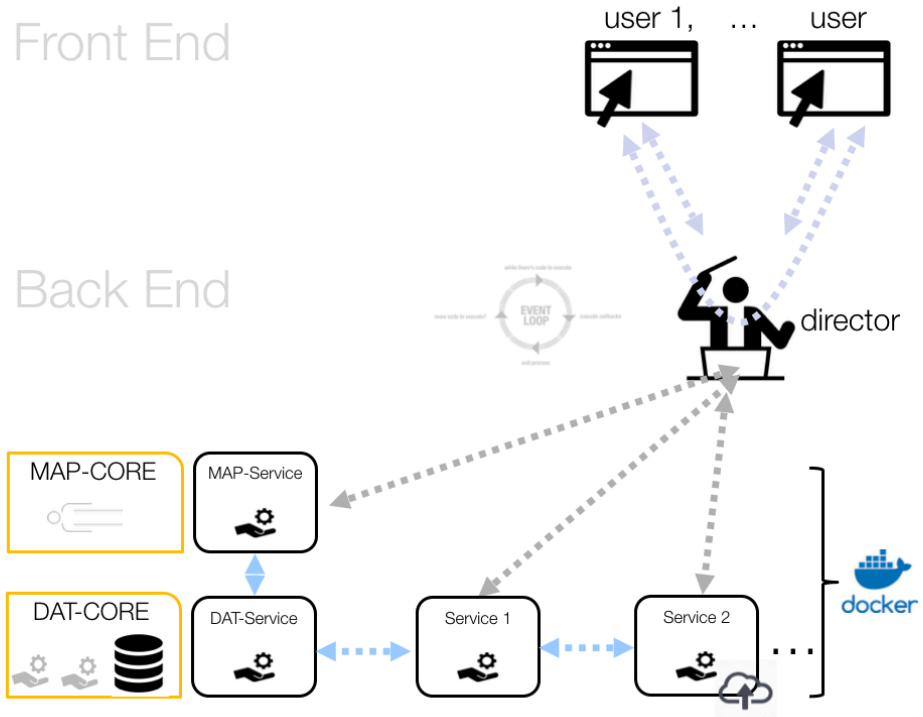
The user access SIM-CORE platform using a web interface. The platform provides access to a curated repository of scientific models and data stored in MAP and DAT-COREs. Using SIM-CORE, the user can create a new workflow by interconnecting each model or dataset. Each of these building blocks are treated as isolated services that are orchestrated and executed by the framework. The entire execution process is monitored and the chain of custody preserved. If the user decides to share this new model or results with the scientific community, the framework shall enable this giving access to MAP and DAT-CORE services.

Architecture



The framework architecture is divided in four major building blocks. The client-side entirely fills the front-end while the back-end is subdivided into the web server, the orchestration and the computational modules. This subdivision is used as a base to organize the review of available technologies needed to build a reliable and sustainable framework.

Workflow



This is an schematic view of the framework in action. The system is based on the orchestration of contained services that can perform computations or access/interaction to other CORES.

Users obtain information about available and applicable services via the web server, who queries each service's properties (what does it do, what does it require, latency…) stored in MAP/DAT-CORE. The user configures

what and how to use these services by creating a work-flow (pipeline) with them in the UI. Services are packaged as container images, which become standard de facto units for development, shipment, and deployment of software tools.

The users schedules some pipelines to run using the web interface. The framework, at the back-end, coordinates a number of running services, including those with access to MAP and DAT-COREs. The orchestration module is responsible for scheduling, scaling, monitoring, and running services. Services are the execution units offered by the framework. Available services are stored in DAT-CORE.

Frameworks

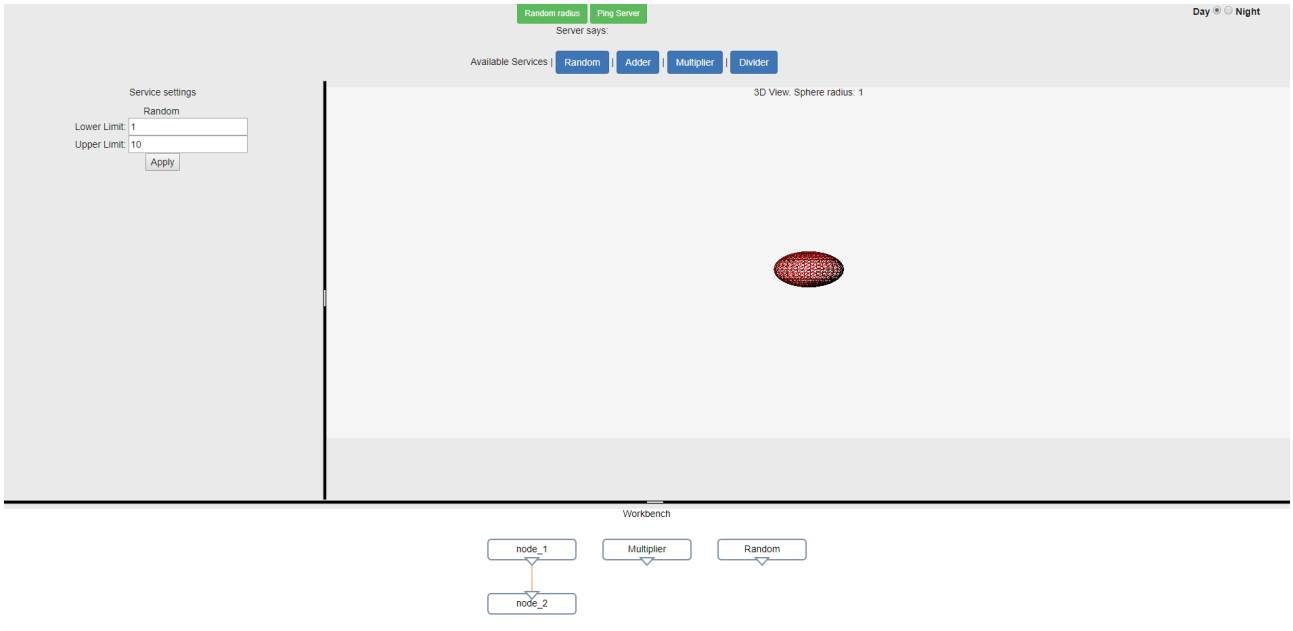
The front-end part corresponds to the software running on the client side. The server-side transmits the program/instructions to the web- browser on the client-side, which interprets and executes them. The front-end interacts directly with the user and for that reason one of main features of front-end frameworks is the Graphical User Interface (GUI). Among the wide variety of frameworks that support the development of front-ends we pre-selected: vue, react and qooxdoo.

In order to compare advantages and disadvantages of the different frameworks, prototype Single-Page-Applications were built for each framework with similar User Interfaces and features.

The following components/aspects were reviewed:

- **Interactive layout**: The layout consists of at least 4 main components: Available services, Settings viewer, 3D renderer and Workbench. The user should be able to interact with the sizing/position of those components. A Results viewer was also implemented for the React and qooxdoo prototypes.
- **3D renderer**: The 3D viewer shows an interactive object. three.js, the most popular JavaScript library that uses WebGL, was used for all four prototypes.
- **Workbench**: The workbench shows how the different computational services are connected to form a pipeline.
- **Data binding in UI**: All the information used in the frontend is contained in a JSON object. This data needs to be converted to something meaningful for the user so that the complexity behind it turns into buttons, text/number inputs...
- **Dynamic styling**: By clicking a checkbox or dropdown menu, the GUI style should be switchable.
- **Front-end/Back-end communication**: Some logic is implemented on the client-side, but the heaviest logic stays on the server side. Thus, the client needs to communicate with the server. E.g., when the web application is started, it needs to ask the server what are the available services. For vue, react and qooxdoo , the web socket socket.io module was used.

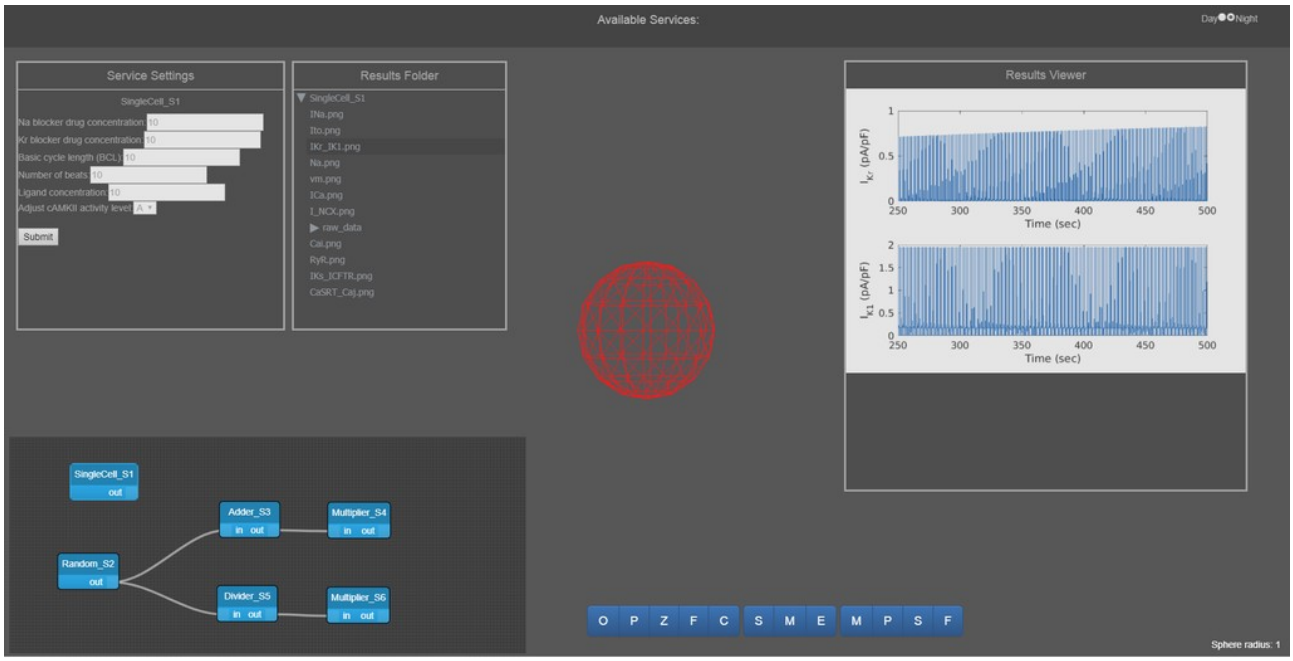
vue



- **Interactive layout**
  - Not very fancy packages found for creating dynamic layouts
  - Many vue dedicated layouts not in a mature status
  - Better to use HTML-CSS basic solutions
  - vue-splitpane module used

- **3D renderer**
  - three.js library used
  - Flexible and easy to use
- **Workbench**
  - Very poor resources found to build flowcharts
  - vue-port-graph module used
- **Data binding in UI**
  - The declarative UI makes it easy to mix HTML code with JavaScript based logic coding
  - Built-in event bus (publish-subscribe) pattern communication
  - Very flexible and easy to use notification system
- **Dynamic styling**
  - Easy to define styles shared between components
- **Front-end/Back-end communication**
  - vue-socket.io module used for communication
  - Easy to set up in both server and client sides
  - Publish-subscribe pattern communication
- **Extra impressions**
  - Kind of a mixture of React.js and Angular.js
  - Templating, scripting and styling well separated
  - Lot of templating logic goes into html side code
  - Lightweight framework that needs extra third party packages/modules to build components
  - It is growing rapidly, regularly introducing cool new features
  - Very active community behind
  - Easy learning curve

react



- **Interactive layout**
  - Many packages provide very good looking interactive layouts
  - react-rnd module used
- **3D renderer**
  - react-three and react-three-renderer are the modules that go on top of three.js

- react-three-renderer (not all three.js features are implemented in this wrapper)

- **Workbench**
  - Very nice packages found to represent a flowchart
  - nodes and links can be extended to fit our needs
  - storm-react-diagrams module used

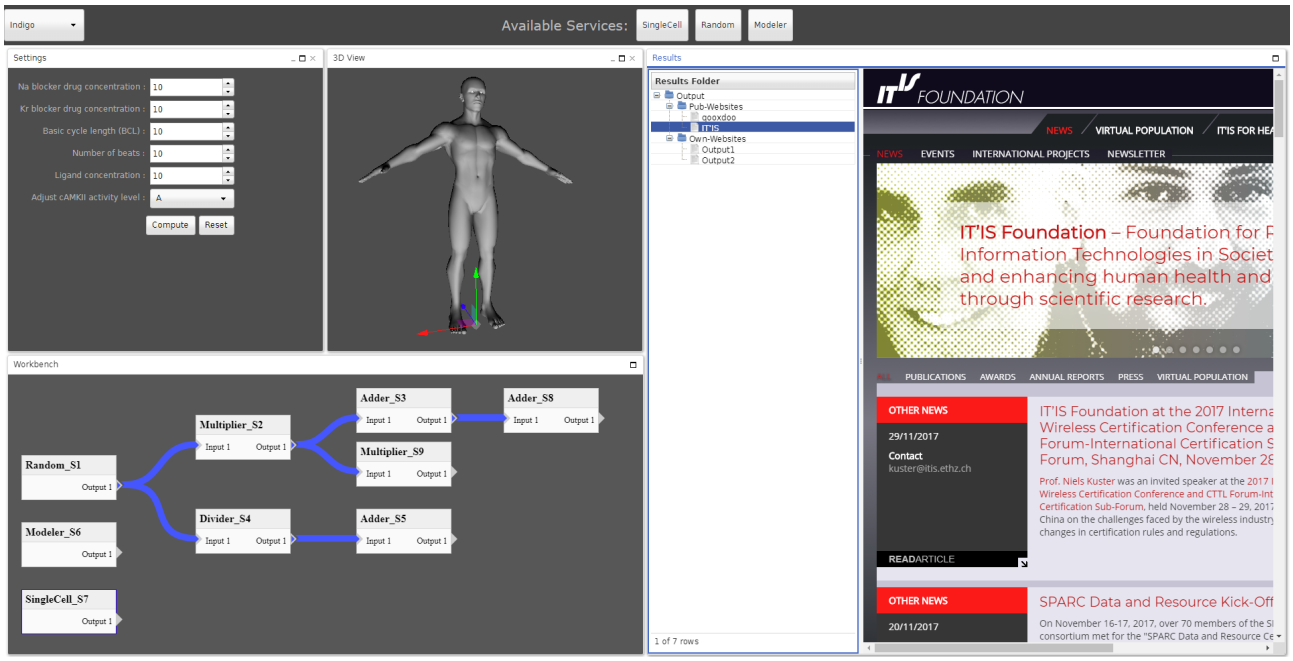
- **Data binding in UI**
  - redux is the most popular module for component communication
  - Based on a store, actions and reducers: mutates the states of the components providing interactivity

- **Dynamic styling**
  - Easy to define styles shared between components

- **Front-end/Back-end communication**
  - socket.io-client together with socket.io modules used for communication
  - Easy to set up in both server and client sides
  - Publish-subscribe pattern communication

- **Extra impressions**
  - HTML, scripting and styling well separated
  - Not easy to mix scripting with templating
  - Lot of third party packages to choose to implement new features
  - Very active community behind and many examples already in place
  - Linear learning curve

qooxdoo



- **Interactive layout**
  - qx.ui.window.Window used to make it look like a Desktop application

- **3D renderer**
  - Using three.js directly
  - three.js related OrbitControls.js and ShaderSkin.js were also used for controlling the camera and adding texture to the head model respectively

- **Workbench**
  - jquery-flowchart.js used

- nodes and links can be extended to fit our needs

- **Data binding in UI**
  - 'json object' -> 'model' qooxdoo built-in conversion used
  - Similar to redux
- **Dynamic styling**
  - Includes different themes that can be extended/customized
  - Well separated from the logic, even though can be part of it
- **Front-end/Back-end communication**
  - socket.io module used
- **Extra impressions**
  - qooxdoo does not have a package manager (yet). Therefore, all 3rd party libraries used need to be manually integrated. This has pros and cons, e.g. an advantage is that dependencies are more under control.
  - Since the community is smaller, fewer pre-built wrappers (integration of 3rd party packages) are available.
  - On the other hand, qooxdoo users typically need fewer 3rd parties, because it tries to be a complete UI framework, much like Qt in the C++ world.
  - qooxdoo comes with conventions and a framework, making Javascript more like object-oriented languages (like C++), supporting well- structured code and lending itself to large web-based applications.

Conclusions

This is a comparative table with the pre-selected frameworks and some key-points

| | vue | react | qooxdoo | |----------|----------|----------|----------| | **License** | MIT licensed | MIT licensed | MIT licensed | | **Popularity** |

Released in 2014

75000 stars in Github

12000 questions in Stack overflow

```
| <ul><li>Released in 2013</li><li>83000 stars in Githu
  b</li><li>65000 questions in Stack overflow</li></u
  l>                                                  | <ul
  ><li>Released in 2009</li><li>600 stars in Github</
  li><li>1000 questions in Stack overflow</li></ul>

  |
```

| **Pros** |

Mix between Angular and React

HTML, CSS and logic nicely separated

```
| <ul><li>Facebook supports it. One of th
  e most popular Front-End JavaScript f
  ramework</li><li>Lot of dedicated and
   easy to install 3rd party libraries
  | <ul><li>Desktop-like web applicatio
  n</li><li>Object Oriented programming
   model</li><li>Lots of native 3rd par
  ty libraries available</li><li>[three
  .js](https://threejs.org/) used direc
  tly</li><li>Complete control over dep
  endencies |
```

| **Cons** |

No specific company supporting it

Not many dedicated 3rd party libraries available

Lack of control over dependencies

|

HTML, CSS and logic not very well separated

Lack of control over dependencies

```
                                        | <ul><li>Not a big community behind</li><li>Some 3rd
                                            party libraries do not integrate very well</li
                                          ></ul>

                                          |
```

The main difference between vue, react and qooxdoo is that the first two provide very light frameworks and programming concepts with simple skeletons to start building on top of, while qooxdoo provides a mature framework with a collection of UI widgets. That means that for vue and react one needs to look for all the UI components that one may want to add and subsequently needs to install them, using the Node Package Manager, together with their dependencies. In qooxdoo , instead, if the original framework doesn't provide the component, you need to download the desired package and keep it within the project.

Based on our review we found that, taking into account that it is easier to build desktop-like applications and the native Object Oriented programming model, qooxdoo is the most suitable framework to implement SIM-CORE's front-end.

Web frameworks

For the back-end, the server-side of the web application, the pre-selection reduced the review to three web-frameworks implemented in different programming languages: express/nodejs in *javascript*, wt in *C++* and flask]/sanic in *python*.

All these web-frameworks were pre-selected since they provide a minimum set of features (included or within packages), namely:

- routing URLs to handlers
- interacting with databases
- support user sessions and authorization
- templating system to render data (e.g HTML, JSON, XML, ...)
- security against web attacks
- communication with client-side (e.g. web-sockets, ...)
- can serve at least one of the pre-selected front-end technologies

The following sections summarizes the reviews and highlights aspects that were found useful to support the final recommendation to implement SIM-CORE.

Javascript express/nodejs

express is probably the smallest web framework on nodejs, and therefore written in javascript. It is a lightweight and unopinionated framework which, in conjunction with the large amount of compatible modules available to extend it, makes it an interesting candidate for this review.

- **Language**: node.js, javascript
- **Popularity**: Github score 92, Stack-overflow score 83
- **License**: MIT license

Pros:

- *Productivity*:
  - Generally very high since the package manager npm solves does most of the heavy lifting in many stages of development: initialization, testing, deployment ...
  - Same language as in client sides
- *Functionality*:
  - Large amount of packages for virtually everything!
  - State-of-the-art package managers as npm, yarn or bower can be used to install and handle effectively module dependencies

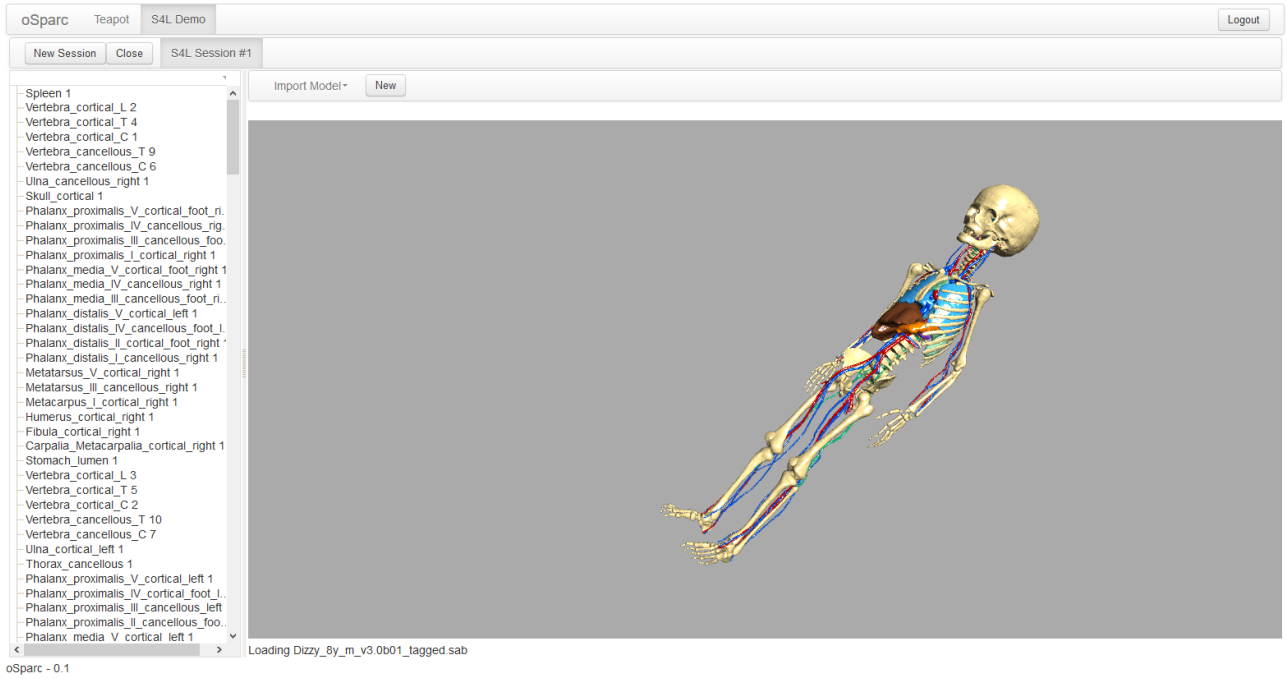
Cons:

- Javascript has low level of integration with python/C++ code.
- Javascript does not have an official built-in or standard-like library like other languages but instead a plethora of *de facto* modules.
- Found that the complexity of module dependencies makes sometimes it very difficult to track down and solve issues.

C++ wt

wt cannot be *strictly* defined neither as a client nor a server-side tech, but rather a C++ library to develop web applications that partially run on both sides. This solution allows writing web GUIs in C++ using a widget abstraction. This paradigm is traditionally used in GUI programming in desktop applications, as in Sim4Life, which makes it very convenient. wt also permits the integration side-by-side of third-party javascript libraries like threejs, to handle the 3D rendering at the client-side. The same type of integration is expected from the client technologies described in previous sections.

The figure shows a prototype website rendering a part of a 3D human model from the ViP family using this technology. More details can be found later in the demonstrations sections.



- Language: C++
- Popularity: Github score 56, Stack-overflow score 40
- License: Dual: GNU General Public + commercial

Pros:

- C++ and widget abstraction allows high reusability and compatibility with existing code-base
- Abstracts request handling and UI rendering
- Integrates session management and lifetime: every user has its own application object and deployment models for with dedicated/shared processes per session.
- Can integrate side-by-side other third-party javascript libraries, e.g. threejs
- C++ integrates very well with other scripting language like python (see boost.python) …), etc
- Other advantages inherent to the C++ language: type safety, speed, support to concurrency (multi-threading, coroutines, …

Cons:

- Heavy lifting to get containers build and run. Build and deploy C++ applications can be time-consuming and complex compared to scripting languages-based libraries. Containerization of the development environment (ie. compilers, …) might lighten this inconvenient but it is definitively more demanding than any other solution based on javascript or other scripting language.
- Not clear separation between server-client. The documentation shows that this is intentionally avoided by design, but it can become an issue when the target is to clearly control the responsibility of each side (e.g. to reduce communication between both sides). First trials shows a high level of communication with the server even for simple operations that could be easily delegated to the client side.
- License scheme makes it incompatible with MIT license, which is the desirable scheme for this platform.

Python flask/sanic

flask is a relatively new framework written in python. It takes advantage of modern features of the language to offer an easy-to-code, lightweight and unopinionated web framework in python. Asynchronous requests are not supported out of the box but new built-in modules in python 3 like asyncio or brand new frameworks as sanic overcome this important drawback.

- **Language**: python 3
- **Popularity**: Github score 91, Stack-overflow score 77
- **License**: BSD/MIT license

Pros:

- *Productivity*: light-weight complete web-framework.
- *Functionality*: standard web-framework
- Coroutines/asynchronous APIs with sanic or asyncio
- Python is a very popular and rich language with extensive built-in library
- Python has a strong integration with C++ (pyd modules)

Cons:

- As all lightweight and unopinionated frameworks, the responsibility to add standard features (via 3rd party or in-house modules) heavily depends on the dev-team with the consequent risk of accumulating technical debt as the codebase grows.

Conclusions

Based on our review, the selected web-framework candidate is flask/sanic in **python**. express could be used to prototype or as a dummy server to test services or APIs. wt is not recommended, nonetheless provided the large amount of modules already available in C++, the evolution of the framework should be monitored in future extensions.

Communication and Interoperability

In order to achieve interoperability among different services, both within the SIM-CORE platform (e.g. among computational services) and outside (e.g., with other SPARC-CORE services), a stable and maintainable communication model is required. This interprocess communication (IPC) can be established using different technologies. Services can communicate synchronously with a request/response model (REST, Apache Thrift), or they can use asynchronous message-based communication mechanisms as the Advanced Message Queuing Protocol (AMQP). For this review we focused on three different technologies:

- RESTful
- RPC with Apache Thrift
- AMQP

RESTful API

Almost always based on the simple and familiar HTTP protocol and widely used today. REST uses the HTTP verbs to get or manipulate resources that are represented using a URL.

Pros

- Simple protocol
- API easily verifiable from within a browser or command line
- Does not require a broker

Cons

- Synchronous communication only, server must always respond
- client/server must be up and running at the same time
- URL for every service must be known

RPC Apache Thrift

A framework that supports multiple language for clients and servers by using a compiler that auto generates code from interface definitions. Supports C++, Java, Python, Node.js, ..., and is developed by facebook.

Pros

- Synchronous and asychronous communication possible
- Easy integration into already existing code
- Very fast and efficient

Cons

- More work to be done on the client side

AMQP

Protocol for asynchronous message based communication. Allows clients to communicate with each other via messages using an intermediate broker. Communication can be done point-to-point or one-to-all.

Pros

- Decoupling of client/server, client does not need to know server's whereabouts
- Messages are buffered, client/server do not need to be up simultaneously

Cons

- More parts in the system (broker)
- More work needs to be done for request/response interaction (identification system for messages)
- If broker dies, system breaks down

Conclusions

All above mentioned technologies have characteristics that satisfy the communication needs for different parts of the SIM-CORE platform. Our choice is the RESTful APIs for the backend-director communication as well as for communication with the other COREs. For the computational backend the use of messaging for job distribution and REST/RPC for communication with computational services is foreseen.

Introduction

The computational backend involves all services needed to handle the actual computational workload. A computational workflow is described as a pipeline that processes a stream of data in a sequential way. Every pipeline consists of multiple algorithms, each one expecting specific input data and providing specific output data. The pipeline can be built up in the frontend as a directed acyclic graph (dag) where the edges describe input/output and the nodes consist of the algorithms, i.e. the computational kernels. Such kernels include complete standalone solvers, algorithms to calculate specific quantities, or a viewer that renders data into graphs, plots or tables, etc.

Responsibilities

The computational backend:

- schedules the execution of pipelines in an efficient way while respecting the inherent data dependencies
- provides the user with a list of all available algorithms
- provides a mechanism to easily inject new algorithms
- allows control/managing of concurrently running pipelines
- can be dynamically up/down-scaled depending on the current load
- has access to a database with all relevant input and output data

Selection of technology for computational kernel integration

Since a central purpose of the SIM-CORE platform is to allow users to add user-defined algorithms, the technology preselection was based on the criterion that xxx.

Modern scientific libraries and solvers span a broad range of programming languages, are typically very specialized, have many dependencies to numerical libraries, and usually are designed to work best on few, specific platforms. In order to ease the deployment of services consisting of such codes into the heterogeneous SIM-CORE platform, it is desirable to provide contributors with the toolsets and platforms they know best. This can be achieved using containers or virtual machines. Due to the large overhead in terms of hardware consumption, usage of virtual machines has been discarded in favour of the containerized approach. Containers, in contrast to virtual machines, do not emulate the hardware but the operating system itself. This makes them much more lightweight and allows for up to thousands of instances running simultaneously on one host.

There are several approaches to containerization. However docker has become the de-facto standard in industry and academia, and many scientific applications already provide users with docker images of their code. Furthermore, it is possible to use the docker tool that natively allows the orchestration of multiple docker containers among a heterogeneous network of computers. Additionally, all major cloud providers support the technology. If during the future evolution of the platform, more sophisticated means of orchestration are required, there exists the possibility of using kubernetes, which is the major player when it comes to managing containers and for which docker has recently added full support.

The docker framework also allows easy functionality extension on existing images, which will be used to enhance algorithms with an additional layer that makes integration into the SIM-CORE ecosystem possible. A specific use case will be discussed below.

Core components of computational backend

Docker image registry

Considering the technology decision outlined above, another core component of the docker ecosystem is being used for the computational backend, namely the concept of the docker registry. Every computational service is provided as a docker image in a repository that is part of the SIM-CORE platform. These images are being pulled from the registry when required, and a container is created to run (execute) the corresponding service.

In addition to the images themselves, the registry also contains meta- information for the services. This allows to store information such as:

- required input data (format)
- output data (format)
- specific hardware needs (GPU/multicore)
- version number and hashes for identification

This data is being used to check whether two algorithms in the pipeline can be connected or not.

Director

The director acts as bridge between the frontend/backend and the computational backend. It is aware of all available algorithms in the registry and can translate incoming pipelines into workflows and schedules jobs to execute them in the proper order.

All jobs are being kept in a queue and its status can be queried by the client. Job control such as stop/kill/resume is also provided.

Distributed task queue and message broker

All jobs in the platform are being scheduled in a centralized queue based on message passing. Workers can grab tasks from the list and execute them concurrently. For that purpose a broker service that handles all the message passing from the director to the worker is also part of the computational backend. Due to its popularity and wide usage, the celery library has been chosen for the distributed task queue. It is easy to integrate and offers bindings to several languages. It also supports several message brokers and database backends. For the initial prototyping, RabbitMQ is chosen as message broker and MongoDB as database.

Workers

Workers are the services that perform the actual computation. They always appear as pairs of containers: a sidecar and an actual computational service. The sidecar is always alive and is connected to the tasks queue. When required it creates a so-called one-shot container that runs the requested computational service. All the sidecar-computational service interactions happen through the command line interface. Furthermore, since being physically on the same host, the side care and computational service share the filesystem, which allows the sidecar to make input files or other data available to the computational service.

The advantage of this design is that all complex interaction with the system is being abstracted away from the computational service and enables contributors to add algorithms without the need for detailed knowledge of the platform.

Service Orchestration

As mentioned above, SIM-CORE takes advantage of the native docker orchestration tool swarm. If more flexibility is required in the future, it will be possible to use kubernetes to support orchestration.

Example use case

TODO: Explain what is coming here. Motivation… and conclude at the end.

For the sake of simplicity, consider a computational service that evaluates a user defined single variable function in a given interval and a second service that renders that result as a scatter plot. For the function parsing service, C and C++ code is available from a contributor. In addition, the contributor specified the command line arguments for its algorithm. For the visualization part, a default service from the SIM- CORE platform will be used that expects a tab separated list of values as an input and creates an rendered html page with a scatter plot.

Dockerfile

A docker-file contains all commands needed to create a docker image that can be run in a container. For the function evaluator, this file looks as follows:

```
1  FROM alpine
2
3  MAINTAINER  Manuel guidon <guidon@itis.ethz.ch>
4
5  RUN apk add --no-cache g++ bash jq
6
7  WORKDIR /work
8
```

```
  ADD ./code /work
  ADD ./simcore.io /simcore.io
  RUN chmod +x /simcore.io/*

  ENV PATH="/simcore.io:${PATH}"

  RUN gcc -c -fPIC -lm tinyexpr.c -o libtiny.o
  RUN g++ -std=c++11 -o test main.cpp libtiny.o
  RUN rm *.cpp *.c *.h
```

The image is based on a very small Linux distribution called `alpine` with compilers `gcc`, shell `bash` and jason parser `jq`. In addition, to compile the source code into an executable called `test` the `PATH` is being prepended by some scripts from what is called `simcore.io`. This allows to enhance the docker command line interface (cli) by whatever is needed to run the computational service via the sidecar. In this case, there is a `run` command added to the cli. Xxx last sentences are hard to understand xxx.

```
  #!/bin/bash

  arg1=$(cat $INPUT_FOLDER/input.json | jq '.[] | select(.name =="xmin")
.value')
  arg2=$(cat $INPUT_FOLDER/input.json | jq '.[] | select(.name =="xmax")
.value')
  arg3=$(cat $INPUT_FOLDER/input.json | jq '.[] | select(.name =="N")
.value')
  arg4=$(cat $INPUT_FOLDER/input.json | jq '.[] | select(.name =="func")
.value')
  temp="${arg4%\"}"
  temp="${temp#\"}"
  arg5=$OUTPUT_FOLDER/output

  ./test $arg1 $arg2 $arg3 $temp $arg5 > $LOG_FOLDER/log.dat
```

In this case, the sidecar would copy the all input data needed into a file called `input.json` which the above script would parse and pass to the test executable.

After building, the docker image is deployed into the docker registry with the following meta data:

```
{
  "input":[
    {
      "name": "N",
      "value": 10
    },
    {
      "name": "xmin",
      "value": 0.0
    },
    {
      "name": "xmax",
      "value": 1.0
    }
    {
       "name": "func",
       "value" : "sin(x)"
    }
  ],
  "output": tsv
}
```

Finally, the descriptor for this part of the pipeline would look like

```
   {
     "input":
```

```
    [
      {
        "name": "N",
        "value": 10
      },
      {
        "name": "xmin",
       "value": -1.0
     },
     {
       "name": "xmax",
       "value": 1.0
     },
     {
       "name": "func",
       "value": "exp(x)*sin(x)"
     }
   ],
   "container":
   {
     "name": "simcore.io.registry/comp.services/function-parser",
     "tag": "1.1"
   }
}
```

The above scripts and file descriptors represents the current state and steps during the technology evaluation process. It needs to be clarified to what degree this can be simplified for integration of SPARC/3rd party computational services, or assigned to a supporting entity (e.g., IT'IS support within SPARC), or facilitated through increased automation within the SIM-CORE platform (resulting in additional development effort).

Miscellaneous

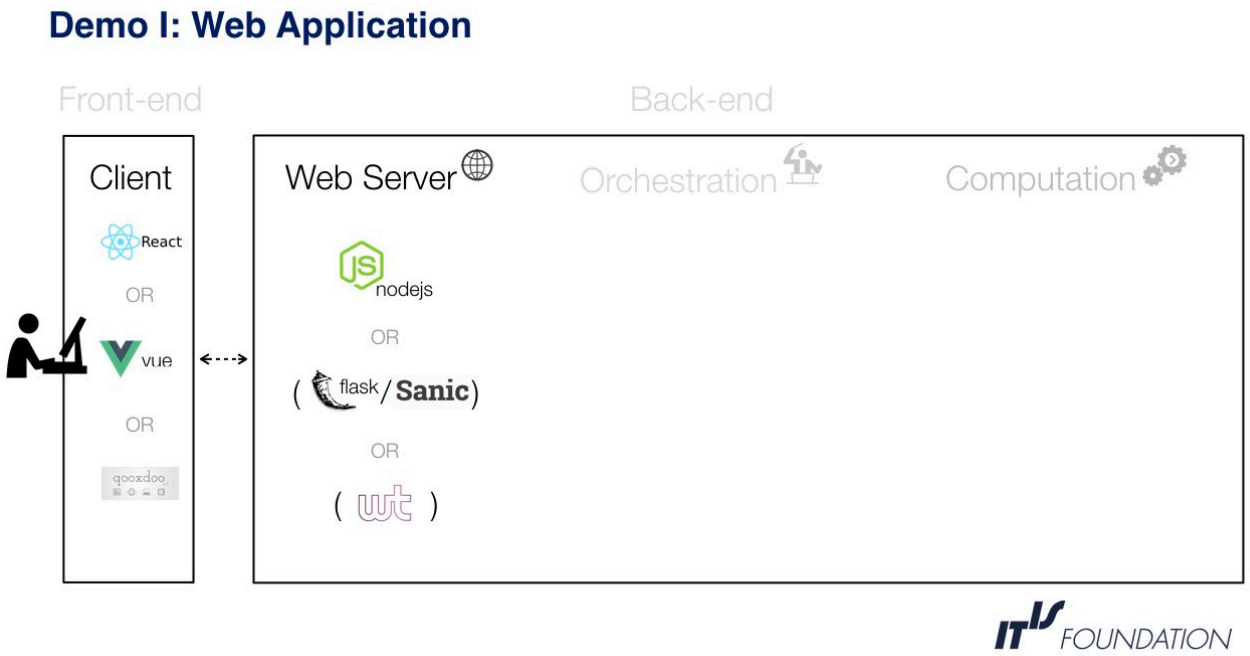
- By the end of 2016 Microsoft added support for docker containers on the Windows family of operating systems. Since docker is operating system agnostic, the SIM-CORE platform automatically supports Linux- and Windows-based computational services.
- Shifter, a new open source project provides a runtime for container images and is specifically suited for HPC on supercomputer architecture. Among other formats it supports docker.
- The MPICH application binary interface (ABI) can be used to link code against the ubuntu MPICH library package and change the binding at runtime to the host ABI compatible MPI implementation. And this is important, because….

Prototypes

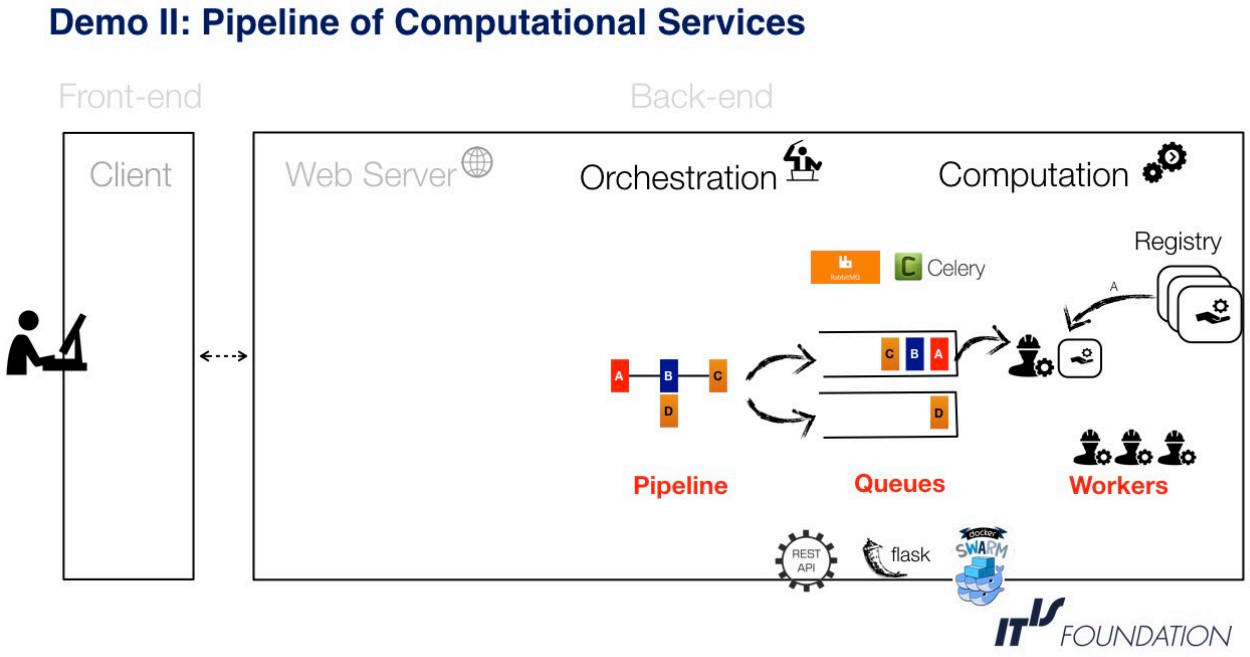
All prototypes and proof-of-concepts created during the review process can be found under the demos folder in the IT'IS GitHub repository. Every subfolder contains a sample program that demonstrates a technology and/or a programming concept. More details about each case is documented in place.

Demonstrators

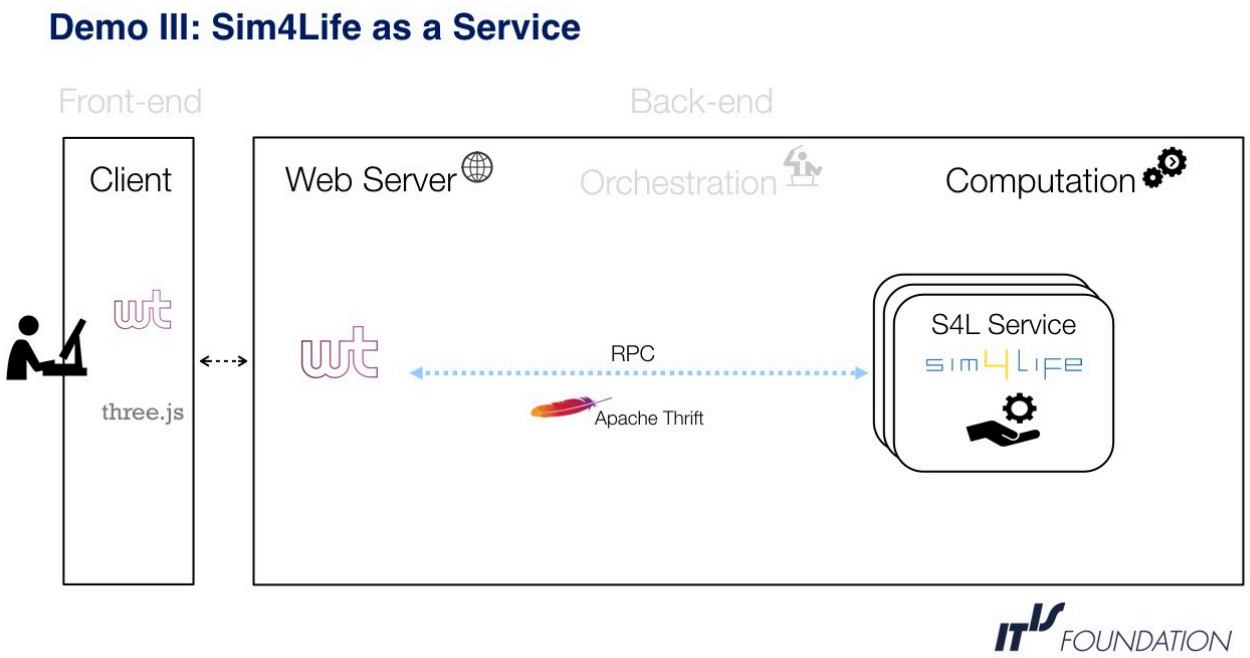
In order to demonstrate the different technologies in a wider context, we have designed three user stories that cover the entire platform front to back.



The first prototype demonstrates a web-application with dynamic content. The client-side implements GUIs with similar features in the three different reviewed technologies and the server-side demonstrates different frameworks to serve these apps. All three front-ends include a 3D viewer, a workbench to schedule task pipelines, a dynamic display of the settings for a given computational service (embedded in service meta- data) and several viewers for results (e.g. 2D plots, tables, etc).



This prototype demonstrates the orchestration and execution of computational services in the back-end. The demonstration starts with a pipeline of tasks that gets scheduled in a queueing system. Every task in the queue is subsequently executed by a worker service that pulls and runs the actual computational service from a docker registry. The worker is implemented as a side-car in order to leverage the computational service from any direct interaction with the platform.



This demonstration builds a web application that offers the entire Sim4Life application as a service. The user can create multiple sessions and get a full anatomical model from the ViP rendered in the browser. To achieve this, the web server communicates with the Sim4Life service via RPC and requests to load, process and transmit the model in chunks. When the information reaches the client-side it is rendered using threejs.

NOTE: These three demonstrations were presented live during the webex meeting with the SPARC Subject Matter Experts on December 13, 2017.

The technology evaluation starts defining the logical architecture of the SIM-CORE framework and establishes a pre-selection of technologies for each logical modules and concepts of interest. Proof-of-concepts and/or full functional prototypes with each technology were built and analyzed during the review process (see `demos` folder in osparc-lab repository at github). The main purpose was to build a more practical opinion on the pros/cons of each option.

Finally, a set of three comprehensive demonstrators were built and presented live during the teleconference with the SPARC Subject Matter Experts on December 13, 2017.

This review concludes with recommendations on technologies to use for different part of SIM-CORE. qooxdoo was selected as the most suitable framework for the front-end. For the web server is flask , a python web framework, the preferred technology. The computational services shall be encapsulated in docker containers and orchestrated using swarm. The communication between services shall be accomplished with apache-thrift or REST-API. The former is the preferred method for internal services while the latter shall be used to interact with the API of external services.

- **User Interface**: The user interface shall primarily be a browser- based GUI. It shall be user-friendly and interactive. It shall support interface elements that permit setting parameters and properties, visualizing 3D scenes, displaying data (tables, plots, trees, image-data, etc.) and messages, setting up and displaying work-flows ("workbench"), monitoring and administering computational resources, as well as accessing and managing data (local data and data in the DAT-CORE). With lower priority and/or at a later stage, the user interface should also offer a scripting interface (Python-based). Through the GUI, users shall be able to access data and models (including search functionality), manage access rights, set up studies and workflows, execute and monitor services locally and in the cloud, as well as perform analysis and post-processing. Xxx workbench/flowchart editor xxx
- **Services Infrastructure**: Service environments that make it easy to run Python scripts, to run executables (Unix-based operating systems, such as Linux, are currently the only platforms used by contacted SPARC teams developing their own solvers/executables) that take parameters from the command line or standardized input file formats, to compile and execute C, C++, and Java code, to execute R scripts, and xxx shall be provided. It shall be easy for users to register new services by using one of the predefined service environments and inserting their executables/code/… and it must be simple to specify the required input parameters such that they are exposed through the online GUI. By providing information about the generated output, it shall be possible to present the output using standardized viewers (plots, 3D views, slice field views, isosurface views…). It shall be possible to link services with compatible outputs and inputs into pipelines. Advanced users shall have the possibility of creating services that manage dedicated sections of the online GUI, allowing them superior control also over visualization and interaction. When executing a service, all the information required to reproduce the execution (input data, parameters, version of service…) shall be stored, to facilitate reproduction.
- **Specialized Services**: A range of commonly required services shall be provided. On the physics solver side, this includes primarily electromagnetic exposure. Typically, the most required variant of electromagnetic solver is an electroquasistatic solver (for exposure through electrodes) xxx with support for anisotropic electrical conductivity (to handle the important impact of anisotropic neural tissue) xxx. Another relevant electromagnetic solver is the magnetoquasistatic solver (for exposure by coils and loops). Selected teams will also require acoustic propagation, biomechanics, optics, and tissue damage solvers (not all of these can be provided by IT'IS). The platform must support either the import of discretized models (meshes, voxels), or provide discretization services (voxeler, mesher), or both. On the electrophysiological modeling side, the most commonly required solver that shall be supported is NEURON from Yale. Predefined, diameter- parameterized fiber models shall be provided, which must include the MRG model, and shall be assignable along user-defined trajectories. There is large request for an unmyelinated fiber model, but it is currently unclear, which model is suitable. It shall also be possible to load and simulate complex neuron models, defined as .hoc files. It must be possible to couple electromagnetic exposure conditions with electrophysiological neuron models. Metamodeling services that can vary parameters of services or pipelines to perform optimization tasks, assess sensitivity, propagate uncertainty, and perform model order reduction shall be provided. In addition, a specialized optimization functionality to find optimal stimulation parameters (currents, pulse-shapes) for multi-contact electrodes that produce selective stimulation of fibers is required. A Python service with libraries such as numpy, scipy, pyplot/plotly that provide users with analysis and post processing functionalities akin to Matlab shall be offered (Supporting Matlab within a dedicated service could be problematic at the current time, due to licensing issues in combination with Docker technology. Octave could be a highly compatible alternative and Python with suitable extensions also provides similar functionality to Matlab.). The creation of a SBML/CellML service to support the MAP-CORE vision is desirable. A coupling service shall allow setups involving more advanced execution schemes and data exchange than simple pipelining, within the limitations imposed by latency and bandwidth. This shall include iterative coupling and bidirectional coupling. A service allowing (ontological) annotation of anatomical and physiological models will be provided by the MAP-CORE.
- **Models**: The platform shall offer anatomical models of humans (man and woman) and rat. We choose a rat because rats were most popular in the vote conducted via Slack, and because it is anticipated that imaging rats and identifying selected peripheral nerves will be easier for rats than mice, because of their size, resulting in superior model quality. Further animal models that shall be considered at a later point include mice, cats and a monkey. The anatomical models shall be pre-functionalized with the trajectories of major peripheral nerves. Through a service, it shall be possible to obtain for simulation purposes discretized representation of the models or sub-regions thereof. It shall be possible to register additional layers of information relative to the anatomical models. These include measurement and simulation data, image data, nerve trajectories, anatomical detail, device models, as well as electrophysiological models.
- **Computational Infrastructure**: The platform shall initially run on HPC hardware localized at IT'IS. Subsequently, the platform shall run on the Amazon cloud and be flexibly scalable, depending on user

demand. It should be considered to add support for execution of selected services (e.g., NEURON simulations) on dedicated high-performance computing resources (e.g., NSG portal).

- **Image-based Modeling**: Functionality shall be available that permits to segment image-data and convert it into anatomical models. It shall be facilitated to generate nerve models from histological images/information, trajectories, and knowledge about fibre type and property distributions. Morphing functionality for applying transformation fields to anatomical model and thus parameterize their geometry shall be provided.
- **Data storage and annotation**: It shall be possible to store models, services, studies, results, and additional data in the DAT-CORE, the cloud, or locally. It shall be possible to enhance data with meta- information, such as links to related information (e.g., links to documentation and validation data), ontological descriptors, or quality certification. It shall be possible to relate data to points or regions within specific anatomical models. Data shall be referenceable (unique identifier, linkable) and traceable (origin of data, versions).

TODO Recommendations on tech

Requirements and specifications have been established based on interviews with SPARC teams as well as based on the results of the technology evaluation. The requirements are very much aligned with prior expectations and only minor adaptation of the proposal and possibly a change in prioritization is suggested. The technology evaluation has resulted in a clear idea of the technologies and approach that should be applied to implement the SPARC SIM-CORE platform. The design of the platform will enable maximal flexibility with regard to the wide variety of existing and envisioned user-generated modeling services. It will also offer user-friendly interfaces, allowing users to engage on different levels with the platform, ranging from simple execution of existing models to the advanced generation of services with fine-grained control over dedicated user-interface elements, depending on expertise. Due to the modularity of the chosen approach, it will be simple to extend and adapt the platform at later time-points, and it will be feasible to revisit some of the choices as technology evolves and to replace layers or components of the implementation without the need for a complete redesign of the platform.